SANDIA REPORT

SAND2013-3181
Unlimited Release
Printed April 2013

Supersedes SAND2012-10087
Dated November 2012

The Portals 4.0.1 Network Programming Interface

Brian W. Barrett, Ron Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler,
Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson

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

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation,
a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's
National Nuclear Security Administration under contract DE-AC04-94AL85000.

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

SAND2013-3181
Unlimited Release
Printed April 2013

Supersedes SAND2012-10087
dated November 2012

The Portals 4.0.1 Network Programming Interface

Brian W. Barrett
Ron Brightwell
Kevin Pedretti
Kyle Wheeler
Scalable System Software Department

Scott Hemmert
Scalable Computer Architecture Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
{bwbarre, rbbrigh, ktpedre,
kbwheel, kshemme}@sandia.gov

Rolf Riesen
IBM
rolf.riesen@ie.ibm.com

Keith Underwood
Central Architecture and Planning
Intel Corporation
P.O. Box 5800
Albuquerque, NM 87185-1319
Keith.D.Underwood@intel.com

Arthur B. Maccabe
Computer Science and Mathematics
Oak Ridge National Laboratory
Oak Ridge, TN 37831
maccabeab@ornl.gov

Trammell Hudson
c/o OS Research
114 Pierrepont Street #5
Brooklyn, NY 11201
hudson@osresearch.net

Abstract

This report presents a specification for the Portals 4.0 network programming interface. Portals 4.0 is intended to allow scalable, high-performance network communication between nodes of a parallel computing system. Portals 4.0 is well suited to massively parallel processing and embedded systems. Portals 4.0 represents an adaption of the data movement layer developed for massively parallel processing platforms, such as the 4500-node Intel TeraFLOPS machine. Sandia's Cplant cluster project motivated the development of Version 3.0, which was later extended to Version 3.3 as part of the Cray Red Storm machine and XT line. Version 4.0 is targeted to the next generation of machines employing advanced network interface architectures that support enhanced offload capabilities.

Acknowledgments

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

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 contributors.

Contents

Appendix

List of Figures

List of Tables

List of Implementation Notes

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 massively parallel processing (MPP) systems, such as the nCUBE-2 and the Intel Paragon, when network bandwidth became comparable to memory bandwidth. An intermediate memory-to-memory copy now meant that only half the available network 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 was used until 2006 on the ASCI Red supercomputer, the first general-purpose machine to break the one teraflops barrier.

Although very successful on architectures with lightweight kernels, such as ASCI Red, Portals 2.0proved difficult to port to Cplant [4] 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 was born [5].

Portals 3.0 evolved over three revisions to Portals 3.3 [18]. In the interim, the system context has changed significantly. Many newer systems are capable of offloading the vast majority of the Portals implementation to the network interface. Indeed, the rapid growth of bandwidth and available silicon area relative to the small decrease in memory latency has made it *desirable* to move latency sensitive tasks like Portals matching to dedicated hardware better suited to it. The implementation of Version 3.3 on ASC Red Storm (Cray XT3/XT4/XT5) illuminated many challenges that have arisen with these advances in technology. In this report, we document Version 4.0 as a response to two specific challenges discovered on Red Storm. Foremost, while the performance of I/O buses has improved dramatically, the latency to cross an I/O bus has not fallen as dramatically as processor, memory and network performance has increased, negatively impacting target message rates. In addition, partitioned global address space (PGAS) models have risen in prominence and require lighter weight semantics compared to message passing.

Nomenclature

ACK	Acknowledgment.
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.
ASC	Advanced Simulation and Computing.
ASCI Red	Intel TeraFLOPS system installed at Sandia National Laboratories. First general-purpose system to break the one teraflops 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.
LE	List Entry.
MD	Memory Descriptor.
ME	Matching 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 an *atomic* operation, which updates data 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). A node may also act as a *gateway* between networks.
OS	Operating System.
PM	Message passing layer for SCoreD [11].
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.
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 the Portals network programming interface for communication between nodes in a system area network. Portals is designed to provide the building blocks necessary to create a diverse set of scalable, high performance application programming interfaces and language support run-times. The Portals API is designed to support a machine with two million or more cores.

This document is divided into several sections:

Section 1 – Introduction.
The purpose and scope of the Portals API

Section 2 – An Overview of the Portals 4.0 API.
A brief overview of the Portals API, introducing the key concepts and terminology used in the description of the API

Section 3 – The Portals 4.0 API.
The functions and semantics of the Portals API in detail

Section 4 – Guide to Implementors.
A guide to implementors, highlighting subtleties of the standard that are critical to an implementation's design

Appendix A – Portals Design Guidelines.
The guiding principles behind the Portals API design

Appendix B – README-template.
A template for a README file to be provided by each implementation

Appendix C – Summary of Changes.
A list of changes between versions since Portals 3.3

1.2 Purpose

Portals aims to provide a scalable, high performance interface network programming interface for High Performance Computing (HPC) systems. Portals provides an interface to support both the Message Passing Interface (MPI) [14] standard as well as the various partitioned global address space (PGAS) models, such as Unified Parallel C (UPC), Co-Array Fortran (CAF), and SHMEM [9]. While neither MPI nor PGAS models impose specific scalability limitations, many network programming interfaces do not provide the functionality needed to allow implementations of either model to reach scalability and performance goals.

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

- Connectionless – Many connection-oriented architectures, such as InfiniBand [10], VIA [8] and TCP/IP sockets, have practical limitations on the number of peer connections that can be established. In 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. While Portals provides building blocks to enable flow control (See Section 2.7), it is the responsibility of the application to manage flow control. 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 as fast as memory buses, data has to flow directly into user space.

The following are properties of a network architecture that avoid 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 was originally designed for and implemented on the nCUBE-2 machine as part of the SUNMOS (Sandia/UNM OS) [13] and Puma [19] lightweight kernel development projects. Portals went through three design phases [17], with the most recent one being used on the 13000-node (38,400 cores) Cray Red Storm [2] that became the Cray XT3/XT4/XT5 product line. Portals has been very successful in meeting the needs of such large machines, not only as a layer for a high-performance MPI implementation [7], but also for implementing the scalable run-time environment and parallel I/O capabilities of the machine.

The third-generation Portals implementation was designed for a system where the work required to process a message was long relative to the round trip between the application and the Portals data structures. However, in modern systems where processing is offloaded onto the network interface, the time to post a receive is dominated by the round trip across the I/O bus. This latency has become large relative to message latency and per message overheads (gap). This limitation was exposed by implementations on the Cray Red Storm system. Version 4.0 of Portals addresses this problem by adding the building blocks necessary to support the concept of *unexpected messages*. The second limitation exposed on Red Storm was the relative weight of handling newer PGAS programming models. PGAS programming models do not need the extensive matching semantics required by MPI and I/O libraries and can achieve significantly lower latency and higher message throughput without matching. Version 4.0 of Portals adds a lightweight, non-matching interface to support these semantics as well as lightweight events and acknowledgments. Finally, version 4.0 of Portals reduces the overheads in numerous implementation paths by simplifying events, reducing the size of acknowledgments, and generally specializing interfaces to eliminate functionality that experience has shown to be unnecessary.

1.4 Scalability

The primary goal in the design of Portals is scalability. Portals is designed specifically for an implementation capable of supporting a parallel job running on two million processing cores or more. 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 does not inhibit scalability and it does not guarantee it either. No Portals operation requires global communication or synchronization.

Similarly, a quality implementation is needed for Portals to be scalable. A non-scalable implementation, underlying network protocol, or hardware will result in a non-scalable Portals implementation and application.

To support scalability, the Portals interface maintains a minimal amount of state. By default, Portals provides reliable, ordered delivery of messages between pairs of processes. Portals is 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.

1.5 Communication Model

Portals combines the characteristics of both one-sided and two-sided communication. In addition to more traditional "put" and "get" operations, they define "matching put" and "matching get" operations. The destination of a *put* (or send) is not an explicit address; instead, messages target list entries (potentially with matching semantics or an offset) using the Portals addressing semantics 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 allows the *target* to determine whether incoming messages are acceptable. A *target* process can choose to accept message operations from a specific process or all processes, in addition to the ability to limit messages to a specified *initiator* user id.

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 [20], VIA [8], FM [12],

GM [16], PM [11], 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. Portals 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 Portals.

1.7 Faults

Reliable message transmission is challenging in modern high performance computing systems due to system scale, component failure rates, and application run-times. 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 an event indicating the completion of every operation. Completion events indicate whether the operation completed successfully or not.

Chapter 2

An Overview of the Portals API

In this chapter, we provide an overview of the Portals API and associated semantics. Detailed API functions and option definitions are presented in the next chapter.

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 perform an atomic operation on 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 updating a memory location for, and potentially responding with the result from, an *atomic* 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, this is performed by Portals without the explicit involvement of the application. An implementation of Portals may use dedicated hardware, a operating system driver, a progress thread running in the application process, or some other option to generate the reply.

Figure 2.1 shows the graphical conventions used throughout this document. Some of the data structures created through the Portals API reside in user space to enhance scalability and performance, while others are kept in protected space for protection and to allow an implementation to place these structures into host or NIC memory. We use colors to distinguish between these elements.

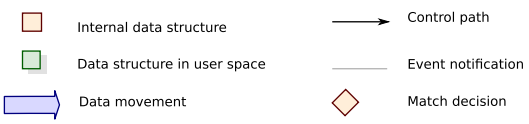


Figure 2.1. Graphical Conventions: Symbols, colors, and stylistic conventions used in the diagrams of this document.

Figures 2.2, 2.3, 2.4, and 2.5 present graphical interpretations of the Portals data movement operations: *put* (send), *get*, and *atomic* (the swap atomic is shown). 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 2.2. The Portals API does not specify a wire protocol. When the data ② has been put into the remote memory descriptor

(or been discarded), the *target* optionally sends an acknowledgment ③ message.

IMPLEMENTATION NOTE 1:

No wire protocol

This document does not specify a wire protocol. Portals requires a reliable communication layer with the semantics and progress rules specified in this document. Implementors are left great freedom in implementation design choices.

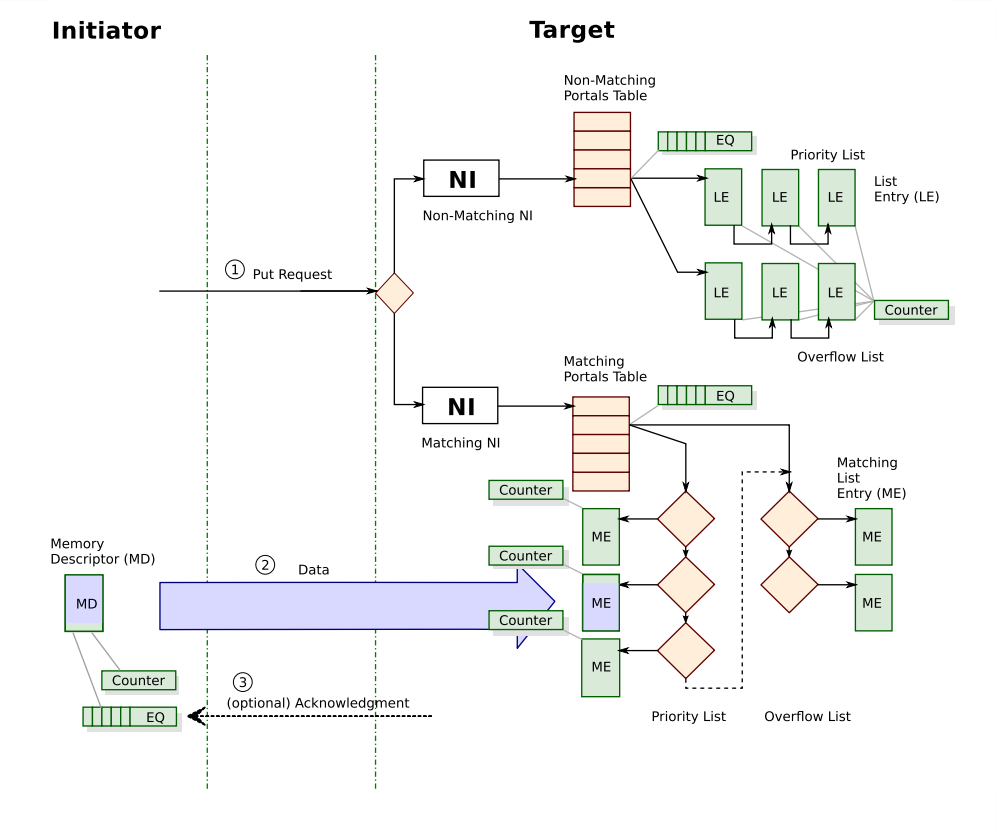


Figure 2.2. Portals Put (Send): Note that the put request ① is part of the header and the data ② is part of the body of a single message. Depending on the network hardware capabilities, the request and data may be sent in a single large packet or several smaller ones.

Figure 2.2 represents several important concepts in Portals 4.0. First, a message targets a *logical* network interface and a user may instantiate up to four logical network interfaces associated with a single *physical* network interface. A portals *physical* network interface is a per-process abstraction of a physical network interface (or group of interfaces). Logical network interfaces may be *matching* or *non-matching* and addressed by either *logical* (rank) or *physical* (nid/pid) identifiers. As indicated in Figure 2.2, separate logical network interfaces have independent resources. The second important concept illustrated in Figure 2.2 is that each portal table entry has three data structures attached: an event queue, a priority list, and an overflow list. The final concept illustrated in Figure 2.2 is that the overflow list is traversed after the priority list. If a message does not match in the priority list (matching interface) or it is empty (either interface), the overflow list is traversed.

Figure 2.2 illustrates another important Portals concept. The space the Portals data structures occupy 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 2 for possible locations of the event queues.

IMPLEMENTATION NOTE 2: Location of event queues and counting events

Note that data structures that can only be accessed through the API, such as counting events and event queues, are intended to reside in user space. However, an implementation is free to place them anywhere it wants.

IMPLEMENTATION NOTE 3: Protected space

Protected space as shown for example in Figure 2.2 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 itself.

Figure 2.3 is a representation of a *get* operation from a *target* that does matching. The corresponding *get* from a non-matching *target* is shown in Figure 2.4. First, the *initiator* sends a request ① to the *target*. As with the *put* operation, the *target* translates the portals addressing information in the request using its local portals structures. Once it has translated the portals addressing information, the *target* sends a *reply*② that includes the requested data.

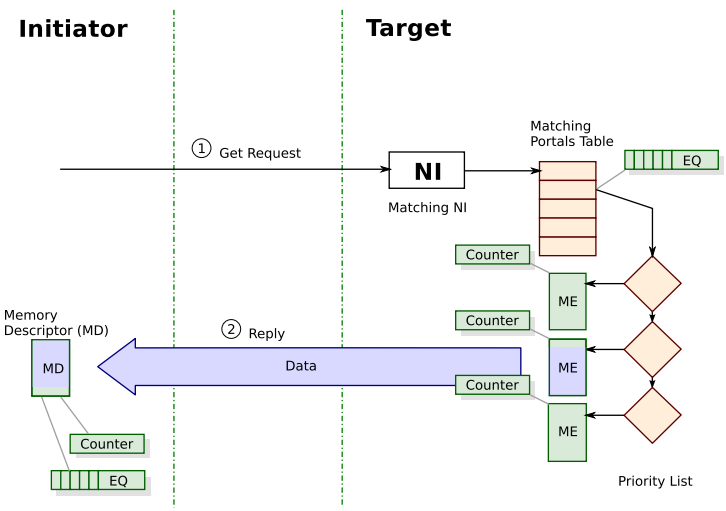


Figure 2.3. Portals Get from a match list entry.

Portals address translation (matching and permissions checks) is only performed at the *target* of an operation. Acknowledgments for *put* and *atomic* and replies to *get* and *atomic* operations bypass the portals address translation structures at the *initiator*. Acknowledgments and replies may only be generated as the result of an action by the *initiator* and therefore do not require the level of protection required at the *target*.

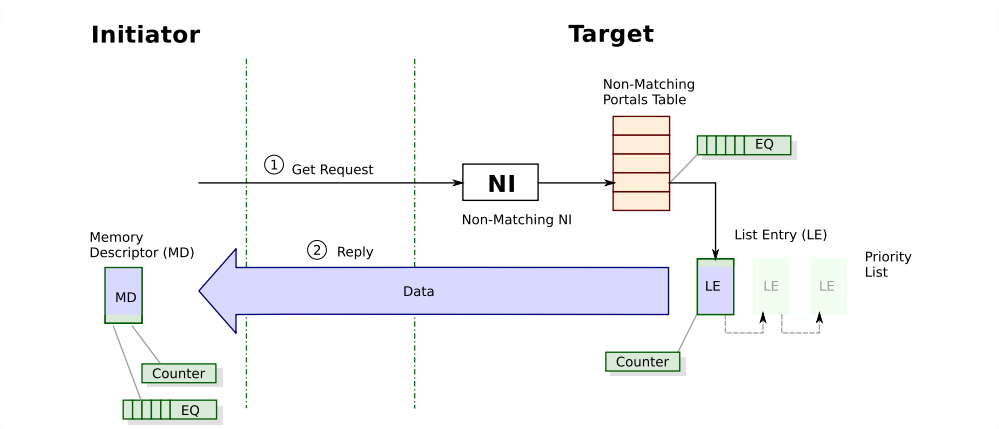


Figure 2.4. Portals Get from a list entry. Note that the first LE will be selected to reply to the *get* request.

The third operation type, *atomic*, is depicted in Figure 2.5 for the swap operation and Figure 2.6 for a summation.

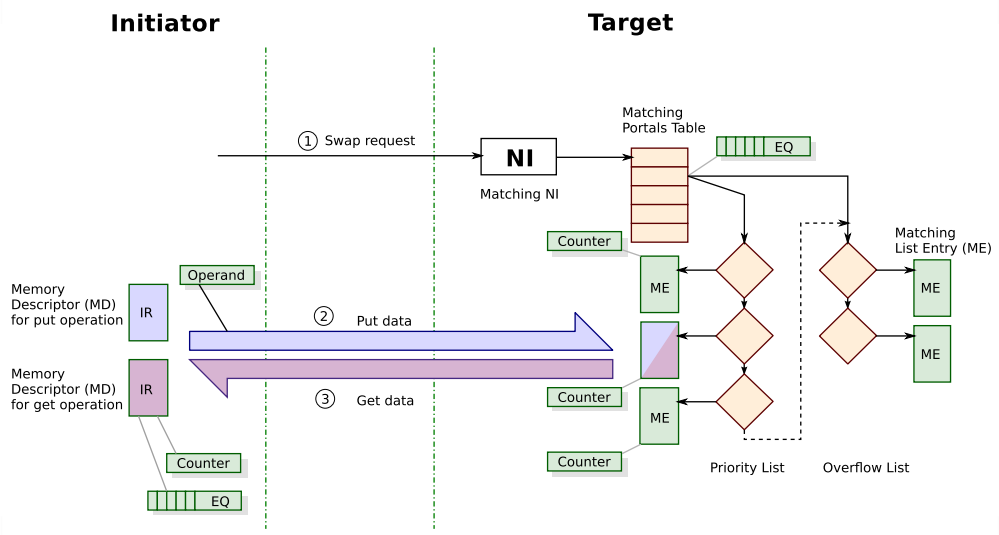


Figure 2.5. Portals Atomic (swap is shown). An atomic swap in memory described by a match list entry using an initiator-side operand.

For the swap operation shown in Figure 2.5, the *initiator* sends a request ①, containing the *put* data and the operand value 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.

The sum operation shown in Figure 2.6 adds the put data into the memory region described by the list entry. The figure shows an optional *acknowledgment* sent back. The result of the summation is not sent back, since the *initiator*

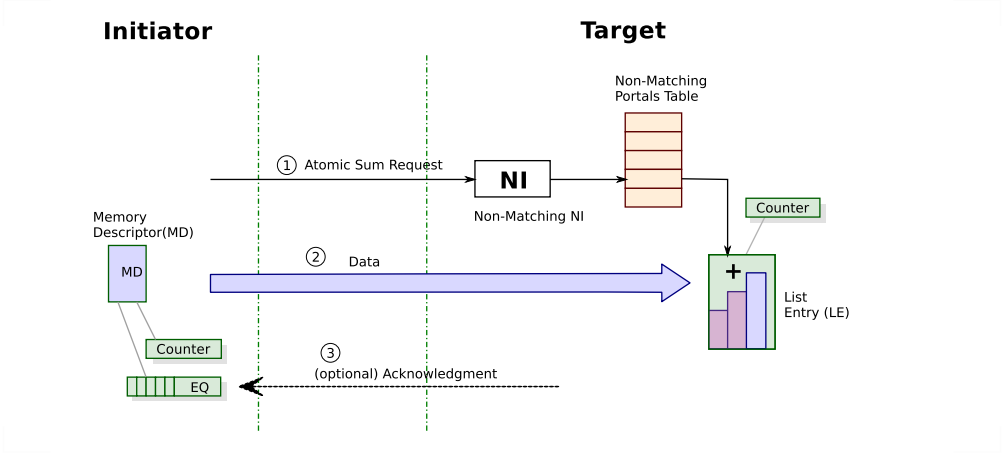


Figure 2.6. Portals Atomic (sum is shown). An atomic sum operation in memory described by a list entry.

used **PtlAtomic()** instead of **PtlFetchAtomic()**.

2.2 Usage

Some of the diagrams presented in this chapter may seem daunting at first sight. However, many of the diagrams show all possible options and features of the Portals building blocks. In actual use, only some of them are needed to accomplish a given function. Rarely will they all be active and used at the same time.

Figure 2.2 shows the complete set of options available for a *put* operation. In practice, a diagram like Figure 2.7 is much more realistic. It shows the Portals structures used to setup a one-sided *put* operation. A user of Portals needs to specify an initiator region where the data is to be taken from, and an unmatched target region to put the data. Offsets can be used to address portions of each region; e.g., a word at a time, and an event queue or a counting event inform the user when an individual transfer has completed.

Another example is Figure 2.6 which is simpler than Figure 2.5 and probably more likely to be used in practice. Atomic operations, such as the one in Figure 2.6 are much more likely to use a single unmatched target region. Such simple constructs can be used to implement global reference counters, or access locks.

2.3 Completion Events

Portals provides two mechanisms for recording completion events: full events (Section 3.13) and counting events (Section 3.14). Full events provide a complete picture of the transaction, including what type of event occurred, which buffer was manipulated, and identifying any errors that occurred. The full event can also carry a small amount of local data and, on the target, a small amount of out-of-band header data. Counting events, on the other hand, are designed to be lightweight and provide only a count of successful and failed operations (or successful bytes delivered). The delivery of events (full events or counting events) may be manipulated when creating a number of other structures.

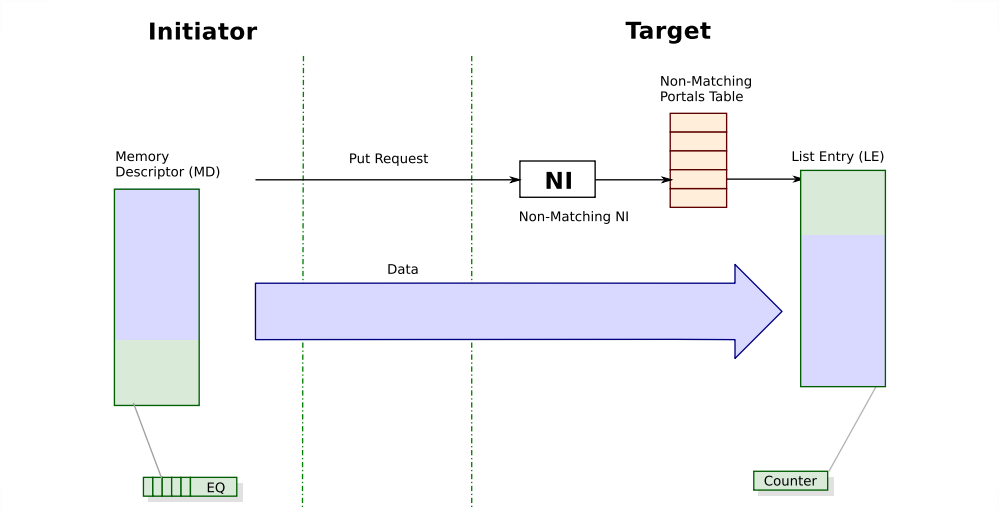


Figure 2.7. Simple Put Example: Not every option or Portals feature is needed to accomplish simple tasks such as the transfer of data from an initiator region to a target region.

2.4 Portals Addressing

One-sided data movement models (e.g., SHMEM [9], ST [20], and MPI-2 [15]) typically use a process identifier and remote address to identify a memory address on a remote node. In some cases, the remote address is specified as a memory buffer identifier and offset. 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.

Portals lists provide one-sided addressing capabilities. Portals list entries serve as a memory buffer identifier that may be persistent or optionally removed from the list after a single use. Traditional one-sided addressing capabilities have proven to be a poor fit for tagged messaging interfaces, such as the Message Passing Interface [6]. To overcome these limitations, Portals also supports match list entries, which include additional semantics for receiver-managed data placement. Matching semantics are discussed in Section 2.4.2.

In addition to matching a pre-posted list entry, an incoming message also must pass a permissions check. The permissions check is *not* a component of identifying the correct buffer. It is *only* applied after the correct buffer has been identified. The permissions check has two components: the target of the message must allow the initiator to access the buffer and must allow the specified operation type. Each list entry and match list entry specifies which types of operations are allowed—put and/or get—as well as a user ID that can be used to identify which initiators are allowed to access the buffer. A failure of the permissions check for an incoming message does not modify the Portals state in any way, except to update the status registers (see Section 3.3.7), and the message itself is discarded.

Figures 2.8 and 2.9 are graphical representations of the structures used by a *target* in the interpretation of a portals address. The initiator's physical network interface and the specified target node identifier are used to route the message to the appropriate node and physical network interface. This logic is not reflected in the diagrams. The *initiator*'s logical network interface and the specified target process ID[1] are used to select the correct *target* process and the logical network interface. Each logical network interface includes a single portal table used to direct message delivery.

[1] A logical *rank* can be substituted for the combination of node ID and process ID when logical endpoint addressing is used.

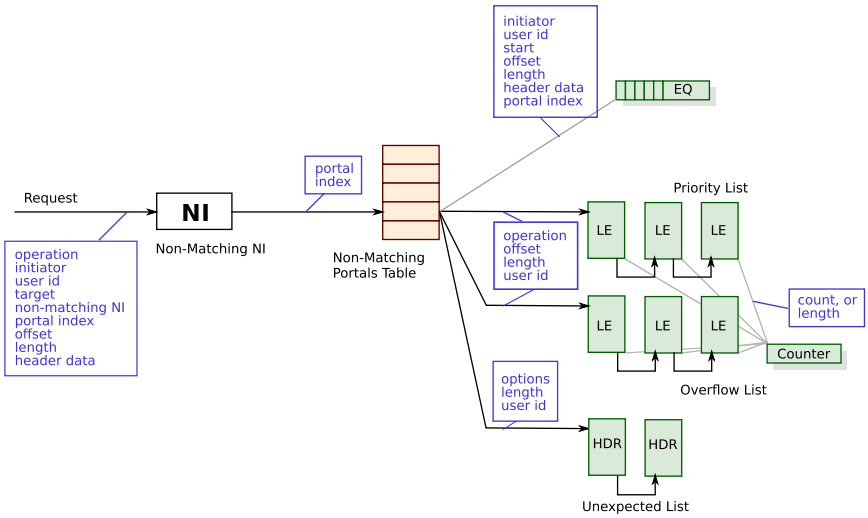


Figure 2.8. Portals Non-Matching Addressing Structures: The example shows the flow of information for a non-matched request at a target. Various pieces of information from the incoming header flow to the Portals structures where they are needed to process the request.

Discussion: *Portals loosely defines the concept of a physical network interface. A physical network interface may be a single hardware network interface or it may represent a collection of hardware network interfaces, with multi-rail support implemented within the Portals implementation.*

For example, in a system like BlueGene/L [1], an implementation may expose a physical network interfaces for the high speed network and another physical network interface for the Ethernet support and I/O network. On the other hand, a system with multiple InfiniBand HCAs may choose to expose a single physical network interface which load balances between the hardware interfaces. In both cases, a portal table will be created for each initialized logical network interface over each physical network interface for each process.

An initiator-specified portal index is used to select an entry in the portal table. Each entry of the portal table identifies three lists and, optionally, an event queue. The priority list and overflow list provide lists of remotely accessible address regions. Applications may append new list entries to either list, allowing complex delivery mechanisms to be built. Incoming messages are first processed according to the priority list and, if no matching entry was found in the priority list, are then processed according to the overflow list. In addition to providing an insertion point in the middle of the combined list structures by allowing insertions at the end of both the priority and overflow lists, the overflow list carries additional semantics to allow unexpected message processing.

The third list that is associated with each portal index is more transparent to the user and provides the building blocks for supporting unexpected messages. Each time a message is delivered into the overflow list, its header is linked into the unexpected list. The user can not insert a header into the unexpected list, but can search the list for matching entries and, optionally, delete the matching entries from the list. Further, when a new list entry is appended to the priority list, the unexpected list is first searched for a match. If a match is found (i.e., had the list entry been on the priority list when the message arrived, the message would have been delivered into that list entry), the list entry is not inserted, the header is removed from the unexpected list, and the application is notified a match was found in the unexpected list. A list entry in the overflow list may disable the use of the unexpected list for messages delivered into that list entry. All unexpected messages associated with a list entry must be handled by posting matching list entries in the priority list or searching and deleting prior to **PtlLEUnlink()** or **PtlMEUnlink()** successfully unlinking the

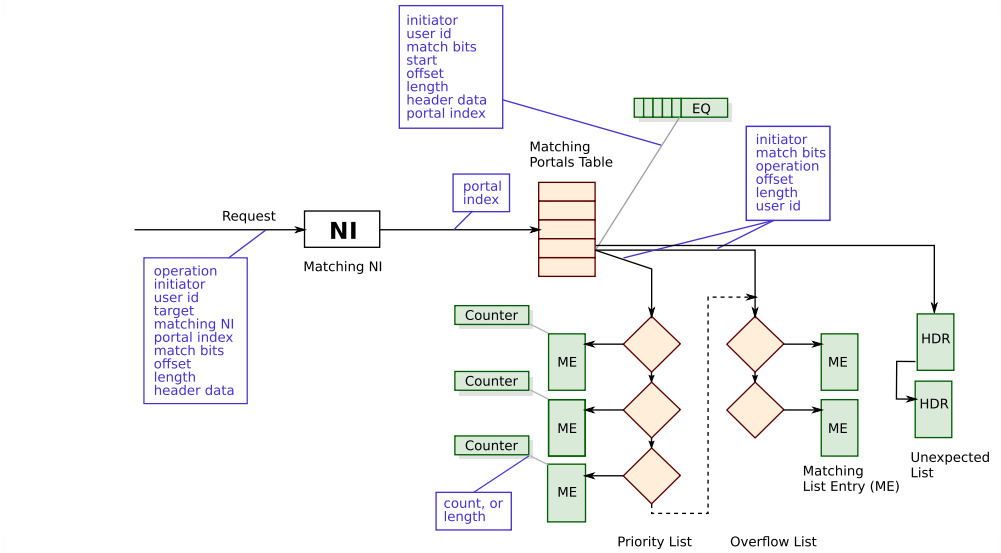


Figure 2.9. Portals Matching Addressing Structures: The example shows the flow of information for a matched request at a target. Various pieces of information from the incoming header flow to the Portals structures where they are needed to process the request.

overflow list entry. Unlike incoming messages, no permissions check is performed during the search of the unexpected queue. Therefore, the user is responsible for ensuring that the overflow list provides sufficient protection to memory and any further permissions checks must be performed by the user based on the overflow event data.

Each data manipulation event (e.g., `PTL_EVENT_PUT`) has a corresponding overflow event (e.g., `PTL_EVENT_PUT_OVERFLOW`) which is generated when a matching header is found in the unexpected list during list entry insertion. The overflow full event includes sufficient information (event type, start address, length, etc.) to determine what operation occurred and where the data was delivered into the overflow list. If the *mlength* in the full event is less than the *rlength*, the message was truncated. It is the responsibility of the application to retrieve the message body, if necessary.

If the incoming message is not delivered into either the priority or overflow list and flow control is not enabled on the portal table entry, the message is discarded and the `PTL_SR_DROP_COUNT` status register is incremented (see Section 3.3.7). If flow control is enabled on the portal table entry, flow control is triggered and a `PTL_PT_FLOWCTRL` full event is generated in the event queue associated with the portal table entry (see Section 2.7).

In typical scenarios, MPI point-to-point communication uses the matching interface and full events, while SHMEM uses the non-matching interface and lightweight counting events. The overflow list may act as either a building block for handling MPI unexpected messages (when the unexpected list is enabled) or as a mechanism for allowing insertion into the middle of a list (when the unexpected list is disabled).

2.4.1 Lists and List Entries

Lists and list entries provide semantics similar to that found in traditional one-sided interfaces. List entries identify a memory region as well as an optional counting event. The memory region specifies the memory to be used in the operation, and the counting event is optionally used to record the occurrence of operations. Information about the

operations is (optionally) recorded in the event queue attached to the portal table entry.

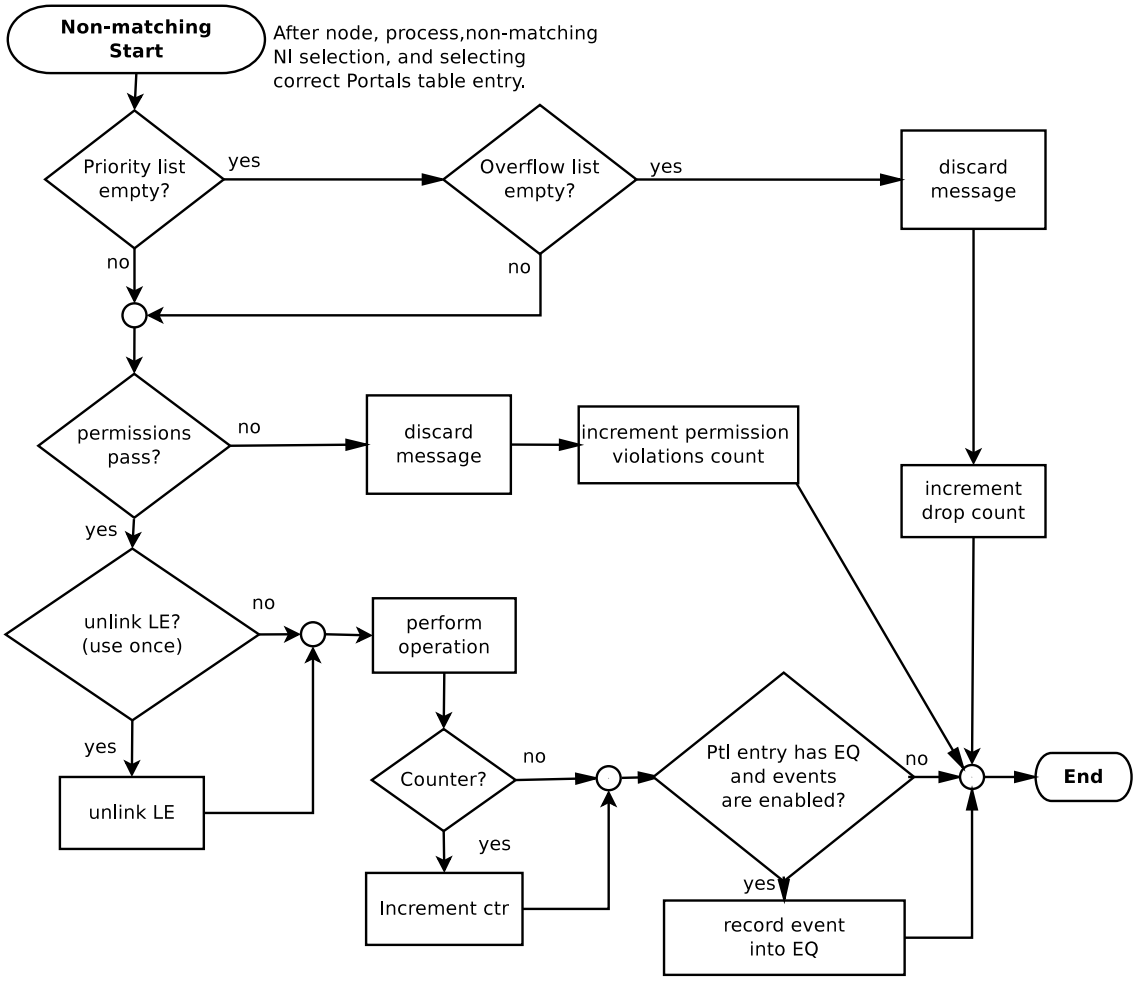


Figure 2.10. **Non-Matching Portals Address Translation.**

Figure 2.10 shows the logical flow of address translation on a non-matching logical network interface. The first list entry (LE) in a list *always* matches. Authentication is provided through fields associated with the LE and act as *permission* fields, which can cause the operation to fail. An operation can fail to fit in the region provided and, if so, will be truncated. Other semantics provided by match list entries—such as locally managed offsets—are not supported. The overflow list is checked after the priority list, if necessary. The non-matching translation path has the same event semantics as a matching interface. The important difference between the non-matching interface and the matching interface is that the address translation semantics for the non-matching interface have no loops. This allows fully pipelined operation for the non-matching address translation.

> **Discussion**: *List entries may be persistent or automatically unlink after first use. Implementations may be able to provide much higher message rates if the priority list contains a persistent list entry at the head of the list. One-sided programming interfaces such as SHMEM and MPI-2 one-sided should be able to take advantage of this performance gain.*

2.4.2 Match Lists and Match List Entries

In addition to the standard address components (process identifier, memory buffer identifier, and offset), a portals address can include information identifying the *initiator* (source) of the message and a set of match bits. This addressing model is appropriate for supporting 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.

For a matching logical network interface, each match list entry specifies two bit patterns: a set of "do not care" bits (ignore bits) and a set of "must match" bits (match bits). Along with the source node ID (NID) and the source process ID (PID), these bits are used in a matching function to select the correct match list entry. In addition, if truncation is disabled (`PTL_ME_NO_TRUNCATE` is set), the message must fit in the buffer. If the message does not fit, the message does not match that entry and matching continues with the next entry.

In addition to *initiator*-specified offsets, match list entries also support locally managed offsets, which allow efficient packing of multiple messages into a single match list entry. When locally managed offsets are enabled, the initiator-specified offset is ignored. A match list entry may additionally specify a minimum available space threshold (*min_free*), after which a persistent match list entry is automatically unlinked. The combination of locally managed offsets, minimum free thresholds, and overflow list semantics allow for the efficient implementation of MPI unexpected messages.

Figure 2.11 illustrates the steps involved in translating a portals address when matching is enabled, starting from the first element in a priority list. If the match criteria specified in the match list entry are met, the permissions check passes, and the match list entry accepts the operation, the operation (*put*, *get*, or *atomic*) is performed using the memory region specified in the match list entry. Note that matching is done using the match bits, ignore bits, and either the node identifier and process identifier or the .

If the match list entry specifies that it is to be unlinked based on the *min_free* semantic or if it is a use once match list entry, the match list entry is removed from the match list, and the resources associated with the match list entry are reclaimed. If there is an event queue specified in the portal table entry and the match list entry accepts the full event, the operation is logged in the event queue. An event is delivered when no more actions, as part of the current operation, will be performed on this match list entry.

If the match criteria specified in the match list entry are not met, the address translation continues with the next match list entry. If the end of the priority list has been reached, address translation continues with the overflow list. Once a matching match list entry has been identified, if the permissions check fails or the match list entry rejects the operation, the matching ceases and the message is dropped without modifying the list state.

2.5 Modifying Data Buffers

Users pass data buffers into the Portals implementation as either a source of data or the destination of data. For buffers where data is being delivered (e.g. at the *target*, or in a reply buffer at the *initiator*), the Portals API allows user memory to be used as a scratch space as long as the operation is larger that *max_atomic_size*. That means an implementation can utilize user memory as scratch space and staging buffers for operations larger than this threshold. When the operation is larger than *max_atomic_size*, the user memory is not guaranteed to reflect exactly the data that has arrived until the operation succeeds and the event is delivered. In fact, for operations larger than *max_atomic_size*, 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 used to receive data in an essentially unpredictable fashion.

The Portals API explicitly prohibits modifying the buffer passed into a *put*. Similarly, an implementation must not alter data in a user buffer that is used in a *reply* operation. This is independent of whether the operation succeeds or fails.

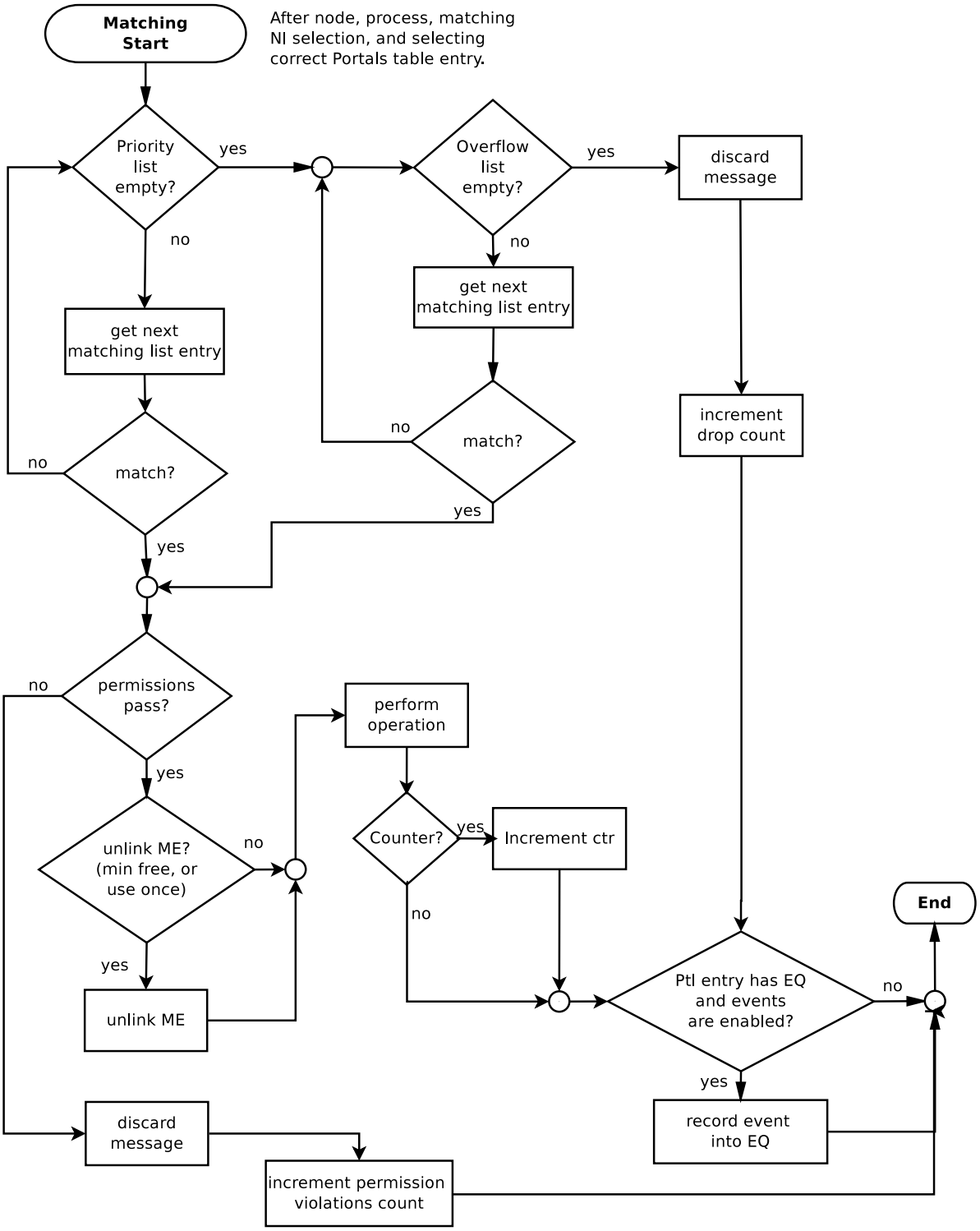


Figure 2.11. Matching Portals Address Translation.

2.6 Ordering

There are two types of ordering typically defined by higher-level languages and message passing APIs: message ordering and data ordering. The message ordering definition controls the order in which messages are processed by the match engine between a pair of endpoints. The data ordering definition controls the order of data delivery into memory. Message and data ordering are complex subjects with a variety of high-level definitions in programming languages and message passing APIs. As such, Portals has a variety of options to control message and data ordering. As a general overview, Portals guarantees byte-granularity data ordering for short messages between a pair of endpoints when targeting a specific list entry or match list entry. For all messages regardless of size, message ordering is provided unless it is disabled using the `PTL_MD_UNORDERED` option in the **ptl_md_t**. This supports the MPI two-sided message ordering requirements while providing the flexibility to disable ordering when it is not needed.

2.6.1 Short Message Ordering Semantics

The default ordering semantics for Portals messages differ for short and long messages. The threshold between "short" and "long" is defined by two parameters, the maximum write-after-write size (*max_waw_ordered_size*) and the maximum write-after-read size (*max_war_ordered_size*). Both parameters are controlled by the *desired* and *actual* arguments of **PtlNIInit()**. Note that replies and acknowledgments do not require ordering.

When one message that stores data (*put*, *atomic*) is followed by a message that stores data or retrieves data (*put*, *atomic*, *get*) from the same initiator to the same target and both messages are less than the *max_waw_ordered_size* in length, a byte from the second message that targets the same offset within the *same* LE (or ME) as a byte from the first message will perform its access after the byte from the first message. Similarly, when one message that retrieves data (*get*) is followed by a second message that stores data (*put*, *atomic*) from the same initiator to the same target and both messages are less than *max_war_ordered_size* in length, a byte from the second message that targets the same offset within the same LE (or ME) as a byte from the first message will perform its access after the byte from the first message.

The order in which individual bytes of a single message are delivered is always unspecified. The order in which non-overlapping bytes of two different messages is *not* specified unless the implementation sets the `PTL_TOTAL_DATA_ORDERING` option in the *actual features* limits field. When total data ordering is provided and the short message constraints are met, the first message must be entirely delivered before any part of the second message is delivered. Support for the ordering of bytes between messages is an optional feature, since some implementations may be unable to provide such strict ordering semantics.

2.6.2 Long Message Ordering Semantics

The default ordering semantics for Portals messages that have a length that is longer than the *max_waw_ordered_size* (or *max_war_ordered_size*, as appropriate) are much weaker. For long messages, the ordering semantics only require that messages sent between a pair of processes are matched at the target in the order they were sent. The underlying implementation is free to deliver the *body* of two messages in whatever order is necessary. This provides additional flexibility to the underlying implementation. For example, the implementation can use a retransmission protocol that only retransmits a portion of a lost message without violating ordering. Similarly, an implementation is free to use adaptive routing to deliver the body of the message. Note that replies and acknowledgments do not require ordering.

> **Discussion**: *The specified ordering semantics of Portals are not necessarily sufficient to allow a* `shmem_fence()` *operation to be treated as a no-op. Portals only guarantees ordering semantics sufficient for* `shmem_fence()` *to be a no-op when* `PTL_TOTAL_DATA_ORDERING` *is returned in the options field of the actual limits* and *the operations are both shorter than max_waw_ordered_size.*

2.6.3 Relative Ordering of Operations in Overlapping Portals

The result of two simultaneous operations targeting the same memory address through different list entries is undefined. The result of a *put* or *atomic* operation transferring data from a memory location (within a memory descriptor) which is currently the target of a remote operation (within a list entry) is also undefined. Data is only available for transmit after the event corresponding to the arriving message has been delivered. Triggered operations are safe, since they do not trigger until the counting event is delivered.

2.6.4 Ordering of Unexpected Messages

Unexpected messages pose a particular challenge for ordering semantics. The unexpected list maintains insertion ordering, although entries in the middle of the list may be removed first based on matching criteria. Data delivery into the overflow list entry which generated the entry in the unexpected list is ordered according to the previously defined rules within that list entry, but have no ordering relative to other list entries.

2.6.5 Relaxing Message Ordering

In many modern networks, adaptive routing can be used to improve the overall network throughput. For these networks, it may be useful for the application to express to the implementation when it is possible to relax the ordering on messages. Portals provides two mechanisms to relax ordering. First, when the application calls **PtlNIInit()**, it can specify a *max_waw_ordered_size* and *max_war_ordered_size* of zero in **ptl_ni_limits_t** (see Sections 3.6.1 and 3.6.2). This informs the application that data ordering is not needed (e.g. in the two sided semantics for MPI). Second, the application can set the `PTL_MD_UNORDERED` option on the **ptl_md_t** used to send the data (see Section 3.10). This turns off both message and data ordering.

2.7 Flow Control

Historically, on some large machines, MPI over Portals has run into problems where the number of unexpected messages has caused the exhaustion of event queue space or buffer space set aside for unexpected messages. MPI implementations over past versions of Portals have handled the overflow by aborting the application. Other networks, such as InfiniBand, use "receiver not ready" NACKs and retransmits at the hardware level. Unfortunately, this is known to prohibit parallelism in the NIC and is detrimental to InfiniBand performance in some areas. In attempting to address this challenge, Portals 4.0 adopts the philosophy that resource exhaustion is an exceptional operating mode and recovery may be slow, but must be possible.

When resources are exhausted, whether they are user allocated resources like EQ entries or implementation level resources, the implementation may choose to block new message processing for a constrained amount of time. If the resources remain exhausted, the behavior of Portals depends on the type of operation which caused the exhaustion and, potentially, options set by the user.

A local operation which generates events (such as a call to **PtlLEAppend()** or **PtlMEAppend()**) or a response from a target-side operation (such as an acknowledgement or reply) is not required to trigger flow control and may cause the event queue to overflow, resulting in dropped events. An implementation may chose to trigger flow control for local operations, but is not required to do so.

> **Discussion**: *The user must use some care when posting a new list entry to ensure that local events do not overflow the event queue. A sufficiently large event queue, drained before posting the list entry, will provide sufficient protection. Implementations may choose to perform more resource exhaustion checking to prevent overflowing the event queue, but are not required to do so.*

A target-side operation (such as the processing of an incoming *put* or *get* operation) which targets a portal table entry on which the PTL_PT_FLOWCTRL option has not been set will not trigger flow control. If the message failed to match in the priority or overflow lists or the message matched in the overflow list and unexpected headers list is full, the message will be dropped, with the PTL_SR_DROP_COUNT status register incremented as specified in Section 2.4. An acknowledgement or reply event will not be generated in this case. If the constrained resource was an event queue, the message will be delivered and any acknowledgement or reply will be generated, but the target-side event will be lost. If there is no space remaining in the unexpected headers list, the incoming message will not match any list entries in the overflow list, which will cause the message to be dropped as described previously.

A target-side operation (such as the processing of an incoming *put* or *get* operation) which targets a portal table entry on which the PTL_PT_FLOWCTRL option has been set will trigger flow control. When flow control is triggered, the implementation must disable the portal table entry and deliver a PTL_EVENT_PT_DISABLED full event to the application (See Implementation Note 12). At this point, all messages targeting that portal table entry for that process must be dropped until **PtlPTEnable()** is called, including the message that caused the flow control event. Messages that are dropped due to a flow control event do not modify any portion of the buffer described by the target list entry or match list entry. In addition, the PTL_EVENT_ACK or PTL_EVENT_REPLY event associated with that message (and subsequent in flight messages) indicate failure. The *ni_fail_type* of any generated full event must be PTL_NI_PT_DISABLED.

> **Discussion**: *It is important to note that remote flow control failure notification is only delivered to the initiator of an operation in the PTL_EVENT_ACK or PTL_EVENT_REPLY event; thus, it is necessary for a user to request acknowledgments at the initiator to be notified of a flow control situation.*

While any internal, potentially implementation specific, resource exhaustion can cause a flow control event, three Portals level resource exhaustion types must cause a flow control event when they occur. If flow control is enabled, the following three scenarios must invoke flow control. First, if an event queue attached to a portal table entry is full and the message would generate a full event, flow control must be invoked. Second, if a message arrives at a portal table entry and does not find a match in either the priority list or the overflow list, flow control must be invoked. Finally, if the space available to buffer unexpected message headers is exhausted (e.g. as indicated by *max_unexpected_headers*), flow control must be invoked.

> **Discussion**: *The application must be involved in flow control recovery. The difficulty in recovering is largely driven by the ordering constraints of the application. Interfaces with loose ordering semantics (such as GASNet) may be able to reduce resource utilization and re-enable a portal table entry without any global communication. Strictly ordered interfaces, such as MPI, must quiesce the library, ensure that resources are available, reach a global consensus that the network is quiesced (likely using another portal table entry for communication), re-enable the portal table entry, and restart communication. Quiescing the library requires the MPI library to insure that no more messages are in flight targeting the node that has experienced resource exhaustion. Making resources available involves draining all full events from the event queue associated with the portal table entry, replenishing the user allocated buffers on the overflow list, and draining unexpected messages from the Portals implementation.*

2.8 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 waiting (**PtlEQWait()**, **PtlCTWait()**), polling (**PtlEQPoll()**, **PtlCTPoll()**), portal table manipulation functions (**PtlPTDisable()**, **PtlPTEnable()**), and some allocation routines (such as **PtlCTAlloc()**,

PtlCTFree(), **PtlEQAlloc()**, **PtlEQFree()**, **PtlMEUnlink()**), every function in the portals API is non-blocking. Every function in the Portals API is 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. In other words, calls into the Portals API are thread safe. 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	3.6
PT	pt	Portal Table Entry	3.7
MD	md	Memory Descriptor	3.10
LE	le	List Entry	3.11
ME	me	Matching list Entry	3.12
EQ	eq	Event Queue	3.13
CT	ct	Count	3.14

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 `portals4.h` that can be included by programs using Portals. Implementations should also provide a README file that explains implementation specific details. For example, it should list the limits (Section 3.6.1) for this implementation and provide a list of status registers that are provided (Section 3.3.7). See Appendix B for a template.

Numerous data structures are described as C-style structures in the Portals API; however, the definition is not meant to specify a field ordering. The implementation is free to optimize the ordering of data structures.

3.2 Constants

The Portals API defines a number of constants. Constants defined in this specification must be compile time constants. Further, constants whose type is specified to be integral must be valid labels for switch statements.

Constants are generally associated with a base type in which constants are stored. Implementations are given freedom regarding the numeric values used for constants and their associated base types, constrained only by the compile time requirements.

3.3 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 3.5 on page 110 lists all the types defined by Portals.

3.3.1 Sizes

The type **ptl_size_t** is an unsigned 64-bit integral type used for representing sizes. The constant `PTL_SIZE_MAX` represents the largest value a **ptl_size_t** can hold.

3.3.2 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.

The type of a handle is left unspecified, but must be assignable in C. Every Portals object is associated with a specific network interface and the network handle associated with an object's handle may be retrieved by calling **PtlNIHandle()**.

IMPLEMENTATION NOTE 4: Size of handle types

It is highly recommended that a handle type should be no larger than the native machine word size.

The constant `PTL_EQ_NONE`, of type **ptl_handle_eq_t**, is used to indicate the absence of an event queue. Similarly, the constant `PTL_CT_NONE`, of type **ptl_handle_ct_t**, indicates the absence of a counting event. See Section 3.10.1 for uses of these values. The special constant `PTL_INVALID_HANDLE` is used to represent an invalid handle.

IMPLEMENTATION NOTE 5: Unique handles

The encoding of handles is not specified by the Portals API. An implementation may reuse handle values, however the implementation is responsible for handling race conditions between threads calling release and acquire functions (such as **PtlMDRelease()** and **PtlMDBind()**).

3.3.3 Indexes

The type **ptl_pt_index_t** is an integral type used for representing portal table indexes. See Section 3.6.1 and 3.6.2 for limits on values of this type.

3.3.4 Match Bits

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

3.3.5 Network Interfaces

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

3.3.6 Identifiers

The type **ptl_nid_t** is an integral type used for representing node identifiers and **ptl_pid_t** is an integral type for representing process identifiers when physical addressing is used in the network interface (`PTL_NI_PHYSICAL` is set for the network interface). If `PTL_NI_LOGICAL` is set, a *rank* (**ptl_rank_t**) is used instead. **ptl_uid_t** is an integral type for representing user identifiers.

The special values `PTL_PID_ANY` matches any process identifier, `PTL_NID_ANY` matches any node identifier, `PTL_RANK_ANY` matches any rank, and `PTL_UID_ANY` matches any user identifier. See Section 3.11 and 3.12 for uses of these values.

3.3.7 Status Registers

Each network interface maintains an array of status registers that can be accessed using the **PtlNIStatus()** function (Section 3.6.4). The type **ptl_sr_index_t** defines the type of indexes that can be used to access the status registers. A small number of indexes are defined for all implementations:

Status Register Indexes (ptl_sr_index_t)

`PTL_SR_DROP_COUNT`	Identifies the status register that counts the dropped requests for the interface.
`PTL_SR_PERMISSION_VIOLATIONS`	Counts the number of attempted permission violations.
`PTL_SR_OPERATION_VIOLATIONS`	Counts the number of attempted operation violations

A permission violation is a violation of the user id check, while an operation violation is a violation of the allowed operation types (put and/or get). Note that these three operations are orthogonal such that permission violations and operations violations should not increment `PTL_SR_DROP_COUNT`. Other indexes (and registers) may be defined by the implementation.

The type **ptl_sr_value_t** defines the type of values held in status registers. This is a signed integer type. The size is

implementation dependent but must be at least 32 bits.

3.4 Function Arguments and Return Codes

Unless otherwise noted, an implementation is not required to check the validity of any arguments to a Portals function call. The argument to many Portals functions is a pointer to a type (because the argument is a pointer to a structure and/or because the argument is an output parameter). Unless otherwise noted, a pointer must point to a valid instance of the specified type; NULL is not generally a valid argument.

The Portals 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_ARG_INVALID**. 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 3.7 on page 113 lists all return codes used by Portals.

3.5 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. The initialization state of Portals is reference counted so that repeated calls to **PtlInit()** and **PtlFini()** within a process (collection of threads) do not invalidate Portals state until the reference count reaches zero. Portals is *initialized* upon successful completion of the first call to **PtlInit()** and *finalized* upon successful completion of the first call to **PtlFini()** that results in the reference count reaching zero.

A child process does not inherit any Portals resources from its parent. A child process must initialize Portals in order to obtain new, valid Portals resources. If a child process fails to initialize Portals and then uses the Portals interface, behavior is undefined for both the parent and the child.

3.5.1 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 and may be safely called more than once. Each call to **PtlInit()** increments a reference count. **PtlInit()** cannot be called after the Portals library has been finalized.

Function Prototype for PtlInit

```
int PtlInit(void);
```

Return Codes

PTL_OK	Indicates success.
PTL_FAIL	Indicates an error during initialization.

3.5.2 PtlFini

The **PtlFini()** function allows an application to clean up after the Portals library is no longer needed by a process. Each call to **PtlFini()** decrements the reference count that was incremented by **PtlInit()**. When the reference count reaches zero, all Portals resources are freed. Once the Portals resources are freed, 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. Each call to **PtlInit()** should be matched by a corresponding **PtlFini()**.

Function Prototype for PtlFini

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

3.6 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 handled internally by the Portals implementation, embedded in the underlying network, or handled by the application. Interfaces are treated as independent entities to make it easier to cache information on individual network interface cards.

A Portals *physical* network interface is a per-process abstraction of a physical network interface (or group of interfaces). A physical network interface can not be used directly, but can be used by a process to instantiate up to four *logical* network interfaces. All logical network interfaces associated with a single physical network interface share the same network id and process id (nid/pid), but all other resources are unique to a logical network interface. A logical network interface can be initialized to provide either matching or non-matching Portals addressing and either logical or physical addressing of network endpoints through the data movement calls. These two options are independent and all four logical network interface options must be supported by each physical network interface.

Once initialized, each logical interface provides a portal table and a collection of status registers. In order to facilitate the development of portable Portals applications, a compliant implementation must provide at least 64 portal table entries. See Section 3.6.4 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 list entry) is also associated with a specific logical network interface. The association to a logical network interface is established when the object is created, and the **PtlNIHandle()** function (Section 3.6.5) may be used to determine the logical network interface with which an object is associated.

Each logical network interface is initialized and shut down independently. The initialization routine, **PtlNIInit()**, returns an interface object handle which is used in all subsequent portals operations. The **PtlNIFini()** function is used to shut down a logical 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.

3.6.1 The Network Interface Limits Type

The function **PtlNIInit()** accepts a pointer to a structure of desired limits and can fill a structure with the actual values supported by the network interface. Resource limits are specified independently for each logical network interface. The two structures are of type **ptl_ni_limits_t** and include the following members:

```
typedef struct {
    int max_entries;
    int max_unexpected_headers;
    int max_mds;
    int max_cts;
    int max_eqs;
    int max_pt_index;
    int max_iovecs;
    int max_list_size;
    int max_triggered_ops;
    ptl_size_t max_msg_size;
    ptl_size_t max_atomic_size;
    ptl_size_t max_fetch_atomic_size;
    ptl_size_t max_waw_ordered_size;
    ptl_size_t max_war_ordered_size;
    ptl_size_t max_volatile_size;
    unsigned int features;
} ptl_ni_limits_t;
```

Limits

max_entries	Maximum number of match list entries or list entries that can be allocated at any one time (only one of the two exists on an interface).
max_unexpected_headers	Maximum number of unexpected headers that the implementation can buffer.
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_cts	Maximum number of counting events that can be allocated at any one time.
max_pt_index	Largest portal table index for this interface, valid indexes range from 0 to *max_pt_index*, inclusive. An interface must support a *max_pt_index* of at least 63.
max_iovecs	Maximum number of I/O vectors for a single memory descriptor, list entry, or match list entry for this interface.
max_list_size	Maximum number of entries that can be attached to the list on any portal table index.
max_triggered_ops	Maximum number of triggered operations that can be outstanding.
max_msg_size	Maximum size (in bytes) of a message (*put*, *get*, or *reply*).
max_atomic_size	Maximum size (in bytes) that can be passed to an atomic operation. Any byte within an operation that is less than *max_atomic_size* is guaranteed to only be written to the user memory buffer once.
max_fetch_atomic_size	Maximum size (in bytes) that can be passed to an atomic operation that returns the prior value to the initiator.
max_waw_ordered_size	Maximum size (in bytes) of a message that will guarantee “per-address” data ordering for a write followed by a write (consecutive *put* or *atomic* or a mixture of the two) and a write followed by a read (*put* followed by a *get*) An interface must provide a *max_waw_ordered_size* of at least 64 bytes.

max_war_ordered_size	Maximum size (in bytes) of a message that will guarantee "per-address" data ordering for a read followed by a write (*get* followed by a *put* or *atomic*). An interface must provide a *max_war_ordered_size* of at least 8 bytes.
max_volatile_size	Maximum size (in bytes) that can be passed as the *length* of a *put* or *atomic* for a memory descriptor with the `PTL_MD_VOLATILE` option set.
features	A bit mask of features supported by the the Portals implementation. Currently, three features are defined. `PTL_TARGET_BIND_INACCESSIBLE` is discussed in Section 3.11 and 3.12, `PTL_TOTAL_DATA_ORDERING` is discussed in Section 2.6, and `PTL_COHERENT_ATOMICS` is discussed in Section 3.15.4.

3.6.2 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. An additional call to **PtlSetMap()** must be made before communication calls are made on a logically addressed interface (See Section 3.6.6). Calls to **PtlNIInit()** increment a reference count on the network interface and must be matched by a call to **PtlNIFini()**. If **PtlNIInit()** gets called more than once *per logical interface*, then the implementation should fill in *actual* and *ni_handle* with the values obtained by the first caller and should ignore the *pid* argument. **PtlGetId()** or **PtlGetPhysId()** (Section 3.9) can be used to retrieve the *pid*.

> **Discussion**: *Proper initialization of a logical network interface that uses logical endpoint addressing requires the user to call **PtlSetMap()**, creating a mapping of logical* ranks *to physical node IDs and process IDs. The physical address (NID/PID) associated with a logical network interface may be obtained by calling **PtlGetPhysId()**. The physical address may then be shared through an outside mechanism (including another Portals logical interface) to establish a consistent mapping of rank to NID/PID.*

Function Prototype for PtlNIInit

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

Arguments

iface	**input**	Identifies the physical network interface to be initialized. (See Section 3.3.5 for a discussion of values used to identify network interfaces.)
options	**input**	This field contains options that are requested for the network interface. Values for this argument can be constructed using a bitwise OR of the values defined below. Either `PTL_NI_MATCHING` or `PTL_NI_NO_MATCHING` must be set, but not both. Either `PTL_NI_LOGICAL` or `PTL_NI_PHYSICAL` must be set, but not both, to specify the endpoint addressing mode.

pid	**input**	Identifies the desired process identifier (for well known process identifiers). The specified *pid* must either be non-negative and less than the value `PTL_PID_MAX` or be `PTL_PID_ANY`. The value `PTL_PID_ANY` may be used to let the Portals library select a process identifier. See Section 3.9 for more information on process identifiers.
desired	**input**	If not NULL, points to a structure that holds the desired limits. If NULL, either previously set limits or implementation defined defaults will be used.
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 the interface handle.

options

`PTL_NI_MATCHING`	Request that the interface specified in *iface* be opened with matching enabled.
`PTL_NI_NO_MATCHING`	Request that the interface specified in *iface* be opened with matching disabled. `PTL_NI_MATCHING` and `PTL_NI_NO_MATCHING` are mutually exclusive.
`PTL_NI_LOGICAL`	Request that the interface specified in *iface* be opened with logical endpoint addressing (e.g. GASNet node and rank or SHMEM PE).
`PTL_NI_PHYSICAL`	Request that the interface specified in *iface* be opened with physical endpoint addressing (e.g. NID/PID). `PTL_NI_LOGICAL` and `PTL_NI_PHYSICAL` are mutually exclusive.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_PID_IN_USE	Indicates that *pid* is currently in use.
PTL_NO_SPACE	Indicates that **PtlNIInit()** was not able to allocate the memory required to initialize the interface.

> **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. However, the use of desired is implementation dependent and an implementation may choose to ignore the request or provide limits based on a previous request.*

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. An implementation must provide at least the resources specified by *actual*, unless bounded by another resource such as available application memory or machine capabilities.

3.6.3 PtlNIFini

The **PtlNIFini()** function is used to release the resources allocated for a network interface. The release of network interface resources is based on a reference count that is incremented by **PtlNIInit()** and decremented by **PtlNIFini()**. Resources can only be released when the reference count reaches zero. Once the release of resources has begun, 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*, *atomic*) or return values (*acknowledgment* and *reply*) for this interface are undefined until the interface is reinitialized by another call to **PtlNIInit()**.

Function Prototype for PtlNIFini

```
int PtlNIFini(ptl_handle_ni_t ni_handle);
```

Arguments

ni_handle	**input**	An interface handle to shut down.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.6.4 PtlNIStatus

The **PtlNIStatus()** function returns the value of a status register for the specified interface. See Section 3.3.7 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**	An interface handle.
status_register	**input**	The index of the status register.
status	**output**	On successful return, this location will hold the current value of the status register.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.6.5 PtlNIHandle

The **PtlNIHandle()** function returns the network interface handle 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**	The object handle.
ni_handle	**output**	On successful return, this location will hold the network interface handle associated with *handle*.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.6.6 PtlSetMap

The **PtlSetMap()** function initializes the mapping from *logical* endpoint identifiers (rank) to *physical* endpoint identifiers (nid/pid) for the given logically addressed logical network interface. A process must ensure that the logical mapping is set before the specified logically addressed logical network interface may be used in any portals calls other than **PtlNIInit()**, **PtlGetMap()**, and **PtlGetPhysId()**. If the map of the other logically addressed logical network interface associated with the same physical network interface as the specified interface handle has not been set by a call to **PtlSetMap()**, the implementation may choose to set the mapping on both logical network interfaces. It is erroneous to call **PtlSetMap()** on a physically addressed logical network interface. Subsequent calls (either by different threads or the same thread) to **PtlSetMap()** will overwrite any mapping associated with the logical network interface; hence, libraries must take care to ensure reasonable interoperability.

Function Prototype for PtlSetMap

```
int PtlSetMap(ptl_handle_ni_t ni_handle,
              ptl_size_t map_size,
              const ptl_process_t *mapping);
```

Arguments

ni_handle	**input**	The interface handle identifying the network interface which should be initialized with *mapping*. The network interface handle must refer to a logically addressed network interface.
map_size	**input**	The number of elements in *mapping*.
mapping	**input**	Points to an array of **ptl_process_t** structures where entry N in the array contains the NID/PID pair that is associated with the logical rank N.

Return Codes

PTL_OK	Indicates success.
PTL_IGNORED	Indicates success, but that the implementation does not support dynamic changing of the logical identifier map, likely due to integration with a static run-time system.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_SPACE	Indicates that **PtlSetMap()** was not able to allocate the memory required to initialize the map.

Discussion: ***PtlSetMap()*** *is a local operation and the map set by different communicating processes may be different. The rank field of target-side events may be unexpected in cases where the two processes have different maps.*

3.6.7 PtlGetMap

The **PtlGetMap()** function retrieves the mapping from *logical* identifiers (rank) to *physical* identifiers (nid/pid) for the specified logically addressed logical network interface. If the *map_size* is smaller than the actual map size, the first *map_size* entries in the map will be copied into *mapping*. If the *map_size* is larger than the actual map size, the entire map is copied into *mapping* and the buffer beyond the *actual_map_size* entry is left unmodified. It is erroneous to call **PtlGetMap()** on a physically addressed logical network interface.

Function Prototype for PtlGetMap

```
int PtlGetMap(ptl_handle_ni_t ni_handle,
              ptl_size_t map_size,
              ptl_process_t *mapping,
              ptl_size_t *actual_map_size);
```

Arguments

ni_handle	**input**	The network interface handle from which the map should be retrieved. The network interface handle must refer to a logically addressed logical network interface.
map_size	**input**	The length of *mapping* in number of elements.
mapping	**output**	Points to an array of **ptl_process_t** structures where entry N in the array will be populated with the NID/PID pair that is associated with the logical rank N.
actual_map_size	**output**	On return, *actual_map_size* contains the size, in number of elements, of the map currently associated with the logical interface. May be bigger than *map_size* or the *mapping* array.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_SPACE	Indicates that there was no map set on the logical network interface.

3.7 Portal Table Entries

A portal index refers to a portal table entry. The assignment of these indexes can either be statically or dynamically managed, and will typically be a combination of both. A portal table entry must be allocated before being used. From a user perspective, messages that arrive traverse list entries or match list entries in the order they were appended within a single portal table index. Resource exhaustion (Section 2.7) is handled independently on different portal table entries.

3.7.1 PtlPTAlloc

The **PtlPTAlloc()** function allocates a portal table entry and sets flags that pass options to the implementation.

Function Prototype for PtlPTAlloc

```
int PtlPTAlloc(ptl_handle_ni_t ni_handle,
               unsigned int options,
               ptl_handle_eq_t eq_handle,
               ptl_pt_index_t pt_index_req,
               ptl_pt_index_t *pt_index);
```

Arguments

ni_handle	**input**	The interface handle to use.
options	**input**	This field contains options that are requested for the portal index. Values for this argument can be constructed using a bitwise OR of the values defined below.

eq_handle	**input**	The event queue handle used to log the events related to the list entries attached to the portal table entry. If this argument is `PTL_EQ_NONE`, events related to this portal table entry are not logged.
pt_index_req	**input**	The value of the portal index that is requested. If the value is set to `PTL_PT_ANY`, the implementation can return any portal index.
pt_index	**output**	On successful return, this location will hold the portal index that has been allocated.

options

`PTL_PT_ONLY_USE_ONCE`	Hint to the underlying implementation that all entries attached to the priority list on this portal table entry will have the `PTL_ME_USE_ONCE` or `PTL_LE_USE_ONCE` option set.
`PTL_PT_ONLY_TRUNCATE`	Hint to the underlying implementation that all entries attached to the priority list on this portal table entry will not have the `PTL_ME_NO_TRUNCATE` option set.
`PTL_PT_FLOWCTRL`	Enable flow control on this portal table entry (see Section 2.7).

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_PT_FULL	Indicates that there are no free entries in the portal table.
PTL_PT_IN_USE	Indicates that the Portal table entry requested is in use.
PTL_PT_EQ_NEEDED	Indicates that flow control is enabled and there is no EQ attached.

Discussion: *The `PTL_PT_ONLY_USE_ONCE` and `PTL_PT_ONLY_TRUNCATE` options are hints to the implementation that convey that the user will be employing certain common usage scenarios when using the priority list. Use of these options may allow the implementation to optimize the matching logic. Note that the optimal set of options may vary depending on whether matching or non-matching logical network interfaces are used. For a matching logical network interface, an implementation likely may optimize the case where both `PTL_PT_ONLY_USE_ONCE` and `PTL_PT_ONLY_TRUNCATE` are specified. For a non-matching logical network interface, pre-posted persistent LEs are likely to provide better performance.*

3.7.2 PtlPTFree

The **PtlPTFree()** function releases the resources associated with a portal table entry. Objects associated with the portal table entry, such as list entries and event queues, are not freed as the result of a call to **PtlPTFree()**.

Function Prototype for PtlPTFree

```
int PtlPTFree( ptl_handle_ni_t ni_handle,
               ptl_pt_index_t pt_index);
```

Arguments

ni_handle	**input**	The interface handle on which the *pt_index* should be freed.
pt_index	**input**	The portal index that is to be freed.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_PT_IN_USE	Indicates that *pt_index* is currently in use (e.g. a match list entry is still attached).

3.7.3 PtlPTDisable

The **PtlPTDisable()** function indicates to an implementation that no new messages should be accepted on the specified portal table entry. The function blocks until the portal table entry status has been updated, all messages being actively processed are completed, and all events are delivered. Since **PtlPTDisable()** waits until the portal table entry is disabled before it returns, it does not generate a `PTL_EVENT_PT_DISABLED` event. Processing of operations targeting other portal table entries and local operations continues after a call to **PtlPTDisable()**.

Function Prototype for PtlPTDisable

```
int PtlPTDisable(ptl_handle_ni_t ni_handle,
                 ptl_pt_index_t pt_index);
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal index that is to be disabled.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

3.7.4 PtlPTEnable

The **PtlPTEnable()** function indicates to an implementation that a previously disabled portal table entry should be re-enabled. This is used to enable portal table entries that were automatically or manually disabled. The function

blocks until the portal table entry is enabled.

Function Prototype for PtlPTEnable

```
int PtlPTEnable(ptl_handle_ni_t ni_handle,
                ptl_pt_index_t pt_index);
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The value of the portal index to enable.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

Discussion: ***PtlPTEnable()*** *re-enables a portal table entry, allowing incoming messages to match against list entries associated with the portal table entry. Messages may have been dropped while the portal table entry was disabled. Higher level communication protocols with strict ordering constraints may have to quiesce messages and retransmit after re-enabling a portal table entry (See Section 2.7).*

3.8 User Identification

Every process runs on behalf of a user. User identifiers are included in the trusted portion of the header of a portals message. They can be used at the *target* to limit access to list entries (Section 3.11 and Section 3.12). The uid is common across logical network interfaces within the same process, even if the logical network interfaces are over different physical network interfaces.

3.8.1 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.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

3.9 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 node identifier and process identifier can be aggregated by the application into a rank, which is translated by the implementation into a network identifier and process identifier. It is an implementation decision whether two physical network interfaces in the same node have the same node or process identifiers. All logical network interfaces which share the same physical network interface share the same node and process identifiers.

The Portals API defines a type, **ptl_process_t**, for representing process identifiers, and two functions, **PtlGetId()** and **PtlGetPhysId()**, 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).*

3.9.1 The Process Identification Type

The **ptl_process_t** type is a union that can represent the process as either a physical address or a logical address within the machine. The physical address uses two identifiers to represent a process identifier: a node identifier *nid* and a process identifier *pid*. In turn, a logical address uses a logical index within a translation table specified by the application (the *rank*) to identify another process.

```
typedef union {
    struct {
        ptl_nid_t nid;
        ptl_pid_t pid;
    } phys;
    ptl_rank_t rank;
} ptl_process_t;
```

3.9.2 PtlGetId

Function Prototype for PtlGetId

```
int PtlGetId(ptl_handle_ni_t ni_handle,
             ptl_process_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. If the interface is logically addressed, the logical address is returned. If the interface is physically addressed, the physical address is returned.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

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

3.9.3 PtlGetPhysId

Function Prototype for PtlGetPhysId

```
int PtlGetPhysId(ptl_handle_ni_t ni_handle,
                 ptl_process_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. The physical address is always returned, even for logically addressed network interfaces.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

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

3.10 Memory Descriptors

A memory descriptor contains information about a region of a process' memory and optionally points to an event queue and counting event where information about the operations performed on the memory descriptor are recorded. Memory descriptors are initiator side resources that are used to encapsulate the association of a network interface (NI) with a description of a memory region. They provide an interface to register memory (for operating systems that require it) and to carry that information across multiple operations (an MD is persistent until released). **PtlMDBind()** is used to create a memory descriptor and **PtlMDRelease()** is used to unlink and release the resources associated with a memory descriptor.

A memory descriptor describes a memory region using a base address and length; however, it is not a requirement for all of the memory described by the memory descriptor to be allocated or accessible within the application. For example, an application can create a memory descriptor that covers the entire virtual address range by setting *start* to NULL and *length* to `PTL_SIZE_MAX`, even though the entire region is not currently allocated. If the application issues a portals operation (e.g. *put*) that would access an unallocated region of the MD, the implementation may either cause a segmentation fault of the application or may simply fail the operation. If a full event is delivered, it must set *ni_fail_type* to `PTL_NI_SEGV`. If the memory descriptor sets the `PTL_IOVEC` option, the memory region(s) described by the **ptl_iovec_t** must all be accessible within the application.

IMPLEMENTATION NOTE 6:	Memory descriptors that bind inaccessible memory The implementation is responsible for handling any issues, such as the memory registration required by some platforms, that arise from the ability of an MD to cover all of the virtual address space. While some implementations may have elegant solutions to this issue (e.g. lightweight kernels or NIC hardware translation caching), other implementations may require registration caching schemes.

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 the memory descriptors.

```
typedef struct {
    void *start;
    ptl_size_t length;
    unsigned int options;
    ptl_handle_eq_t eq_handle;
    ptl_handle_ct_t ct_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. There are no restrictions on buffer alignment, the starting address or the length of the region; although messages that are not natively aligned (e.g. to a four byte or eight byte boundary) may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

options
: Specifies the behavior of the memory descriptor. Options include the use of scatter/gather vectors and control of events associated with this memory descriptor. Values for this argument can be constructed using a bitwise OR of the following values:

`PTL_MD_EVENT_SEND_DISABLE`
: Specifies that this memory descriptor should not generate send events (`PTL_EVENT_SEND`). This flag does not affect counting events.

`PTL_MD_EVENT_SUCCESS_DISABLE`
: Specifies that this memory descriptor should not generate full events if the *ni_fail_type* would be **PTL_OK**. This flag does not affect counting events. Disabling full events for successful operations is useful in scenarios when a counting event is sufficient for completion, but more information is needed for error recovery.

`PTL_MD_EVENT_CT_SEND`
: Enable the counting of `PTL_EVENT_SEND` events.

`PTL_MD_EVENT_CT_REPLY`
: Enable the counting of `PTL_EVENT_REPLY` events.

`PTL_MD_EVENT_CT_ACK`
: Enable the counting of `PTL_EVENT_ACK` events.

`PTL_MD_EVENT_CT_BYTES`
: By default, counting events count events. When set, this option causes bytes to be counted instead for success events. Byte counts must be incremented exactly once per operation. The increment is by the *mlength* that would be specified by the associated full event. Failure events always increment the count by one.

`PTL_MD_UNORDERED`
: Indicate to the Portals implementation that messages sent from this memory descriptor do not have to arrive at the target in order. Note that this has no impact on acknowledgments or replies, which are never required to be ordered.

`PTL_MD_VOLATILE`
: Indicate to the Portals implementation that the application may modify any send buffers associated with this memory descriptor immediately following the return from a portals operation. Operations should not return until it is safe for the application to reuse any send buffers. The Portals implementation is not required to honor this option unless the size of the operation is less than or equal to *max_volatile_size*. Note that the MD can be of any size, but the Portals implementation must honor this option as long as the operation (e.g. *put*) uses a length less than or equal to *max_volatile_size*. If the application sets `PTL_MD_VOLATILE` and violates the *max_volatile_size*, the operation may fail.

`PTL_IOVEC`
: Specifies that the `start` argument is a pointer to an array of type **ptl_iovec_t** (Section 3.10.2) and the `length` argument is the length of the array of **ptl_iovec_t** elements. 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 array of **ptl_iovec_t** elements referred to by the `start` argument cannot be changed or released for the lifetime of the memory descriptor.

eq_handle	The event queue handle 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.
ct_handle	A handle for counting events associated with the memory region. If this argument is `PTL_CT_NONE`, operations performed on this memory descriptor are not counted.

3.10.2 The I/O Vector Type

The **ptl_iovec_t** type is used to describe scatter/gather buffers of a memory descriptor, list entry, or match list entry in conjunction with the `PTL_IOVEC` option. The **ptl_iovec_t** type is intended to be a type definition of the `struct iovec` type on systems that already support this type.

The **ptl_iovec_t** array is passed as the *start* field when creating a memory descriptor, list entry, or match list entry must not be modified or destroyed by the application or implementation for the life of the descriptor or entry. Descriptors or entries using **ptl_iovec_t** types may be mixed with offsets (local and remote). The offset is computed as if the region described by the **ptl_iovec_t** type were a single contiguous region.

> **Discussion**: *Performance conscious users should not mix offsets (local or remote) with* ***ptl_iovec_t*** *. While this is a* supported *operation, it may have unexpected performance consequences.*

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

Members

iov_base	The byte aligned start address of the vector element
iov_len	The length (in bytes) of the vector element

3.10.3 PtlMDBind

The **PtlMDBind()** operation is used to create a memory descriptor to be used by the *initiator*. On systems that require memory registration, the **PtlMDBind()** operation should invoke the appropriate memory registration functions.

Function Prototype for PtlMDBind

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

Arguments

ni_handle	**input**	The network interface handle with which the memory descriptor will be associated.
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 implementation.
md_handle	**output**	On successful return, this location will hold the newly created memory descriptor handle. 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_ARG_INVALID	Indicates that an invalid argument was passed. Argument checking is implementation dependent, but this may indicate that an invalid *ni_handle* was used, an invalid event queue was associated with the *md*, or other contents in the *md* were illegal.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the memory descriptor.

IMPLEMENTATION NOTE 7: Optimization for Duplicate Memory Descriptors

Because the *eq_handle* and *ct_handle* are bound to the memory descriptor on the initiator, there are usage models where it is necessary to create numerous memory descriptors that only differ in their *eq_handle* or *ct_handle* field. Implementations may desire to optimize for this usage model.

3.10.4 PtlMDRelease

The **PtlMDRelease()** function 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.) Only memory descriptors with no pending operations may be unlinked. A memory descriptor is considered to have pending operations if an operation has been started and the corresponding `PTL_EVENT_SEND` or `PTL_EVENT_REPLY` operation has not been delivered. A memory descriptor may be released before a `PTL_EVENT_ACK` event is delivered, in which case the acknowledgment will be discarded.

Function Prototype for PtlMDRelease

```
int PtlMDRelease(ptl_handle_md_t md_handle);
```

Arguments

md_handle	**input**	The memory descriptor handle to be released.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.11 List Entries and Lists

A list is a chain of list entries. Examples of lists include the priority list and the overflow list. Each list entry (LE) describes a memory region and includes a set of options. It is the target side analogue of the memory descriptor (MD) for non-matching logical network interfaces. The **PtlLEAppend()** function appends a single list entry to the specified list on the specified portal index and returns the list entry handle. List entries can be dynamically removed from a list using the **PtlLEUnlink()** function.

Like a memory descriptor, a list entry describes a memory region using a base address and length. A zero-length list entry may be created by setting *start* to NULL and *length* to 0. Zero-length buffers (NULL LE) are useful to record events. Messages that are outside the bounds of the LE are truncated to zero bytes (e.g. zero-length buffers or an offset beyond the length of the LE). If the interface set the `PTL_TARGET_BIND_INACCESSIBLE` bit in the *features* field of the *actual* limits (See Section 3.6.1), then it is not a requirement for all of the memory described by the list entry to be allocated or accessible within the application. For example, an application could create a list entry that covers the entire virtual address range by setting *start* to NULL and *length* to `PTL_SIZE_MAX`, even though the entire region is not currently allocated. If an incoming operation (e.g. *put*) attempts to access an unallocated region of the LE, the implementation may either cause a segmentation fault of the application or may simply fail the operation. If a full event is delivered, it must set *ni_fail_type* to `PTL_NI_SEGV`. The target may, however, set the `PTL_LE_IS_ACCESSIBLE` option to indicate that the entire memory space described by the LE is accessible. If the list entry sets the `PTL_IOVEC` option, the memory region(s) described by the **ptl_iovec_t** must all be accessible within the application.

IMPLEMENTATION NOTE 8:

List entries that bind inaccessible memory

If the implementation returns `PTL_TARGET_BIND_INACCESSIBLE`, then the implementation is responsible for handling any issues, such as the memory registration required by some platforms, that arise from the ability of an LE to cover all of the virtual address space. While some implementations may have elegant solutions to this issue (e.g. lightweight kernels or NIC hardware translation caching), other implementations may require a software thread on the target to implement a remote registration caching scheme like Firehose [3].

List entries can be appended to either the priority list or the overflow list associated with a portal table entry; however, when attached to an overflow list, additional semantics are implied that require the implementation to track messages that arrive in list entries. Essentially, the memory region identified is provided to the implementation for use in managing unexpected messages. Buffers provided in the overflow list will post a full event (`PTL_EVENT_AUTO_UNLINK`) when the buffer space has been consumed, to notify the application that more buffer space may be needed. When the application is free to reuse the buffer (i.e. the implementation is done with it), another full event (`PTL_EVENT_AUTO_FREE`) will be posted. The `PTL_EVENT_AUTO_FREE` full event will be posted after *all* other events associated with the buffer have been delivered.

Discussion: *It is the responsibility of the application to ensure that the implementation has sufficient*

buffer space to manage unexpected messages (i.e. in the unexpected list). Failure to do so will cause messages to be dropped. The `PTL_EVENT_ACK` *at the initiator will indicate the failure as described in Section 3.13.3. Note that overflow events can readily exhaust the event queue. Proper use of the API will generally require the application to post at least two (and typically several) buffers so that the application has time to notice the* `PTL_EVENT_AUTO_UNLINK` *and replace the buffer. In many usage scenarios, however, the application may choose to have only persistent list entries—list entries without the* `PTL_LE_USE_ONCE` *option set—in the priority list. Thus, overflow list entries will not be required.*

It is the responsibility of the implementation to determine when a buffer that is automatically unlinked from an overflow list can be reused. It must note that it is no longer holding state associated with the buffer and post a `PTL_EVENT_AUTO_FREE` *full event after all other events associated with that buffer have been delivered.*

List entries can be appended to a network interface with the `PTL_NI_NO_MATCHING` option set (a non-matching network interface). A matching network interface requires a match list entry.

3.11.1 The List Entry Type

The **ptl_le_t** type defines the visible parts of a list entry. Values of this type are used to initialize the list entries.

```
typedef struct {
    void *start;
    ptl_size_t length;
    ptl_handle_ct_t ct_handle;
    ptl_uid_t uid;
    unsigned int options;
} ptl_le_t;
```

Members

start, *length*	Specify the memory region associated with the list entry. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. There are no restrictions on buffer alignment, the starting address or the length of the region; although messages that are not natively aligned (e.g. to a four byte or eight byte boundary) may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.
ct_handle	A handle for counting events associated with the memory region. If this argument is `PTL_CT_NONE`, operations performed on this list entry are not counted.
uid	Specifies the user ID that may access this list entry. The user ID may be set to a wildcard (`PTL_UID_ANY`). If the access control check fails, then the message is dropped without modifying Portals state. This is treated as a permissions failure and the status register indexed by `PTL_SR_PERMISSION_VIOLATIONS` is incremented. This failure is also indicated to the initiator. If a full event is delivered to the initiator, the *ni_fail_type* in the `PTL_EVENT_ACK` event must be set to `PTL_NI_PERM_VIOLATION`.

options	Specifies the behavior of the list entry. 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 and control event delivery. Values for this argument can be constructed using a bitwise OR of the following values:
PTL_LE_OP_PUT	Specifies that the list entry will respond to *put* operations. By default, list entries reject *put* operations. If a *put* operation targets a list entry where PTL_LE_OP_PUT is not set, it is treated as an operations failure and PTL_SR_OPERATION_VIOLATIONS is incremented. If a full event is delivered to the initiator, the *ni_fail_type* in the PTL_EVENT_ACK event must be set to PTL_NI_OP_VIOLATION.
PTL_LE_OP_GET	Specifies that the list entry will respond to *get* operations. By default, list entries reject *get* operations. If a *get* operation targets a list entry where PTL_LE_OP_GET is not set, it is treated as an operations failure and PTL_SR_OPERATION_VIOLATIONS is incremented. If a full event is delivered to the initiator, the *ni_fail_type* in the PTL_EVENT_ACK event must be set to PTL_NI_OP_VIOLATION. **Note**: It is not considered an error to have a list entry that does not respond to either *put* or *get* operations: Nor is it considered an error to have a list entry that responds to both *put* and *get* operations. In fact, a list entry must be configured to respond to both put and get operations to properly handle a **PtlFetchAtomic()** or **PtlSwap()** operation.
PTL_LE_USE_ONCE	Specifies that the list entry will only be used once and then unlinked. If this option is not set, the list entry persists until it is explicitly unlinked.
PTL_LE_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. See Section 3.13.3 for exceptions to this rule. This applies to both full and counting events. Acknowledgments are never sent for *get* operations. The data sent in the *reply* serves as an implicit acknowledgment.
PTL_LE_UNEXPECTED_HDR_DISABLE	Specifies that the header for a message delivered to this list entry should not be added to the unexpected list. This option only has meaning if the list entry is inserted into the overflow list. By creating a list entry which truncates messages to zero bytes, disables comm events, and sets this option, a user may create a list entry which consumes no target side resources.
PTL_IOVEC	Specifies that the start argument is a pointer to an array of type **ptl_iovec_t** (Section 3.10.2) and the length argument is the length of the array. This allows for a scatter/gather capability for list entries. A scatter/gather list entry behaves exactly as a list entry that describes a single virtually contiguous region of memory. All other semantics are identical. The array of **ptl_iovec_t** elements referred to by the start argument cannot be changed or released until the list entry is unlinked.
PTL_LE_IS_ACCESSIBLE	Indicate that this list entry only contains memory addresses that are accessible by the application.
PTL_LE_EVENT_LINK_DISABLE	Specifies that this list entry should not generate a PTL_EVENT_LINK full event indicating the list entry successfully linked.
PTL_LE_EVENT_COMM_DISABLE	Specifies that this list entry should not generate full events that indicate a communication operation. This includes PTL_EVENT_GET, PTL_EVENT_PUT, PTL_EVENT_ATOMIC, and PTL_EVENT_SEARCH.

PTL_LE_EVENT_FLOWCTRL_DISABLE	Specifies that this list entry should not generate a PTL_EVENT_PT_DISABLED full event indicating a flow control failure when the current list entry generated the failure.
PTL_LE_EVENT_SUCCESS_DISABLE	Specifies that this list entry should not generate full events if the *ni_fail_type* would be **PTL_OK**. This flag does not affect counting events. Disabling full events for successful operations is useful in scenarios when a counting event is sufficient for completion, but more information is needed for error recovery.
PTL_LE_EVENT_OVER_DISABLE	Specifies that this list entry should not generate overflow list full events. This includes PTL_EVENT_PUT_OVERFLOW, PTL_EVENT_GET_OVERFLOW, PTL_EVENT_ATOMIC_OVERFLOW, and PTL_EVENT_FETCH_ATOMIC_OVERFLOW.
PTL_LE_EVENT_UNLINK_DISABLE	Specifies that this list entry should not generate auto-unlink (PTL_EVENT_AUTO_UNLINK) or free (PTL_EVENT_AUTO_FREE) full events.
PTL_LE_EVENT_CT_COMM	Enable the counting of communication full events (PTL_EVENT_PUT, PTL_EVENT_GET, PTL_EVENT_ATOMIC).
PTL_LE_EVENT_CT_OVERFLOW	Enable the counting of overflow events (PTL_EVENT_PUT_OVERFLOW, PTL_EVENT_GET_OVERFLOW, PTL_EVENT_ATOMIC_OVERFLOW, PTL_EVENT_FETCH_ATOMIC_OVERFLOW).
PTL_LE_EVENT_CT_BYTES	By default, counting events count events. When set, this option causes bytes to be counted instead for success events. Byte counts must be incremented exactly once per operation. The increment is by the number of bytes counted (*mlength*). Failure events always increment the count by one.

Discussion: *When the PTL_LE_USE_ONCE option is set, an event associated with a target side operation (e.g. a PTL_EVENT_PUT full event) also implies that the associated list entry has unlinked; hence, it is safe on these list entries to set the PTL_LE_EVENT_UNLINK_DISABLE option.*

PTL_LE_EVENT_FLOWCTRL_DISABLE only disables flow control events which are the direct result of an incoming message matching the current list entry. This includes a message matching the list entry but the associated event queue is full or a message matching a list entry in the overflow list but the unexpected headers list is full. If flow control is enabled on the portal table entry and a message does not match in either the priority or overflow lists, a PTL_EVENT_PT_DISABLED event is always generated.

3.11.2 PtlLEAppend

The **PtlLEAppend()** function creates a single list entry and appends this entry to the end of the list specified by *ptl_list* associated with the portal table entry specified by *pt_index* for the portal table for *ni_handle*.

When a list entry is posted to a priority list, the unexpected list is checked to see if a message has arrived prior to posting the list entry. If so, an appropriate overflow full event is generated, the matching header is removed from the unexpected list, and a list entry with the PTL_LE_USE_ONCE option is not inserted into the priority list. If a persistent list entry is posted to the priority list, it may cause multiple overflow events to be generated, one for every matching entry in the unexpected list. No permissions check is performed on a matching message in the unexpected list. No searching of the unexpected list is performed when a list entry is posted to the overflow list. When the list entry has been linked (inserted) into the specified list, a PTL_EVENT_LINK event is generated.

Discussion: *Generally speaking, the user should attempt to insure that persistent list entries (or match list entries) are inserted before messages arrive that match them. Inserts of persistent entries could have*

unexpected performance and resource usage characteristics if a large unexpected list has accumulated, since a ***PtlLEAppend()*** *that appends a persistent LE can cause multiple matches.*

List Entry Type Constants (ptl_list_t)

PTL_PRIORITY_LIST	The priority list associated with a portal table entry
PTL_OVERFLOW_LIST	The overflow list associated with a portal table entry

Function Prototype for PtlLEAppend

```
int PtlLEAppend(ptl_handle_ni_t ni_handle,
                ptl_pt_index_t pt_index,
                const ptl_le_t *le,
                ptl_list_t ptl_list,
                void *user_ptr,
                ptl_handle_le_t *le_handle);
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal table index where the list entry should be appended.
le	**input**	Provides initial values for the user-visible parts of a list entry. Other than its use for initialization, there is no linkage between this structure and the list entry maintained by the API.
ptl_list	**input**	Determines whether the list entry is appended to the priority list or the overflow list.
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. 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 full events associated with operations on this list entry.
le_handle	**output**	On successful return, this location will hold the newly created list entry handle.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the match list entry.
PTL_LIST_TOO_LONG	Indicates that the resulting list is too long. The maximum length for a list is defined by the interface.

Discussion: *Tying commands to a user-defined value is useful at the target when the command 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 list entries by the MPI implementation without a table look up or a search for the appropriate MPI Request.

3.11.3 PtlLEUnlink

The **PtlLEUnlink()** function can be used to unlink a list entry from a list. If **PtlLEUnlink()** returned **PTL_OK**, it is an error to use the list entry handle after the call to **PtlLEUnlink()**. **PtlLEUnlink()** will return **PTL_IN_USE** if the list entry is on the overflow list and has associated unexpected headers.

PtlLEUnlink() is frequently used to implement the cancel of receive operations in higher level protocols. If the list entry handle passed to **PtlLEUnlink()** has pending operations, e.g., an unfinished *put* operation or the list entry is in the overflow list and there are unexpected headers associated with the list entry, then **PtlLEUnlink()** will return **PTL_IN_USE**, and the list entry will not be unlinked. An implementation must ensure that list entry handles remain valid for calls to **PtlLEUnlink()** until the next call to **PtlLEAppend()** after the last event associated with the list entry is delivered to an event queue or counting event. If the list entry has been unlinked before a call to **PtlLEUnlink()** but before the next call to **PtlLEAppend()**, **PtlLEUnlink()** must return **PTL_IN_USE**.

IMPLEMENTATION NOTE 9:

PtlLEUnlink() and unlinked handles

PtlLEUnlink() may be used to unlink list entries which are use-once. In this case, there is a race condition between a network operation causing a list entry to unlink and the list entry being explicitly unlinked. Requiring the handle to remain valid until the next call to **PtlLEAppend()** allows higher level protocols to implement the serialization necessary to prevent such race conditions from impacting correctness. A Portals implementation does not need to limit the lifespan of handles to that specified. For example, a generation counter embedded in the handle may allow the handle to remain valid for the purposes of **PtlLEUnlink()** for significantly longer than specified.

Function Prototype for PtlLEUnlink

```
int PtlLEUnlink(ptl_handle_le_t le_handle);
```

Arguments

le_handle	**input**	The list entry handle to be unlinked.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_IN_USE	Indicates that the list entry has pending operations and cannot be unlinked.

3.11.4 PtlLESearch

The **PtlLESearch()** function is used to search for a message in the unexpected list associated with a specific portal table entry specified by *pt_index* for the portal table for *ni_handle*. **PtlLESearch()** uses the exact same search of the unexpected list as **PtlLEAppend()**; however, the list entry specified in the **PtlLESearch()** call is never linked into a priority list.

The **PtlLESearch()** function can be called in two modes. If *ptl_search_op* is set to `PTL_SEARCH_ONLY`, the unexpected list is searched, but matching entries are left in the list. If *ptl_search_op* is set to `PTL_SEARCH_DELETE`, the unexpected list is searched and any matching items are deleted. When used with `PTL_SEARCH_ONLY`, a `PTL_EVENT_SEARCH` event with *ni_fail_type* `PTL_NI_OK` is generated when a matching message is found in the unexpected list. When used with `PTL_SEARCH_DELETE`, the event that is generated corresponds to the type of operation that is found (e.g. `PTL_EVENT_PUT_OVERFLOW`, `PTL_EVENT_GET_OVERFLOW`, `PTL_EVENT_ATOMIC_OVERFLOW`, or `PTL_EVENT_FETCH_ATOMIC_OVERFLOW`). In either case, if no matching message is found, a `PTL_EVENT_SEARCH` event is generated with a failure indication of `PTL_NI_NO_MATCH`. If the list entry specified in the **PtlLESearch()** call is persistent, an event is generated for every match in the unexpected list. No permissions check is performed during search; only matching criteria are used to determine if an event should be generated. Users should use the generated event data to perform any required permissions check.

Event generation for the search functions works just as it would for an append function. If a search is performed with full events disabled (either through option or through the absence of an event queue on the portal table entry), the search will succeed, but no full events will be generated. Status registers, however, are handled slightly differently for a search in that a **PtlLESearch()** never causes a status register to be incremented.

> **Discussion**: *Searches with persistent entries could have unexpected performance and resource usage characteristics if a large overflow list has accumulated, since a **PtlLESearch()** that uses a persistent LE can cause multiple matches.*

List Entry Search Operation Constants (ptl_search_op_t)

`PTL_SEARCH_ONLY`	Use the LE/ME to search the overflow list, without consuming an item in the list.
`PTL_SEARCH_DELETE`	Use the LE/ME to search the overflow list and delete the item from the list.

Function Prototype for PtlLESearch

```
int PtlLESearch(ptl_handle_ni_t ni_handle,
                ptl_pt_index_t pt_index,
                const ptl_le_t *le,
                ptl_search_op_t ptl_search_op,
                void *user_ptr);
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal table index that should be searched.

le	**input**	Provides values for the user-visible parts of a list entry to use for searching.
ptl_search_op	**input**	Determines whether the function only searches the list or searches the list and deletes the matching entries from the list.
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. 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 full events associated with operations on this list entry.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

Discussion: *Tying commands to a user-defined value is useful at the target when the command 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 match list entries by the MPI implementation without a table look up or a search for the appropriate MPI Request.*

3.12 Match List Entries and Matching Lists

Matching list entries add matching semantics to the basic list constructs. Each match list entry (ME) adds a set of match criteria to the basic memory region description in the list entry. The match criteria added can be used to reject incoming requests based on process identifier or the match bits provided in the request. The **PtlMEAppend()** function appends a single match list entry to the specified portal index and returns the match list entry handle. Matching list entries can be dynamically removed from a list using the **PtlMEUnlink()** function.

Like a list entry, a match list entry describes a memory region using a base address and length. A zero-length list entry may be created by setting *start* to NULL and *length* to 0. Zero-length buffers (NULL ME) are useful to record events. If truncation is not disabled, messages that are outside the bounds of the ME are truncated to zero bytes (e.g. zero-length buffers or an offset beyond the length of the ME). If the interface set the `PTL_TARGET_BIND_INACCESSIBLE` bit in the *features* field of the *actual* limits (See Section 3.6.1), then it is not a requirement for all of the memory described by the match list entry to be allocated or accessible within the application. For example, an application could create a match list entry that covers the entire virtual address range by setting *start* to NULL and *length* to `PTL_SIZE_MAX`, even though the entire region is not currently allocated (See Implementation Note 8. If an incoming operation (e.g. *put*) attempts to access an unallocated region of the ME, the implementation may either cause a segmentation fault of the application or may simply fail the operation. If a full event is delivered, it must set *ni_fail_type* to `PTL_NI_SEGV`. The target may, however, set the `PTL_LE_IS_ACCESSIBLE` option to indicate that the entire memory space described by the ME is accessible. If the match list entry sets the `PTL_IOVEC` option, the memory region(s) described by the **ptl_iovec_t** must all be accessible within the application.

Matching list entries can be appended to either the priority list or the overflow list associated with a portal table entry; however, when attached to an overflow list, additional semantics are implied that require the implementation to track messages that arrive in match list entries. Essentially, the memory region identified is provided to the implementation for use in managing unexpected messages; however, the application may use the match bits and other matching criteria to further constrain how these buffers are used. Buffers provided in the overflow list will post a full event

(PTL_EVENT_AUTO_UNLINK) when the buffer space has been consumed, to notify the application that more buffer space may be needed. When the application is free to reuse the buffer (i.e. the implementation is done with it), another full event (PTL_EVENT_AUTO_FREE) will be posted. The PTL_EVENT_AUTO_FREE full event will be posted after *all* other events associated with the buffer have been posted to the event queue.

Incoming match bits are compared to the match bits stored in the match list entry using the ignore bits as a mask. An optimized version of this is shown in the following code fragment:

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

Discussion: *It is the responsibility of the application to ensure that the implementation has sufficient buffer space to manage unexpected messages. Failure to do will cause messages to be dropped. The PTL_EVENT_ACK at the initiator will indicate the failure as described in Section 3.13.3. Note that overflow events can readily exhaust the event queue. Proper use of the API will generally require the application to post at least two (and typically several) buffers so that the application has time to notice the PTL_EVENT_AUTO_UNLINK and replace the buffer.*

It is the responsibility of the implementation to determine when a buffer unlinked from an overflow list can be reused. It must note that it is no longer holding state associated with the buffer and deliver a PTL_EVENT_AUTO_FREE full event after all other events associated with that buffer have been delivered.

Match list entries may only be appended to a matching network interface. The interpretation of the *match_id* field in a match list entry is determined by whether the network interface is physically or logically addressed.

3.12.1 The Match List Entry Type

The **ptl_me_t** type defines the visible parts of a match list entry. Values of this type are used to initialize and update the match list entries.

```
typedef struct {
    void *start;
    ptl_size_t length;
    ptl_handle_ct_t ct_handle;
    ptl_uid_t uid;
    unsigned int options;
    ptl_process_t match_id;
    ptl_match_bits_t match_bits;
    ptl_match_bits_t ignore_bits;
    ptl_size_t min_free;
} ptl_me_t;
```

Members

start, *length*

Specify the memory region associated with the match list entry. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. There are no restrictions on buffer alignment, the starting address or the length of the region; although messages that are not natively aligned (e.g. to a four byte or eight byte boundary) may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

ct_handle

A handle for counting events associated with the memory region. If this argument is `PTL_CT_NONE`, operations performed on this match list entry are not counted.

min_free

When the unused portion of a match list entry (length - local offset) falls below this value, the match list entry automatically unlinks . A *min_free* value of 0 disables the *min_free* capability (the free space cannot fall below 0). This value is only used if `PTL_ME_MANAGE_LOCAL` is set.

uid

Specifies the user ID that may access this match list entry. The user ID may be set to a wildcard (`PTL_UID_ANY`). If the access control check fails, then the message is dropped without modifying Portals state. This is treated as a permissions failure and the status register indexed by `PTL_SR_PERMISSION_VIOLATIONS` is incremented. This failure is also indicated to the initiator. If a full event is delivered to the initiator, the *ni_fail_type* in the `PTL_EVENT_ACK` full event must be set to `PTL_NI_PERM_VIOLATION`.

options

Specifies the behavior of the match list entry. 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 and control event delivery. Values for this argument can be constructed using a bitwise OR of the following values:

`PTL_ME_OP_PUT`

Specifies that the match list entry will respond to *put* operations. By default, match list entries reject *put* operations. If a *put* operation targets a list entry where `PTL_ME_OP_PUT` is not set, it is treated as an operations failure and `PTL_SR_OPERATION_VIOLATIONS` is incremented. If a full event is delivered to the initiator, the *ni_fail_type* in the `PTL_EVENT_ACK` event must be set to `PTL_NI_OP_VIOLATION`.

`PTL_ME_OP_GET`

Specifies that the match list entry will respond to *get* operations. By default, match list entries reject *get* operations. If a *get* operation targets a list entry where `PTL_ME_OP_GET` is not set, it is treated as an operations failure and `PTL_SR_OPERATION_VIOLATIONS` is incremented. If a full event is delivered to the initiator, the *ni_fail_type* in the `PTL_EVENT_ACK` event must be set to `PTL_NI_OP_VIOLATION`.

Note: It is not considered an error to have a match list entry that responds to both *put* and *get* operations. In fact, a match list entry must be configured to respond to both put and get operations to properly handle a **PtlFetchAtomic()** or **PtlSwap()** operation.

PTL_ME_MANAGE_LOCAL	Specifies that the offset used in accessing the memory region is managed locally. By default, the offset is in the incoming message. 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 match list entry. If both *put* and *get* operations are performed on a match list entry, the value of that single variable is updated each time.
PTL_ME_NO_TRUNCATE	Specifies that the length provided in the incoming request cannot be reduced to match the memory available in the region. This will cause the matching to fail for a match list entry and continue with the next entry. (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 truncated.
PTL_ME_USE_ONCE	Specifies that the match list entry will only be used once and then automatically unlinked by the implementation. If this option is not set, the match list entry persists until it is explicitly unlinked or another unlink condition is triggered.
PTL_ME_MAY_ALIGN	Indicate that messages deposited into this match list entry may be aligned by the implementation to a performance optimizing boundary. Essentially, this is a performance hint to the implementation to indicate that the application does not care about the specific placement of the data. This option is only relevant when the PTL_ME_MANAGE_LOCAL option is set.
PTL_ME_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. See Section 3.13.3 for exceptions to this rule. This applies to both standard and counting events. Acknowledgments are never sent for *get* operations. The data sent in the *reply* serves as an implicit acknowledgment.
PTL_ME_UNEXPECTED_HDR_DISABLE	Specifies that the header for a message delivered to this match list entry should not be added to the unexpected list. This option only has meaning if the match list entry is inserted into the overflow list. By creating a match list entry which truncates messages to zero bytes, disables comm events, and sets this option, a user may create a match list entry which consumes no target side resources.
PTL_IOVEC	Specifies that the start argument is a pointer to an array of type **ptl_iovec_t** (Section 3.10.2) and the length argument is the length of the array. This allows for a scatter/gather capability for match list entries. A scatter/gather match list entry behaves exactly as a match list entry that describes a single virtually contiguous region of memory. All other semantics are identical.
PTL_ME_IS_ACCESSIBLE	Indicate that this match list entry only contains memory addresses that are accessible by the application.
PTL_ME_EVENT_LINK_DISABLE	Specifies that this match list entry should not generate a PTL_EVENT_LINK full event indicating the list entry successfully linked.
PTL_ME_EVENT_COMM_DISABLE	Specifies that this match list entry should not generate full events that indicate a communication operation. This includes PTL_EVENT_GET, PTL_EVENT_PUT, PTL_EVENT_ATOMIC, and PTL_EVENT_SEARCH.
PTL_ME_EVENT_FLOWCTRL_DISABLE	Specifies that this match list entry should not generate a PTL_EVENT_PT_DISABLED full event that indicate a flow control failure.

PTL_ME_EVENT_SUCCESS_DISABLE	Specifies that this match list entry should not generate full events if the *ni_fail_type* would be **PTL_OK**. This flag does not affect counting events. Disabling full events for successful operations is useful in scenarios when a counting event is sufficient for completion, but more information is needed for error recovery.
PTL_ME_EVENT_OVER_DISABLE	Specifies that this match list entry should not generate overflow list full events. This includes PTL_EVENT_PUT_OVERFLOW, PTL_EVENT_GET_OVERFLOW, PTL_EVENT_ATOMIC_OVERFLOW, and PTL_EVENT_FETCH_ATOMIC_OVERFLOW.
PTL_ME_EVENT_UNLINK_DISABLE	Specifies that this match list entry should not generate auto-unlink (PTL_EVENT_AUTO_UNLINK) or free (PTL_EVENT_AUTO_FREE) full events.
PTL_ME_EVENT_CT_COMM	Enable the counting of communication events (PTL_EVENT_PUT, PTL_EVENT_GET, PTL_EVENT_ATOMIC).
PTL_ME_EVENT_CT_OVERFLOW	Enable the counting of overflow events (PTL_EVENT_PUT_OVERFLOW, PTL_EVENT_GET_OVERFLOW, PTL_EVENT_ATOMIC_OVERFLOW, PTL_EVENT_FETCH_ATOMIC_OVERFLOW).
PTL_ME_EVENT_CT_BYTES	By default, counting events count events. When set, this option causes bytes to be counted instead for success events. Byte counts must be incremented exactly once per operation. The increment is by the number of bytes counted (*mlength*). Failure events always increment the count by one.
match_id	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 physical identifiers in the **ptl_process_t** structure, or PTL_RANK_ANY can be used to wildcard the rank for logical addressing.
match_bits, *ignore_bits*	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.

Discussion: *The default behavior from Portals 3.3 (no truncation and locally managed offsets) has been changed to match the default semantics of the list entry, which does not provide matching.*

When the PTL_ME_USE_ONCE option is set, an event associated with a target side operation (e.g. a PTL_EVENT_PUT event) also implies that the associated match list entry has unlinked; hence, it is safe on these match list entries to set the PTL_ME_EVENT_UNLINK_DISABLE option.

PTL_ME_EVENT_FLOWCTRL_DISABLE only disables flow control events which are the direct result of an incoming message matching the current match list entry. This includes a message matching the match list entry but the associated event queue is full or a message matching a match list entry in the overflow list but the unexpected headers list is full. If flow control is enabled on the portal table entry and a message does not match in either the priority or overflow lists, a PTL_EVENT_PT_DISABLED event is always generated.

Although the MD, ME, and LE can all map inaccessible memory, only the ME and LE have an option to allow the user to indicate to the implementation that the entire region is accessible. This is because the typical usage model for the MD is expected to bind inaccessible memory, while a very common usage model for both the ME and LE is expected to only use accessible memory.

3.12.2 PtlMEAppend

The **PtlMEAppend()** function creates a single match list entry. If `PTL_PRIORITY_LIST` or `PTL_OVERFLOW_LIST` is specified by *ptl_list*, this entry is appended to the end of the appropriate list specified by *ptl_list* associated with the portal table entry specified by *pt_index* for the portal table for *ni_handle*.

When a match list entry is posted to the priority list, the unexpected list is searched to see if a matching message has been delivered in the overflow list prior to the posting of the match list entry. If so, an appropriate overflow event is generated, the matching header is removed from the unexpected list, and a match list entry with the `PTL_ME_USE_ONCE` option is not inserted into the priority list. If a persistent match list entry is posted to the priority list, it may cause multiple overflow events to be generated, one for every matching entry in the unexpected list. No permissions checking is performed on a matching message in the unexpected list. No searching of the unexpected list is performed when a match list entry is posted to the overflow list. When the list entry has been linked (inserted) into the specified list, a `PTL_EVENT_LINK` event is generated.

> **Discussion**: *Generally speaking, the user should attempt to insure that persistent match list entries (or simple list entries) are inserted before messages arrive that match them. Appending of persistent entries could have unexpected performance and resource usage characteristics if a large unexpected list has accumulated, since a **PtlMEAppend()** that appends a persistent ME can cause multiple matches.*

See the **PtlLEAppend()** definition in Section 3.11.2 for the definition of **ptl_list_t**.

Function Prototype for PtlMEAppend

```
int PtlMEAppend(ptl_handle_ni_t ni_handle,
                ptl_pt_index_t pt_index,
                const ptl_me_t *me,
                ptl_list_t ptl_list,
                void *user_ptr,
                ptl_handle_me_t *me_handle);
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal table index where the match list entry should be appended.
me	**input**	Provides initial values for the user-visible parts of a match list entry. Other than its use for initialization, there is no linkage between this structure and the match list entry maintained by the API.
ptl_list	**input**	Determines whether the match list entry is appended to the priority list or the overflow list.
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. 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 full events associated with operations on this match list entry.
me_handle	**output**	On successful return, this location will hold the newly created match list entry handle.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the match list entry.
PTL_LIST_TOO_LONG	Indicates that the resulting list is too long. The maximum length for a list is defined by the interface.

Discussion: *Tying commands to a user-defined value is useful at the target when the command 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 match list entries by the MPI implementation without a table look up or a search for the appropriate MPI Request.*

IMPLEMENTATION NOTE 10:

Checking *match_id* Argument

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

3.12.3 PtlMEUnlink

The **PtlMEUnlink()** function can be used to unlink a match list entry from a list. If **PtlMEUnlink()** returned **PTL_OK**, it is an error to use the match list entry handle after the call to **PtlMEUnlink()**. **PtlMEUnlink()** should return **PTL_IN_USE** if the match list entry is on the overflow list and has associated unexpected headers.

PtlMEUnlink() is frequently used to implement the cancel of receive operations in higher level protocols. If the list entry handle passed to **PtlMEUnlink()** has pending operations, e.g., an unfinished *put* operation, then **PtlMEUnlink()** will return **PTL_IN_USE**, and the list entry will not be unlinked. An implementation must ensure that list entry handles remain valid for calls to **PtlMEUnlink()** until the next call to **PtlMEAppend()** after the last event associated with the list entry is delivered to an event queue or counting event (See Implementation Note 9). If the match list entry has been unlinked before a call to **PtlMEUnlink()** but before the next call to **PtlMEAppend()**, **PtlMEUnlink()** must return **PTL_IN_USE**.

Function Prototype for PtlMEUnlink

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

Arguments

me_handle	**input**	The match list entry handle to be unlinked.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_IN_USE	Indicates that the match list entry has pending operations and cannot be unlinked.

3.12.4 PtlMESearch

The **PtlMESearch()** function is used to search for a message in the unexpected list associated with a specific portal table entry specified by *pt_index* for the portal table for *ni_handle*. **PtlMESearch()** uses the exact same search of the unexpected list as **PtlMEAppend()**; however, the match list entry specified in the **PtlMESearch()** call is never linked into a priority list.

The **PtlMESearch()** function can be called in two modes. If *ptl_search_op* is set to `PTL_SEARCH_ONLY`, the unexpected list is searched to support the MPI_Probe functionality. If *ptl_search_op* is set to `PTL_SEARCH_DELETE`, the unexpected list is searched and any matching items are deleted from the list. When used with `PTL_SEARCH_ONLY`, a `PTL_EVENT_SEARCH` event with *ni_fail_type* `PTL_NI_OK` is generated when a matching message was found in the unexpected list. When used with `PTL_SEARCH_DELETE`, the event that is generated corresponds to the type of operation that is found (e.g. `PTL_EVENT_PUT_OVERFLOW`, `PTL_EVENT_GET_OVERFLOW`, `PTL_EVENT_ATOMIC_OVERFLOW`, or `PTL_EVENT_FETCH_ATOMIC_OVERFLOW`). In either case, if no matching message is found, a `PTL_EVENT_SEARCH` event is generated with a failure indication of `PTL_NI_NO_MATCH`. If the match list entry specified in the **PtlMESearch()** call is persistent, a full event is generated for every match in the unexpected list. No permissions checking is performed during search; only matching criteria are used to determine if an event should be generated. Users should use the generated event data to perform any required permissions check.

Event generation for the search functions works just as it would for an append function. If a search is performed with full events disabled (either through option or through the absence of an event queue on the portal table entry), the search will succeed, but no events will be generated. Status registers, however, are handled slightly differently for a search in that a **PtlMESearch()** never causes a status register to be incremented.

See the **PtlLESearch()** definition in Section 3.11.4 for the definition of *ptl_search_op* and important notes associated with implementing and using **PtlMESearch()**.

Function Prototype for PtlMESearch

```
int PtlMESearch(ptl_handle_ni_t ni_handle,
                ptl_pt_index_t pt_index,
                const ptl_me_t *me,
                ptl_search_op_t ptl_search_op,
                void *user_ptr);
```

Arguments

ni_handle	**input**	The interface handle to use.
pt_index	**input**	The portal table index that should be searched.
me	**input**	Provides values for the user-visible parts of a match list entry to use for searching.

ptl_search_op	**input**	Determines whether the function only searches the list or searches the list and deletes the matching entries from the list.
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. 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 full events associated with operations on this match list entry.

Return Codes

PTL_OK	Indicates success.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

Discussion: *Tying commands to a user-defined value is useful at the target when the command 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 match list entries by the MPI implementation without a table look up or a search for the appropriate MPI Request.*

3.13 Events and Event Queues

Event queues are used to log operations performed on memory descriptors, list entries, match list entries, or portal table entries. In particular, they signal the end of a data transmission into or out of a memory region. They can also be used to hold acknowledgments for completed *put* operations and indicate when a list entry has been unlinked. Multiple memory descriptors or list entries can share a single event queue.

In addition to the **ptl_handle_eq_t** type, the Portals API defines two types associated with full events: The **ptl_event_kind_t** type is an integral type which defines the kinds of events that can be stored in an event queue. The **ptl_event_t** type defines the structure that is placed into event queues.

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 full 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 full event, and the **PtlEQPoll()** function can be used to test or wait on multiple event queues.

3.13.1 Kinds of Events

The Portals API defines sixteen types of events that can be logged:

Event Type Constants (ptl_event_kind_t)

`PTL_EVENT_GET`	A *get* operation completed at the *target*. Portals will not read from memory on behalf of this operation once this event has been logged.

PTL_EVENT_GET_OVERFLOW	A list entry posted by **PtlLEAppend()** or **PtlMEAppend()** matched a *get* header in the unexpected list.
PTL_EVENT_PUT	A *put* operation completed at the *target*. Portals will not alter memory on behalf of this operation once this event has been logged.
PTL_EVENT_PUT_OVERFLOW	A list entry posted by **PtlLEAppend()** or **PtlMEAppend()** matched a *put* header in the unexpected list.
PTL_EVENT_ATOMIC	An *atomic* operation that does not return data to the *initiator* completed at the *target*. Portals will not read from or alter memory on behalf of this operation once this event has been logged.
PTL_EVENT_ATOMIC_OVERFLOW	A list entry posted by **PtlLEAppend()** or **PtlMEAppend()** matched an *atomic* header in the unexpected list for an operation which does not return data to the *initiator*.
PTL_EVENT_FETCH_ATOMIC	An *atomic* operation that returns data to the initiator completed at the *target*. These include **PtlFetchAtomic()** and **PtlSwap()**. Portals will not read from or alter memory on behalf of this operation once this event has been logged.
PTL_EVENT_FETCH_ATOMIC_OVERFLOW	A list entry posted by **PtlLEAppend()** or **PtlMEAppend()** matched an *atomic* header in the unexpected list for an operation which returns data to the *initiator*.
PTL_EVENT_REPLY	A *reply* operation has completed at the *initiator*, either due to a *get* operation or an *atomic* which returned data to the initiator. This event is logged after the data (if any) from the reply has been written into the memory descriptor. Receipt of a PTL_EVENT_REPLY indicates remote completion of the operation.
PTL_EVENT_SEND	A *put* or *atomic* has completed at the *initiator*. This event is logged after it is safe to reuse the buffer, but does not mean the message has been processed by the *target*.
PTL_EVENT_ACK	An *acknowledgment* was received. This event is logged when the acknowledgment is received. Receipt of a PTL_EVENT_ACK indicates remote completion of the operation. Remote completion indicates that local completion has also occurred.
PTL_EVENT_PT_DISABLED	Resources exhaustion has occurred on this portal table entry, which has entered a flow control situation. See Section 2.7.
PTL_EVENT_LINK	A list entry posted by **PtlLEAppend()** or **PtlMEAppend()** has successfully linked into the specified list.
PTL_EVENT_AUTO_UNLINK	A list entry/match list entry was automatically unlinked (Sections 3.12.2 and 3.11.2). A PTL_EVENT_AUTO_UNLINK event is generated even if the list entry/match list entry passed into the **PtlLEAppend()**/**PtlMEAppend()** operation was marked with the PTL_LE_USE_ONCE/PTL_ME_USE_ONCE option and found a corresponding unexpected message before being "linked" into the priority list. A PTL_EVENT_AUTO_UNLINK must be delivered after all PTL_EVENT_GET, PTL_EVENT_PUT, PTL_EVENT_ATOMIC, and PTL_EVENT_FETCH_ATOMIC events associated with the list entry/match list entry have been delivered.
PTL_EVENT_AUTO_FREE	A list entry/match list entry previously automatically unlinked from the overflow list is now free to be reused by the application. A PTL_EVENT_AUTO_FREE event is generated when Portals will not generate any further events which resulted from messages delivered into the specified overflow list entry. This also indicates that the unexpected list contains no more items associated with this entry.

PTL_EVENT_SEARCH	A **PtlLESearch()** or **PtlMESearch()** call completed. If a matching message was found in the overflow list, PTL_NI_OK is returned in the *ni_fail_type* field of the event and the event queue entries are filled in as if it were an overflow event. Otherwise, a failure is recorded in the *ni_fail_type* field using PTL_NI_NO_MATCH, the *user_ptr* is filled in correctly, and the other fields are undefined.

Overflow events are used to indicate that a message matching the list entry or match list entry posted by **PtlLEAppend()** or **PtlMEAppend()** was previously delivered into the overflow list and its header was found in the unexpected list (See Section 2.4). The operation was processed as specified by the list entry in the overflow list to which it matched, meaning that all, some, or none of the message may have been written to or read from the matching list entry in the overflow list. The full event's *start* will point to the start of the message (or where the message was read, in the case of a *get* operation). The *rlength* and *mlength* of the full event may be used to determine whether the message was fully delivered or truncated.

> **Discussion**: *When an application wishes to record unexpected messages, it may place an entry on the overflow list which has no memory associated with it and truncates all messages to zero bytes. The hdr_data field, along with a higher-level protocol, may be used to complete the transaction at a later time. In the case of MPI, a number of match list entries on the overflow list with locally managed offsets may additionally be used to optimize unexpected short messages.*

3.13.2 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.

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 an *atomic* operation.

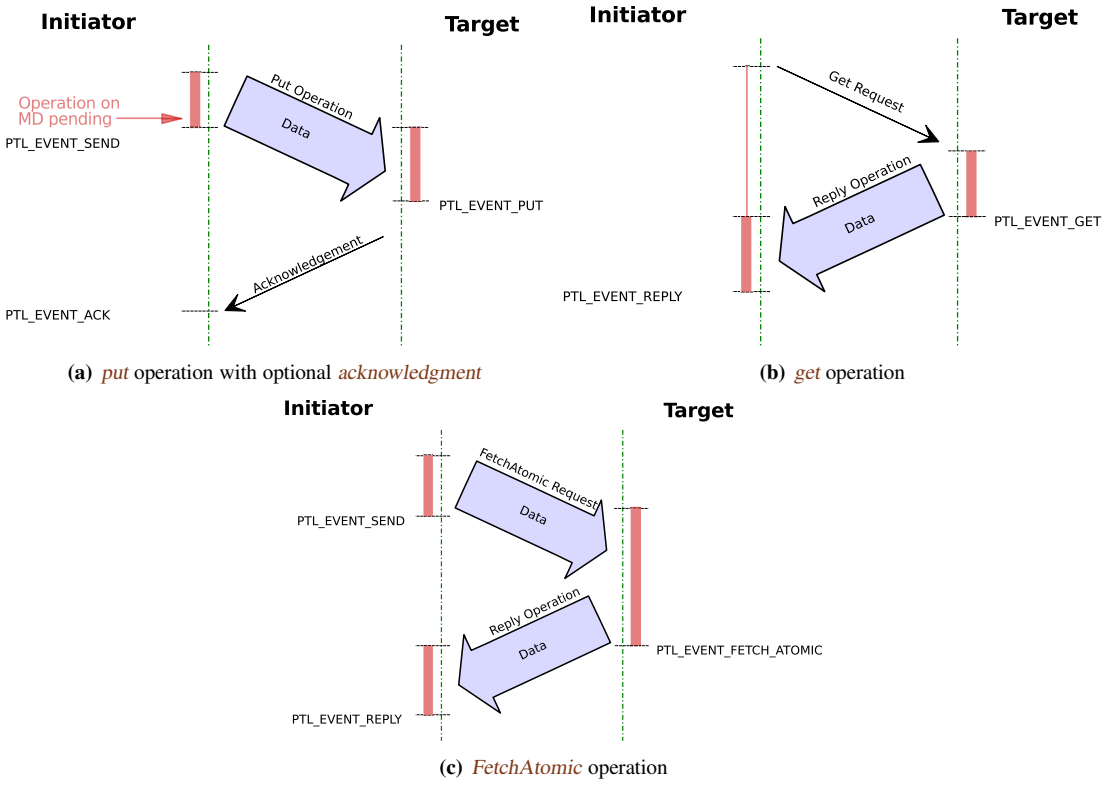
When the initiator of an operation receives a remote completion event (e.g. PTL_EVENT_ACK), local completion is also implied. While no ordering is required between local and remote completion events at the initiator (i.e. there is no guaranteed ordering between PTL_EVENT_SEND and PTL_EVENT_ACK for the same operation), a user may reuse a buffer after the remote completion event is received.

If, as a result of any of the operations shown in the diagrams of Figure 3.1, a match list entry is unlinked, then a PTL_EVENT_AUTO_UNLINK event is generated on the *target*. This is not shown in the diagrams. No *initiator* events are generated if the memory descriptor does not have an attached event queue. Similarly, no *target* events are generated if the portal table entry associated with the matched list entry does not have an attached event queue. See the description of PTL_EQ_NONE on page 47 of Section 3.10.1) for more information. The various types of events can also be disabled by type (e.g. see the description of PTL_ME_EVENT_COMM_DISABLE and PTL_ME_EVENT_UNLINK_DISABLE on page 66, also in Section 3.12.1.).

Table 3.2 summarizes the portals event types and where each event type may be generated.

3.13.3 Failure Notification

There are three ways in which operations may fail to complete successfully: the system (hardware or software) can fail in a way that makes the message undeliverable, a permissions violation can occur at the target, or resources can be exhausted at a target that has enabled flow-control. In any other scenario, every operation that is started will



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

(b) *get* operation

(c) *FetchAtomic* 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 or match list entries during those periods.

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 full event. Portals defines a number of event failure constants:

Event Failure Type Constants (ptl_ni_fail_t)

`PTL_NI_OK`	The operation causing the event was successful.
`PTL_NI_UNDELIVERABLE`	Indicates a system failure that prevents message delivery.
`PTL_NI_PT_DISABLED`	Indicates that the portal table entry at the *target* was disabled and did not process the operation, either because the entry was disabled with **PtlPTDisable()** or because the entry provides flow control and a resource has been exhausted. This failure type should only be returned on *initiator* events.
`PTL_NI_DROPPED`	Indicates that the message associated with this full event was dropped at the *target* for reasons other than a disabled portal table entry. This failure type should only be returned on *initiator* events.

PTL_NI_PERM_VIOLATION	Indicates that the remote Portals addressing has indicated a permissions violation for the operation that caused this event. This failure type should only be returned on *initiator* events.
PTL_NI_OP_VIOLATION	Indicates that the remote Portals addressing has indicated an operation violation for the operation that caused this event. This failure type should only be returned on *initiator* events.

To allow PTL_EVENT_SEND events to be local operations, all errors requiring remote information are delivered in PTL_EVENT_ACK or PTL_EVENT_REPLY events. This means that a PTL_EVENT_ACK will be delivered if it is requested, except when: 1) the message is successfully delivered at the target and the remote target has disabled event generation, 2) flow control is not enabled on the target portal table entry and the message does not match in either the priority list or overflow list or the message matches in the overflow list and the unexpected headers list is full, or 3) a locally generated failure is delivered in the PTL_EVENT_SEND. Certain classes of failures (e.g. a PTL_NI_UNDELIVERABLE that results from the network bifurcating) may require a local timeout to guarantee that the PTL_EVENT_ACK or PTL_EVENT_REPLY event is delivered.

Discussion: *Because remote errors are indicated in the PTL_EVENT_ACK or PTL_EVENT_REPLY events, the PTL_EVENT_SEND event only guarantees that the Portals implementation will not touch the buffer again. If the user intends to recover from a remote error, then the user cannot determine that an operation is done until the PTL_EVENT_ACK or PTL_EVENT_REPLY event is received.*

IMPLEMENTATION NOTE 11:

Completion of portals operations

Portals guarantees that every operation started will finish with an 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 an operation initiation and a corresponding event as short as possible. That includes operations that do not complete successfully. Timeouts of underlying protocols should be chosen accordingly.

3.13.4 The Event Structure

An event queue contains **ptl_event_t** structures. An operation on the *target* needs information about the local match list entry modified, the initiator of the operation and the operation itself. The *initiator*, in contrast, can track all information about the attempted operation; however, it does need the result of the operation and a pointer to resolve back to the local structure tracking the information about the operation.

Many fields in the **ptl_event_t** structure only have meaning for a subset of the event types. Further, an implementation is not required to provide all fields in the **ptl_event_t** structure when the event is reporting an error. Table 3.3 defines which fields are defined in both success and error conditions.

Table 3.2. Event Type Summary: A list of event types and where (*initiator* or *target*) they can occur.

Event Type	*initiator*	*target*
PTL_EVENT_GET		•
PTL_EVENT_GET_OVERFLOW		•
PTL_EVENT_PUT		•
PTL_EVENT_PUT_OVERFLOW		•
PTL_EVENT_ATOMIC		•
PTL_EVENT_ATOMIC_OVERFLOW		•
PTL_EVENT_FETCH_ATOMIC		•
PTL_EVENT_FETCH_ATOMIC_OVERFLOW		•
PTL_EVENT_REPLY	•	
PTL_EVENT_SEND	•	
PTL_EVENT_ACK	•	
PTL_EVENT_PT_DISABLED		•
PTL_EVENT_LINK		•
PTL_EVENT_AUTO_UNLINK		•
PTL_EVENT_AUTO_FREE		•
PTL_EVENT_SEARCH		•

```
typedef struct {
    void *start;
    void *user_ptr;
    ptl_hdr_data_t hdr_data;
    ptl_match_bits_t match_bits;
    ptl_size_t rlength;
    ptl_size_t mlength;
    ptl_size_t remote_offset;
    ptl_uid_t uid;
    ptl_process_t initiator; /* nid, pid or rank */
    ptl_event_kind_t type;
    ptl_list_t ptl_list;
    ptl_pt_index_t pt_index;
    ptl_ni_fail_t ni_fail_type;
    ptl_op_t atomic_operation;
    ptl_datatype_t atomic_type;
} ptl_event_t;
```

Members

start	The starting location (virtual, byte address) where the message has been placed. The *start* variable is the sum of the *start* variable in the list entry and the offset used for the operation. The offset can be determined by the operation (Section 3.15) for a remote managed match list entry or by the local memory descriptor (Section 3.12). In the case of iovecs, the *start* is still the first address where the message was placed or read from, even if multiple iovec entries were used. When an append call matches a message that has arrived in the overflow list, the start address points to the address in the overflow list where the matching message resides. This may require the application to copy the message to the desired buffer.
user_ptr	The user-specified value associated with the local command that generated the full event. Note that, unlike *hdr_data*, the *user_ptr* is a locally-generated value. For example, the *user_ptr* for a full event of type `PTL_EVENT_PUT` is the *user_ptr* specified to the associated call to **PtlLEAppend()** or **PtlMEAppend()**. For further discussion of *user_ptr*, see Section 3.12.2.
hdr_data	64 bits of out-of-band user data (Section 3.15.2).
match_bits	The match bits specified by the *initiator*. This field should be set to 0 if the event is associated with a non-matching list entry.
rlength	The length (in bytes) specified in the request.
mlength	The length (in bytes) of the data that was manipulated by the operation. For `PTL_EVENT_SEND` events, the manipulated length is the number of bytes sent, which may be larger than the number of bytes delivered (which can be determined by examining the *mlength* of the associated `PTL_EVENT_ACK` event). For `PTL_EVENT_PUT`, `PTL_EVENT_GET`, `PTL_EVENT_ATOMIC`, or `PTL_EVENT_FETCH_ATOMIC` events, the manipulated length is the number of bytes manipulated (delivered into or read from memory) at the target, which may be less than the *rlength* in the case of truncated operations. For `PTL_EVENT_SEARCH` and the overflow events, the manipulated length is the same value as the *mlength* returned in the corresponding `PTL_EVENT_PUT`, `PTL_EVENT_GET`, `PTL_EVENT_ATOMIC`, or `PTL_EVENT_FETCH_ATOMIC` event generated when the operation completed in the list entry on the overflow list.
remote_offset	The offset requested/used by the other end of the communication. At the initiator, this is the displacement (in bytes) into the memory region that the operation used at the target. The offset can be determined by the operation (Section 3.15) for a remote managed offset in a match list entry or by the match list entry (Section 3.12) at the target for a locally managed offset. At the target, this is the offset requested by the initiator.
uid	The user identifier of the *initiator*.
initiator	The identifier of the *initiator*.
type	Indicates the type of the full event.
ptl_list	The list entry or match list entry list in which the operation was delivered (See Sections 3.11.2 and 3.12.2).
pt_index	The portal table index where the message arrived.

ni_fail_type	Used to convey the failure of an operation. Success is indicated by `PTL_NI_OK`; see section 3.13.3.
atomic_operation	If this full event corresponds to an atomic operation, this indicates the atomic operation that was performed.
atomic_type	If this full event corresponds to an atomic operation, this indicates the data type of the atomic operation that was performed.

Discussion: *Notably, the full event structure does not contain a handle to the ME, LE, or MD that was associated with the full event. The user_ptr field is provided as the mechanism for the user to determine which ME, LE, or MD an even might be associated with.*

Table 3.3. Event Field Definition: Specification of which fields in a **ptl_event_t** structure are defined for a given event type. Fields marked with a ● are defined for both success and error conditions. Fields marked with a ○ are defined only for success conditions.

Event Type	type	initiator	pt_index	ptl_list	uid	match_bits	rlength	mlength	remote_offset	start	user_ptr	hdr_data	ni_fail_type	atomic_operation	atomic_type
PTL_EVENT_GET	●	●	●		●	●	○	●	○	●	●		●		
PTL_EVENT_GET_OVERFLOW	●	●	●		●	●	○	●	○	●	●		●		
PTL_EVENT_PUT	●	●	●		●	●	○	●	○	●	●	●	●		
PTL_EVENT_PUT_OVERFLOW	●	●	●		●	●	○	●	○	●	●	●	●		
PTL_EVENT_ATOMIC	●	●	●		●	●	○	●	○	●	●	●	●	●	●
PTL_EVENT_ATOMIC_OVERFLOW	●	●	●		●	●	○	●	○	●	●	●	●	●	●
PTL_EVENT_FETCH_ATOMIC	●	●	●		●	●	○	●	○	●	●	●	●	●	●
PTL_EVENT_FETCH_ATOMIC_OVERFLOW	●	●	●		●	●	○	●	○	●	●	●	●	●	●
PTL_EVENT_REPLY	●			○				○	○		●		●		
PTL_EVENT_SEND	●							○			●		●		
PTL_EVENT_ACK	●			○				○	○		●		●		
PTL_EVENT_PT_DISABLED	●		●										●		
PTL_EVENT_LINK	●		●								●		●		
PTL_EVENT_AUTO_UNLINK	●		●								●		●		
PTL_EVENT_AUTO_FREE	●		●								●		●		
PTL_EVENT_SEARCH	●	●	●		●	●	●	●	●	●	●	●	●	●	●

3.13.5 PtlEQAlloc

The **PtlEQAlloc()** function is used to build an event queue. An event queue has room for at least *count* number of full events. If the event queue overflows, older events will be overwritten by new ones in most situations. If flow control is enabled on the portal table entry (See Sections 3.7.1 and 2.7) for an incoming operation, events associated with that operation will not cause an overflow, but will instead trigger a flow control event.

Function Prototype for PtlEQAlloc

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

Arguments

ni_handle	**input**	The interface handle with which the event queue will be associated.
count	**input**	A hint as to the number of full events to be stored in the event queue. An implementation may provide space for more than the requested number of event queue slots.
eq_handle	**output**	On successful return, this location will hold the newly created event queue handle.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the event queue.

IMPLEMENTATION NOTE 12: Size of event queue and reserved space

Because flow control may be enabled on the portal table entries that this EQ is attached to, the implementation should insure that the space allocated for the EQ is large enough to hold the requested number of full events plus the number of portal table entries associated with this *ni_handle*. For each **PtlPTAlloc()** that enables flow control and uses a given EQ, one space should be reserved for a `PTL_EVENT_PT_DISABLED` full event associated with that EQ.

3.13.6 PtlEQFree

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

Function Prototype for PtlEQFree

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

Arguments

eq_handle	**input**	The event queue handle to be released.

Return Codes

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

3.13.7 PtlEQGet

The **PtlEQGet()** function is a non-blocking function that can be used to get the next event in an event queue. 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**	The event queue handle.
event	**output**	On successful return, this location will hold the values associated with the next event in the event queue. *event* must point to a valid **ptl_event_t** structure.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one full event between this full event and the last full 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_ARG_INVALID	Indicates that *eq_handle* is not a valid event queue handle.

3.13.8 PtlEQWait

The **PtlEQWait()** function can be used to block the calling process or thread until there is a full event in an event queue. 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);
```

Arguments

eq_handle	**input**	The event queue handle 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. *event* must point to a valid **ptl_event_t** structure.

Return Codes

PTL_OK	Indicates success.
PTL_EQ_DROPPED	Indicates success (i.e., an event is returned) and that at least one full event between this full event and the last full 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_ARG_INVALID	Indicates that *eq_handle* is not a valid event queue handle.
PTL_INTERRUPTED	Indicates that **PtlEQFree()** or **PtlNIFini()** was called by another thread while this thread was waiting in **PtlEQWait()**. See Implementation note 13 for more information.

IMPLEMENTATION NOTE 13: PTL_INTERRUPTED return code

While adding complexity to the implementation of **PtlEQWait()** and **PtlEQPoll()**, allowing **PtlEQFree()** or **PtlNIFini()** to interrupt the potentially blocking calls is necessary for failure tolerance.

3.13.9 PtlEQPoll

The **PtlEQPoll()** function can be used by the calling process to look for a full 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 full event will be returned in *event* and *which* will contain the index of the event queue from which the event was taken. In the event that multiple threads are polling the same event queue, **PtlEQPoll()** is guaranteed to wake exactly one thread, but the order in which they are awakened is not specified.

If **PtlEQPoll()** returns success, the corresponding full 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 may allow significant optimization. **PtlEQPoll()** should poll the list of queues in a round-robin fashion.

Function Prototype for PtlEQPoll

```
int PtlEQPoll(const ptl_handle_eq_t *eq_handles,
              unsigned int size,
              ptl_time_t timeout,
              ptl_event_t *event,
              unsigned 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 a full 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. *event* must point to a valid **ptl_event_t** structure.
which	**output**	On successful return, this location will contain the index into *eq_handles* 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 full event between this full event and the last full 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_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_EQ_EMPTY	Indicates that the timeout has been reached and all of the event queues are empty.
PTL_INTERRUPTED	Indicates that **PtlEQFree()** or **PtlNIFini()** was called by another thread while this thread was waiting in **PtlEQPoll()**. See Implementation note 13 for more information.

3.14 Lightweight Counting Events

Full events copy a significant amount of data from the implementation to the application. While this data is critical for many uses (e.g. MPI), other programming models (e.g. PGAS) require very little information about individual operations. To support lightweight operations, Portals provide a lightweight event mechanism known as counting events.

Counting events are similar in semantics and event occurrence to full events (See Section 3.13.2). A counting event may be independently enabled/disabled with options on the memory descriptor, list entry, or match list entry, similar to full events. Unlike full events, counting events are disabled by default and must be explicitly enabled for a given event type. Counting events are enabled by attaching a **ptl_handle_ct_t** to a memory descriptor or list entry and specifying which operations are to be counted in the options field. By default, counting events count the total number

of operations; however, a counting event may also count the number of bytes successfully manipulated for counted operations by setting an option on the associated memory descriptor or list entry.

Counting events introduce two additional types: a user-visible representation of the counting event itself, of type **ptl_ct_event_t**, and a handle to a counting event, of type **ptl_handle_ct_t**. A counting event is allocated through a call to **PtlCTAlloc()**, queried with **PtlCTGet()**, **PtlCTWait()**, or **PtlCTPoll()**, set with **PtlCTSet()**, incremented with **PtlCTInc()**, and freed through a call to **PtlCTFree()**. To mirror the failure semantics of the full events, counting events count success and failure events independently.

IMPLEMENTATION NOTE 14:

Minimizing cost of counting events

A quality implementation will attempt to minimize the cost of counting events. In many implementations, this can be done by making the **ptl_handle_ct_t** type a pointer to a **ptl_ct_event_t** structure and providing **PtlCTGet()**, **PtlCTWait()**, **PtlCTSet()**, and **PtlCTInc()** as macros which manipulate the internal structure. This may not be possible in hardware offload implementations, but **PtlCTGet()** should remain as close to a pair of loads in performance as possible.

Counting events are a critical component of triggered operations, described in Section 3.16.

3.14.1 The Counting Event Type

A *ct_handle* refers to a **ptl_ct_event_t** structure. The user visible portion of this structure contains both a count of succeeding events and a count of failing events.

```
typedef struct {
    ptl_size_t success;
    ptl_size_t failure;
} ptl_ct_event_t;
```

Members

success	A count associated with successful events that counts events or bytes.
failure	A count of the number of failed events associated with the counting event.

3.14.2 PtlCTAlloc

The **PtlCTAlloc()** function is used to allocate a counting event that counts either operations or bytes manipulated for operations on associated memory descriptors, list entries, and match list entries. While a **PtlCTAlloc()** call could be as simple as a malloc of a structure holding the counting event, it may be necessary to allocate the counting event in low memory or some other protected space. Also, it may be desirable to place all counting events in a pre-allocated array and make the *ct_handle* a simple index. A newly allocated counting event will have both the *success* and *failure* counts initialized to zero.

Function Prototype for PtlCTAlloc

```
int PtlCTAlloc(ptl_handle_ni_t ni_handle,
               ptl_handle_ct_t *ct_handle);
```

Arguments

ni_handle	**input**	The interface handle with which the counting event will be associated.
ct_handle	**output**	On successful return, this location will hold the newly created counting event handle.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
PTL_NO_SPACE	Indicates that there is insufficient memory to allocate the counting event.

3.14.3 PtlCTFree

The **PtlCTFree()** function releases the resources associated with a counting event. It is up to the user to ensure that no memory descriptors or match list entries are associated with the counting event before it is freed. On a successful return, the counting event has been released and is ready to be reallocated. As a side-effect of **PtlCTFree()**, any triggered operations waiting on the freed counting event whose thresholds have not been met will be deleted.

Function Prototype for PtlCTFree

```
int PtlCTFree(ptl_handle_ct_t ct_handle);
```

Arguments

ct_handle	**input**	The counting event handle to be released.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.14.4 PtlCTCancelTriggered

In certain circumstances, it may be necessary to cancel triggered operations that are pending. For example, an error condition may mean that a counting event will never reach the designated threshold. **PtlCTCancelTriggered()** is provided to handle these circumstances. Upon return from **PtlCTCancelTriggered()**, all triggered operations waiting on *ct_handle* are permanently deleted. The operations are not triggered and will not modify any application-visible state. All other state associated with *ct_handle* is left unchanged.

Function Prototype for PtlCTCancelTriggered

```
int PtlCTCancelTriggered(ptl_handle_ct_t ct_handle);
```

Arguments

ct_handle	**input**	The counting event handle associated with the triggered operations to be canceled.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.14.5 PtlCTGet

The **PtlCTGet()** function is used to obtain the current value of a counting event. Calling **PtlCTSet()** or **PtlCTFree()** in a separate thread while **PtlCTGet()** is executing may yield undefined results in the returned value.

Function Prototype for PtlCTGet

```
int PtlCTGet(ptl_handle_ct_t ct_handle,
             ptl_ct_event_t *event);
```

Arguments

ct_handle	**input**	The counting event handle.
event	**output**	On successful return, this location will hold the current value associated with the counting event. *event* must point to a valid **ptl_ct_event_t** structure.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.14.6 PtlCTWait

The **PtlCTWait()** function provides blocking semantics to wait for a counting event to reach a given value. **PtlCTWait()** returns when either the *success* field of a counting event is greater than or equal to the *test* value or when the *failure* field is non-zero. All threads that are waiting on a single counting event with a given *test* value will return from **PtlCTWait()** when that *test* value is reached.

Function Prototype for PtlCTWait

```
int PtlCTWait(ptl_handle_ct_t ct_handle,
              ptl_size_t test,
              ptl_ct_event_t *event);
```

Arguments

ct_handle	**input**	The counting event handle.
test	**input**	On successful return, the *success* field of the counting event will be greater than this value or the *failure* field of the counting event will be non-zero.
event	**output**	On successful return, this location will hold the current value associated with the counting event. *event* must point to a valid **ptl_ct_event_t** structure.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that *ct_handle* is not a valid counting event handle.
PTL_INTERRUPTED	Indicates that **PtlCTFree()** or **PtlNIFini()** was called by another thread while this thread was waiting in **PtlCTWait()**. See Implementation note 13 for more information.

3.14.7 PtlCTPoll

The **PtlCTPoll()** function can be used to look for one of an array of counting events where the success field has reached its respective test value. Should a counting event reach the test value for any of the counting events contained in the array of counting event handles, the value of the counting event will be returned in *event* and *which* will contain the index of the counting event from which the value was returned. **PtlCTPoll()** will also return whenever the failure field of any of the counting events is non-zero.

PtlCTPoll() provides a timeout to allow applications to poll, block for a fixed period, or block indefinitely. **PtlCTPoll()** should test the list of counting events in a round-robin fashion. This cannot guarantee fairness but meets common expectations.

Function Prototype for PtlCTPoll

```
int PtlCTPoll(const ptl_handle_ct_t *ct_handles,
              const ptl_size_t *tests,
              unsigned int size,
              ptl_time_t timeout,
              ptl_ct_event_t *event,
              unsigned int *which);
```

Arguments

ct_handles	**input**	An array of counting event handles. All of the handles must refer to the same interface.
tests	**input**	An array of success values. **PtlCTPoll()** returns when any counting event in *ct_handles* would return from **PtlCTWait()** with the corresponding *test* in *tests*.
size	**input**	Length of the *ct_handles* and *tests* arrays.
timeout	**input**	Time in milliseconds to wait for an event to occur on one of the counting event handles. The constant `PTL_TIME_FOREVER` can be used to indicate an infinite timeout.
event	**output**	On successful return, this location will hold the current value associated with the counting event that caused **PtlCTPoll()** to return. *event* must point to a valid **ptl_ct_event_t** structure.
which	**output**	On successful return, this location will contain the index into *ct_handles* of the counting event that reached its test value.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates an invalid argument (e.g. a bad *ct_handle*).
PTL_CT_NONE_REACHED	Indicates that none of the counting events reached their test before the timeout was reached.
PTL_INTERRUPTED	Indicates that **PtlCTFree()** or **PtlNIFini()** was called by another thread while this thread was waiting in **PtlCTPoll()**. See Implementation note 13 for more information.

3.14.8 PtlCTSet

The **PtlCTSet()** function is used to set a new value for a counting event. Each field in the counting event is updated atomically relative to other updates of that field. However, there is no guarantee that the two fields are updated atomically relative to each other. The counting event must be updated before returning from **PtlCTSet()**, however the update may not be immediately visible to **PtlCTGet()**, particularly in hardware offload implementations. Both the atomicity of field updates and the delay in updating the user-visible portions of the counting event may be visible to the user, but should not affect correctness in common usage scenarios.

Function Prototype for PtlCTSet

```
int PtlCTSet(ptl_handle_ct_t ct_handle,
             ptl_ct_event_t new_ct);
```

Arguments

ct_handle	**input**	The counting event handle.
new_ct	**input**	On successful return, the value of the counting event will have been set to this value.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.14.9 PtlCTInc

PtlCTInc() provides the ability to increment the *success* or *failure* field of a counting event. The update is atomic relative to other modifications of the counting event. To simplify implementation, the *increment* field can only be non-zero for either the *success* or *failure* field in a given call to **PtlCTInc()**. The counting event must be updated before returning from **PtlCTInc()**, however the update may not be immediately visible to **PtlCTGet()**, particularly in hardware offload implementations. This may be visible to the user, but should not affect correctness in common usage scenarios.

Function Prototype for PtlCTInc

```
int PtlCTInc(ptl_handle_ct_t ct_handle,
             ptl_ct_event_t increment);
```

Arguments

ct_handle	**input**	The counting event handle.
increment	**input**	On successful return, the value of the counting event will have been incremented by this value.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that *ct_handle* is not a valid counting event handle.

3.15 Data Movement Operations

The Portals API provides five data movement operations: **PtlPut()**, **PtlGet()**, **PtlAtomic()**, **PtlFetchAtomic()**, and **PtlSwap()**.

3.15.1 Portals Acknowledgment Type Definition

Portals *put* and *atomic* operations which do not return data may optional request an acknowledgment upon message delivery. Values of the type **ptl_ack_req_t** are used to specify the type of acknowledgment requested by the *initiator*. Acknowledgments are sent by the *target* when the operation has completed (i.e., when the data has been written to a list entry of the *target* process). When counting of acknowledgment events is enabled, the PTL_MD_EVENT_CT_BYTES option is set, and the operation is successful, the manipulated length (*mlength*) from the target is counted. If the event would indicate "failure" or the PTL_MD_EVENT_CT_BYTES option is not set, the number of acknowledgments is counted.

Ack Request Constants (ptl_ack_req_t)

PTL_ACK_REQ	An acknowledgement capable of generating both a full event and counting event is requested.
PTL_CT_ACK_REQ	An acknowledgement capable of generating a counting event is requested. A full event will not be generated, even if an event queue is associated with the memory descriptor.
PTL_OC_ACK_REQ	An acknowledgment capable of generating a counting event upon operation completion is requested. An operation is considered completed when it has successfully completed Portals operation processing at the *target*. PTL_OC_ACK_REQ does not support the PTL_MD_EVENT_CT_BYTES option. The operation completion acknowledgement will indicate success as long as operation processing completed successfully. A message being dropped due to a failure to match or a permissions violation does not represent an operational failure.
PTL_NO_ACK_REQ	No acknowledgement is requested.

Discussion: *The PTL_CT_ACK_REQ and PTL_OC_ACK_REQ acknowledgement types provide significantly weaker semantics than PTL_ACK_REQ, in that the acknowledgement from the target may only contain data necessary to generate a counting event, which may improve efficiency.*

The PTL_OC_ACK_REQ acknowledgement type is useful when only operation counting is required and it is known that there is a list entry at the target that will accept the message. The PTL_OC_ACK_REQ acknowledgement type may be more efficient in some implementations because the PTL_OC_ACK_REQ acknowledgement type communicates no information about the state of the target when the message arrived.

3.15.2 PtlPut

The **PtlPut()** function initiates an asynchronous *put* operation. There are several events associated with a *put* operation: completion of the send on the *initiator* node (PTL_EVENT_SEND) and the receipt of an acknowledgment (PTL_EVENT_ACK) indicating that the operation was accepted by the *target*. The event PTL_EVENT_PUT is used at the *target* node to indicate the end of data delivery. In addition, PTL_EVENT_PUT_OVERFLOW can be used on the *target* node when a new entry being appended to a priority list matches a message that arrived before the corresponding match list entry had been associated with the target portal table entry (Figure 3.1 on page 74).

These (local) events will be logged using full events in the event queue or counting events in the *ct_handle* associated with the memory descriptor (*md_handle*) used in the *put* operation. Using a memory descriptor that does not have either an associated event queue or counting event 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.

The local (*initiator*) offset is used to determine the starting address of the memory region within the region specified by the memory descriptor 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 PtlPut

```
int PtlPut(ptl_handle_md_t md_handle,
           ptl_size_t local_offset,
           ptl_size_t length,
           ptl_ack_req_t ack_req,
           ptl_process_t target_id,
           ptl_pt_index_t pt_index,
           ptl_match_bits_t match_bits,
           ptl_size_t remote_offset,
           void *user_ptr,
           ptl_hdr_data_t hdr_data);
```

Arguments

md_handle	**input**	The memory descriptor handle 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`, `PTL_EVENT_ACK`). If the memory descriptor has a counting event associated with it, it may optionally be used to record the same events.
local_offset	**input**	Offset from the start of the memory descriptor.
length	**input**	Length of the memory region to be sent.
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 or counting event **and** the target memory descriptor enables them.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process (only used when matching is enabled on the network interface).
remote_offset	**input**	The offset into the target memory region (used unless the *target* match list entry has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	A user-specified value that is associated with each command that can generate an event. 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 *initiator* full events associated with this *put* operation.
hdr_data	**input**	64 bits of user data that can be included in the message header. This data is written to the full event generated at the *target* by this operation.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

Discussion: *Tying commands to a user-defined value is useful for quickly locating a user data structure associated with the put operation. 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 a put operation completion full event by the MPI implementation without a table look up or a search for the appropriate MPI Request.*

3.15.3 PtlGet

The **PtlGet()** function initiates a remote read operation. There are two events associated with a get operation. When the data is sent from the *target* node, a `PTL_EVENT_GET` event is registered on the *target* node if the message matched in the priority list. The message can also match in the overflow list, which will cause a `PTL_EVENT_GET` event to be registered on the *target* node and will later cause a `PTL_EVENT_GET_OVERFLOW` to be registered on the *target* node when a matching entry is appended. In either case, when the data is returned from the *target* node, a `PTL_EVENT_REPLY` event is registered on the *initiator* node. (Figure 3.1)

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 PtlGet

```
int PtlGet(ptl_handle_md_t md_handle,
           ptl_size_t local_offset,
           ptl_size_t length,
           ptl_process_t target_id,
           ptl_pt_index_t pt_index,
           ptl_match_bits_t match_bits,
           ptl_size_t remote_offset,
           void *user_ptr);
```

Arguments

md_handle	**input**	The memory descriptor handle 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 full events, such as when the message receive has started. If the memory descriptor has a counting event associated with it, it may optionally be used to record the same events.
local_offset	**input**	Offset from the start of the memory descriptor.
length	**input**	Length of the memory region for the *reply*.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process (only used when matching is enabled on the network interface).
remote_offset	**input**	The offset into the target match list entry (used unless the target match list entry has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**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_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.15.4 Portals Atomics Overview

Portals defines three closely related types of atomic operations. The **PtlAtomic()** function is a one-way operation that performs an atomic operation on data at the *target* with the data in the specified memory descriptor. The **PtlFetchAtomic()** function extends **PtlAtomic()** to be an atomic fetch-and-update operation. The value at the *target* before the operation is returned in a *reply* message and placed into the *get* memory descriptor of the *initiator*. Finally, the **PtlSwap()** operation atomically swaps data (including compare-and-swap and swap under mask, which require an *operand* argument).

The length of the operations performed by a **PtlAtomic()** is restricted to no more than *max_atomic_size* bytes. The *max_atomic_size* limit also guarantees that any byte in an operation (whether an atomic operation or not) that is smaller than *max_atomic_size* will only be written once in the host memory. **PtlFetchAtomic()** and **PtlSwap()** operations can be up to *max_fetch_atomic_size* bytes, except for `PTL_CSWAP` and `PTL_MSWAP` operations and their variants, which are further restricted to the length of the longest native data type.

While the length of an atomic operation is potentially multiple data items, the granularity of the atomic access is limited to the basic datatype. That is, atomic operations from different sources may be interleaved at the level of the datatype being accessed. Furthermore, atomic operations are only atomic with respect to other calls to the Portals API on the same network interface (*ni_handle*). If a network interface returned `PTL_COHERENT_ATOMICS` in the *features* field of **PtlNIInit()**, atomic operations are atomic relative to processor-initiated atomic operations, as well as any other network interface that also returned `PTL_COHERENT_ATOMICS`. In addition, an implementation is only required to support Portals atomic operations that are natively aligned to the size of the datatype, but it may choose to provide support for unaligned accesses. If the list entry sets the `PTL_IOVEC` option, a single datatype may not span multiple iovec entries. Atomicity is only guaranteed for two atomic operations using the same datatype, and overlapping atomic operations that use different datatypes are not atomic with respect to each other. The routine **PtlAtomicSync()** is provided to enable the host (or atomic operations using other datatypes) to modify memory locations that have been previously touched by an atomic operation.

The *target* match list entry must be configured to respond to *put* operations and to *get* operations if a reply is desired. The *length* argument at the initiator is used to specify the size of the request.

There are several events that can be associated with atomic operations. When data is sent from the *initiator* node, a `PTL_EVENT_SEND` event is registered on the *initiator* node. It can be tracked in the event queue and/or in the counting event specified in the *put_md_handle*. The event `PTL_EVENT_ATOMIC` is registered on the *target* node to indicate completion of an atomic operation; and if data is returned from the *target* node, a `PTL_EVENT_REPLY` event is registered on the *initiator* node in the event queue and/or the counting event specified by the *get_md_handle*. Similarly, a `PTL_EVENT_ACK` can be registered on the *initiator* node in the event queue and/or counting event specified by the *put_md_handle* for the atomic operations that do not return data. Note that the target match list entry must have the `PTL_ME_OP_PUT` flag set and must also set the `PTL_ME_OP_GET` flag to enable a reply. As with other Portals operations, the delivery of an event indicates that the data for the associated atomic operation has been updated in application memory. This does not alleviate the requirement that all modifications of a memory location that is accessed by atomic operations must go through the Portals API.

The three atomic functions share two new arguments introduced in Portals 4.0: an operation (**ptl_op_t**) and a datatype (**ptl_datatype_t**), as described below.

Discussion: *To allow upper level libraries with both system defined datatype widths and fixed width datatypes to easily map to Portals, Portals provides fixed width integer types. The one exception is the long double floating-point types (`PTL_LONG_DOUBLE`). Because of the variability in long double encodings across systems and the lack of standard syntax for fixed width floating-point types, Portals uses a system defined width for `PTL_LONG_DOUBLE` and `PTL_LONG_DOUBLE_COMPLEX`.*

Atomic Operation Constants (ptl_op_t)

PTL_MIN	Compute and return the minimum of the initiator and target value.
PTL_MAX	Compute and return the maximum of the initiator and target value.
PTL_SUM	Compute and return the sum of the initiator and target value.
PTL_PROD	Compute and return the product of the initiator and target value.
PTL_LOR	Compute and return the logical OR of the initiator and target value.
PTL_LAND	Compute and return the logical AND of the initiator and target value.
PTL_BOR	Compute and return the bitwise OR of the initiator and target value.
PTL_BAND	Compute and return the bitwise AND of the initiator and target value.
PTL_LXOR	Compute and return the logical XOR of the initiator and target value.
PTL_BXOR	Compute and return the bitwise XOR of the initiator and target value.
PTL_SWAP	Swap the initiator and target value and return the target value.
PTL_CSWAP	A conditional swap. If the value of the operand is equal to the target value, the initiator and target value are swapped. The target value is always returned. This operation is limited to single data items.
PTL_CSWAP_NE	A conditional swap. If the value of the operand is not equal to the target value, the initiator and target value are swapped. The target value is always returned. This operation is limited to single data items.
PTL_CSWAP_LE	A conditional swap. If the value of the operand is less than or equal to the target value, the initiator and target value are swapped. The target value is always returned. This operation is limited to single data items.
PTL_CSWAP_LT	A conditional swap. If the value of the operand is less than the target value, the initiator and target value are swapped. The target value is always returned. This operation is limited to single data items.
PTL_CSWAP_GE	A conditional swap. If the value of the operand is greater than or equal to the target value, the initiator and target value are swapped. The target value is always returned. This operation is limited to single data items.
PTL_CSWAP_GT	A conditional swap. If the value of the operand is greater than the target value, the initiator and target value are swapped. The target value is always returned. This operation is limited to single data items.
PTL_MSWAP	A masked version of the swap operation. Update the bits of the target value that are set to 1 in the operand using the bits in the initiator value. Return the target value. This operation is limited to single data items.

Atomic Datatype Constants (ptl_datatype_t)

PTL_INT8_T	8-bit signed integer

PTL_UINT8_T	8-bit unsigned integer
PTL_INT16_T	16-bit signed integer
PTL_UINT16_T	16-bit unsigned integer
PTL_INT32_T	32-bit signed integer
PTL_UINT32_T	32-bit unsigned integer
PTL_INT64_T	64-bit signed integer
PTL_UINT64_T	64-bit unsigned integer
PTL_FLOAT	32-bit floating-point number
PTL_FLOAT_COMPLEX	32-bit floating-point complex number
PTL_DOUBLE	64-bit floating-point number
PTL_DOUBLE_COMPLEX	64-bit floating-point complex number
PTL_LONG_DOUBLE	System defined long double type
PTL_LONG_DOUBLE_COMPLEX	System defined long double complex type

The legal combinations of atomic operation type, datatype, and function call are shown in Table 3.4. Generally speaking, swap operations are limited to the **PtlSwap()** function and bitwise operation are limited to integral types.

Table 3.4. Legal Atomic Operation, Datatype, and Function Combinations

	Integral Types	Floating-Point Types	Complex Types	PtlAtomic()	PtlFetchAtomic()	PtlSwap()
PTL_MIN	•	•		•	•	
PTL_MAX	•	•		•	•	
PTL_SUM	•	•	•	•	•	
PTL_PROD	•	•	•	•	•	
PTL_LOR	•			•	•	
PTL_LAND	•			•	•	
PTL_BOR	•			•	•	
PTL_BAND	•			•	•	
PTL_LXOR	•			•	•	
PTL_BXOR	•			•	•	
PTL_SWAP	•	•	•			•
PTL_CSWAP	•	•	•			•
PTL_CSWAP_NE	•	•	•			•
PTL_CSWAP_LE	•	•				•
PTL_CSWAP_LT	•	•				•
PTL_CSWAP_GE	•	•				•
PTL_CSWAP_GT	•	•				•
PTL_MSWAP	•					•

3.15.5 PtlAtomic

The **PtlAtomic()** function initiates an asynchronous *atomic* operation. The events behave like the **PtlPut()** function (see Section 3.15.2), with the exception of the target side event, which is a PTL_EVENT_ATOMIC (and PTL_EVENT_ATOMIC_OVERFLOW) instead of a PTL_EVENT_PUT. Similarly, the arguments mirror **PtlPut()** with the

addition of a **ptl_datatype_t** and **ptl_op_t** to specify the datatype and operation being performed, respectively. Operations performed by **PtlAtomic()** are constrained to be no more than *max_atomic_size* bytes and must be aligned at the target to the size of **ptl_datatype_t** passed in the *datatype* argument. **PtlAtomic()** is not atomic relative to other host operations, except those requested through the Portals API.

Function Prototype for PtlAtomic

```
int PtlAtomic(ptl_handle_md_t md_handle,
              ptl_size_t local_offset,
              ptl_size_t length,
              ptl_ack_req_t ack_req,
              ptl_process_t target_id,
              ptl_pt_index_t pt_index,
              ptl_match_bits_t match_bits,
              ptl_size_t remote_offset,
              void *user_ptr,
              ptl_hdr_data_t hdr_data,
              ptl_op_t operation,
              ptl_datatype_t datatype);
```

Arguments

md_handle	**input**	The memory descriptor handle 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`, `PTL_EVENT_ACK`). If the memory descriptor has a counting event associated with it, it may optionally be used to record the same events.
local_offset	**input**	Offset from the start of the memory descriptor referenced by the *md_handle* to use for transmitted data.
length	**input**	Length of the memory region to be sent and/or received. The *length* field must be less than or equal to *max_atomic_size*.
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 or counting event **and** the target memory descriptor enables them.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory region (used unless the target match list entry has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	See the discussion for **PtlPut()**.
hdr_data	**input**	See the discussion for **PtlPut()**.
operation	**input**	The operation to be performed using the initiator and target data.
datatype	**input**	The type of data being operated on at the initiator and target.

Return Codes

PTL_OK	Indicates success.

PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.15.6 PtlFetchAtomic

The **PtlFetchAtomic()** function extends the **PtlAtomic()** function to return the value from the target *prior to the operation being performed*. When data is sent from the *initiator* node, a `PTL_EVENT_SEND` event is registered on the *initiator* node in the event queue and/or the counting event specified by the *put_md_handle*. The event `PTL_EVENT_FETCH_ATOMIC` (and potentially `PTL_EVENT_FETCH_ATOMIC_OVERFLOW`) is registered on the *target* node to indicate completion of an atomic operation; and if data is returned from the *target* node, a `PTL_EVENT_REPLY` event is registered on the *initiator* node in the event queue and/or counting event specified by the *get_md_handle*. It is an error to use memory descriptors bound to different network interfaces in a single **PtlFetchAtomic()** call. The behavior that occurs when the *local_get_offset* into the *get_md_handle* overlaps with the *local_put_offset* into the *put_md_handle* is undefined. Operations performed by **PtlFetchAtomic()** are constrained to be no more than *max_fetch_atomic_size* bytes and must be aligned at the target to the size of **ptl_datatype_t** passed in the *datatype* argument. **PtlFetchAtomic()** is not atomic relative to other host operations, except those requested through the Portals API.

Function Prototype for PtlFetchAtomic

```
int PtlFetchAtomic(ptl_handle_md_t get_md_handle,
                   ptl_size_t local_get_offset,
                   ptl_handle_md_t put_md_handle,
                   ptl_size_t local_put_offset,
                   ptl_size_t length,
                   ptl_process_t target_id,
                   ptl_pt_index_t pt_index,
                   ptl_match_bits_t match_bits,
                   ptl_size_t remote_offset,
                   void *user_ptr,
                   ptl_hdr_data_t hdr_data,
                   ptl_op_t operation,
                   ptl_datatype_t datatype);
```

Arguments

get_md_handle	**input**	The memory descriptor handle that describes the memory into which the result of the operation will be placed. The memory descriptor can have an event queue associated with it to record events, such as when the result of the operation has been returned. Similarly, the memory descriptor can have a counting event to record these events.
local_get_offset	**input**	Offset from the start of the memory descriptor referenced by the *get_md_handle* to use for received data.
put_md_handle	**input**	The memory descriptor handle 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. If the memory descriptor has a counting event associated with it, it may optionally be used to record the same events.
local_put_offset	**input**	Offset from the start of the memory descriptor referenced by the *put_md_handle* to use for transmitted data.

length	**input**	Length of the memory region to be sent and/or received. The *length* field must be less than or equal to *max_atomic_size*.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory region (used unless the target match list entry has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	See the discussion for **PtlPut()**.
hdr_data	**input**	See the discussion for **PtlPut()**.
operation	**input**	The operation to be performed using the initiator and target data.
datatype	**input**	The type of data being operated on at the initiator and target.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.15.7 PtlSwap

The **PtlSwap()** function provides an extra argument (the *operand*) beyond the **PtlFetchAtomic()** function. **PtlSwap()** handles the `PTL_SWAP`, `PTL_CSWAP` (and variants), and `PTL_MSWAP` operations and is subject to the additional restriction that `PTL_CSWAP` (and variants) and `PTL_MSWAP` operations can only be as long as a single datatype item. Events are handled in the same way as they are for **PtlFetchAtomic()**, since **PtlSwap()** is a special case of a **PtlFetchAtomic()**. Like **PtlFetchAtomic()**, receiving a `PTL_EVENT_REPLY` inherently implies that the flow control check has passed on the target node. It is an error to use memory descriptors bound to different network interfaces in a single **PtlSwap()** call. The behavior that occurs when the *local_get_offset* into the *get_md_handle* overlaps with the *local_put_offset* into the *put_md_handle* is undefined. Operations performed by **PtlSwap()** are constrained to be no more than *max_fetch_atomic_size* bytes and must be aligned at the target to the size of **ptl_datatype_t** passed in the *datatype* argument. `PTL_CSWAP` and `PTL_MSWAP` operations are further restricted to one item, whose size is defined by the size of the datatype used. **PtlSwap()** is not atomic relative to other host operations, except those requested through the Portals API.

Function Prototype for PtlSwap

```
int PtlSwap(ptl_handle_md_t get_md_handle,
            ptl_size_t local_get_offset,
            ptl_handle_md_t put_md_handle,
            ptl_size_t local_put_offset,
            ptl_size_t length,
            ptl_process_t target_id,
            ptl_pt_index_t pt_index,
            ptl_match_bits_t match_bits,
            ptl_size_t remote_offset,
            void *user_ptr,
            ptl_hdr_data_t hdr_data,
            const void *operand,
            ptl_op_t operation,
            ptl_datatype_t datatype);
```

Arguments

get_md_handle	**input**	The memory descriptor handle that describes the memory into which the result of the operation will be placed. The memory descriptor can have an event queue associated with it to record events, such as when the result of the operation has been returned. Similarly, the memory descriptor can have a counting event to record these events.
local_get_offset	**input**	Offset from the start of the memory descriptor referenced by the *get_md_handle* to use for received data.
put_md_handle	**input**	The memory descriptor handle 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. If the memory descriptor has a counting event associated with it, it may optionally be used to record the same events.
local_put_offset	**input**	Offset from the start of the memory descriptor referenced by the *put_md_handle* to use for transmitted data.
length	**input**	Length of the memory region to be sent and/or received. The *length* field must be less than or equal to *max_atomic_size* for `PTL_SWAP` operations and can only be as large as a single datatype item for `PTL_CSWAP` and `PTL_MSWAP` operations, and variants of those.
target_id	**input**	A process identifier for the *target* process.
pt_index	**input**	The index in the *target* portal table.
match_bits	**input**	The match bits to use for message selection at the *target* process.
remote_offset	**input**	The offset into the target memory region (used unless the target match list entry has the `PTL_ME_MANAGE_LOCAL` option set).
user_ptr	**input**	See the discussion for **PtlPut()**.
hdr_data	**input**	See the discussion for **PtlPut()**.
operand	**input**	A pointer to the data to be used for the `PTL_CSWAP` (and variants) and `PTL_MSWAP` operations (ignored for other operations). The data pointed to is of the type specified by the *datatype* argument and must be included in the message.
operation	**input**	The operation to be performed using the initiator and target data.
datatype	**input**	The type of data being operated on at the initiator and target.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.15.8 PtlAtomicSync

The **PtlAtomicSync()** function synchronizes the atomic accesses through the Portals API with accesses by the host. When a data item is accessed by a Portals atomic operation, modification of the same data item by the host or by an atomic operation using a different datatype can lead to undefined behavior. When **PtlAtomicSync()** is called, it will block until it is safe for the host (or other atomic operations with a different datatype) to modify the data items touched by previous Portals atomic operations. **PtlAtomicSync()** is called at the target of atomic operations.

IMPLEMENTATION NOTE 15:

Portals Atomic Synchronization

The atomicity definition for Portals allows a network interface to offload atomic operations and to have a non-coherent cache on the network interface. With a non-coherent cache, any access to a memory location by an atomic operation makes it impossible to safely modify that location on the host. **PtlAtomicSync()** is provided to make modifications from the host safe again.

Function Prototype for PtlAtomicSync

```
int PtlAtomicSync();
```

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

3.16 Triggered Operations

For a variety of scenarios, it is desirable to setup a response to incoming messages. As an example, a tree based reduction operation could be performed by having each layer of the tree issue a **PtlAtomic()** operation to its parent after receiving a **PtlAtomic()** from all of its children. To provide this operation, triggered versions of each of the data movement operations are provided. To create a triggered operation, a *trig_ct_handle* and an integer *threshold* are added to the argument list. When the count (the sum of the success and failure fields) referenced by the *trig_ct_handle* argument reaches or exceeds the *threshold* (equal to or greater), the operation proceeds *at the initiator of the operation*. For example, a **PtlTriggeredGet()** or a **PtlTriggeredAtomic()** will not leave the *initiator* until the threshold is reached. A triggered operation does not use the state of the buffer when the application calls the Portals function. Instead, it uses the state of the buffer after the threshold condition is met. Pending triggered operations can

be canceled using **PtlCTCancelTriggered()**.

Triggered operations are processed in order of threshold values, even if the counting event is increased by a large amount at once (such as through a call to **PtlCTInc()**). If a counting event has already reached the *threshold* when a triggered operation is created, that operation is immediately processed.

Triggered operations proceed in the order their trigger threshold is reached, implying ordering within the implementation. While not required, there may be significant performance advantages to ordering calls to triggered operations by threshold.

Discussion: *The use of a trig_ct_handle and threshold enables a variety of usage models. A single match list entry can trigger one operation (or several) by using an independent trig_ct_handle on the match list entry. One operation can be triggered by a combination of previous events (include a combination of initiator and target side events) by having all of the earlier operations reference a single trig_ct_handle and using an appropriate threshold.*

IMPLEMENTATION NOTE 16:

Ordering of Triggered Operations

The semantics of triggered operations imply that (at a minimum) operations will proceed in the order that their trigger threshold is reached. A quality implementation will also release operations that reach their threshold simultaneously on the same *trig_ct_handle* in the order that they are issued. Users should also create triggered operations in ascending *threshold* values to decrease sorting work on implementations.

3.16.1 PtlTriggeredPut

The **PtlTriggeredPut()** function adds triggered operation semantics to the **PtlPut()** function described in Section 3.15.2.

Function Prototype for PtlTriggeredPut

```
int PtlTriggeredPut(ptl_handle_md_t md_handle,
                    ptl_size_t local_offset,
                    ptl_size_t length,
                    ptl_ack_req_t ack_req,
                    ptl_process_t target_id,
                    ptl_pt_index_t pt_index,
                    ptl_match_bits_t match_bits,
                    ptl_size_t remote_offset,
                    void *user_ptr,
                    ptl_hdr_data_t hdr_data,
                    ptl_handle_ct_t trig_ct_handle,
                    ptl_size_t threshold);
```

Arguments

md_handle	**input**	See **PtlPut()** description in Section 3.15.2.

local_offset	**input**	See **PtlPut()** description in Section 3.15.2.
length	**input**	See **PtlPut()** description in Section 3.15.2.
ack_req	**input**	See **PtlPut()** description in Section 3.15.2.
target_id	**input**	See **PtlPut()** description in Section 3.15.2.
pt_index	**input**	See **PtlPut()** description in Section 3.15.2.
match_bits	**input**	See **PtlPut()** description in Section 3.15.2.
remote_offset	**input**	See **PtlPut()** description in Section 3.15.2.
user_ptr	**input**	See **PtlPut()** description in Section 3.15.2.
hdr_data	**input**	See **PtlPut()** description in Section 3.15.2.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.16.2 PtlTriggeredGet

The **PtlTriggeredGet()** function adds triggered operation semantics to the **PtlGet()** function described in Section 3.15.3.

Function Prototype for PtlTriggeredGet

```
int PtlTriggeredGet(ptl_handle_md_t md_handle,
                    ptl_size_t local_offset,
                    ptl_size_t length,
                    ptl_process_t target_id,
                    ptl_pt_index_t pt_index,
                    ptl_match_bits_t match_bits,
                    void *user_ptr,
                    ptl_size_t remote_offset,
                    ptl_handle_ct_t ct_handle,
                    ptl_size_t threshold);
```

Arguments

md_handle	**input**	See **PtlGet()** description in Section 3.15.3.
target_id	**input**	See **PtlGet()** description in Section 3.15.3.
pt_index	**input**	See **PtlGet()** description in Section 3.15.3.
match_bits	**input**	See **PtlGet()** description in Section 3.15.3.

user_ptr	**input**	See **PtlGet()** description in Section 3.15.3.
remote_offset	**input**	See **PtlGet()** description in Section 3.15.3.
local_offset	**input**	See **PtlGet()** description in Section 3.15.3.
length	**input**	See **PtlGet()** description in Section 3.15.3.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.16.3 PtlTriggeredAtomic

The **PtlTriggeredAtomic()** function adds triggered operation semantics to the **PtlAtomic()** function described in Section 3.15.5. When combined with triggered counting increments (**PtlTriggeredCTInc()**) and sets (**PtlTriggeredCTSet()**), triggered atomic operations enable an offloaded, non-blocking implementation of most collective operations.

Function Prototype for PtlTriggeredAtomic

```
int PtlTriggeredAtomic(ptl_handle_md_t md_handle,
                       ptl_size_t local_offset,
                       ptl_size_t length,
                       ptl_ack_req_t ack_req,
                       ptl_process_t target_id,
                       ptl_pt_index_t pt_index,
                       ptl_match_bits_t match_bits,
                       ptl_size_t remote_offset,
                       void *user_ptr,
                       ptl_hdr_data_t hdr_data,
                       ptl_op_t operation,
                       ptl_datatype_t datatype,
                       ptl_handle_ct_t trig_ct_handle,
                       ptl_size_t threshold);
```

Arguments

md_handle	**input**	See **PtlAtomic()** description in Section 3.15.5.
local_offset	**input**	See **PtlAtomic()** description in Section 3.15.5.
length	**input**	See **PtlAtomic()** description in Section 3.15.5.
ack_req	**input**	See **PtlAtomic()** description in Section 3.15.5.

target_id	**input**	See **PtlAtomic()** description in Section 3.15.5.
pt_index	**input**	See **PtlAtomic()** description in Section 3.15.5.
match_bits	**input**	See **PtlAtomic()** description in Section 3.15.5.
remote_offset	**input**	See **PtlAtomic()** description in Section 3.15.5.
user_ptr	**input**	See **PtlAtomic()** description in Section 3.15.5.
hdr_data	**input**	See **PtlAtomic()** description in Section 3.15.5.
operation	**input**	See **PtlAtomic()** description in Section 3.15.5.
datatype	**input**	See **PtlAtomic()** description in Section 3.15.5.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.16.4 PtlTriggeredFetchAtomic

The **PtlTriggeredFetchAtomic()** function adds triggered operation semantics to the **PtlFetchAtomic()** function described in Section 3.15.6.

Function Prototype for PtlTriggeredFetchAtomic

```
int PtlTriggeredFetchAtomic(ptl_handle_md_t get_md_handle,
                            ptl_size_t local_get_offset,
                            ptl_handle_md_t put_md_handle,
                            ptl_size_t local_put_offset,
                            ptl_size_t length,
                            ptl_process_t target_id,
                            ptl_pt_index_t pt_index,
                            ptl_match_bits_t match_bits,
                            ptl_size_t remote_offset,
                            void *user_ptr,
                            ptl_hdr_data_t hdr_data,
                            ptl_op_t operation,
                            ptl_datatype_t datatype,
                            ptl_handle_ct_t trig_ct_handle,
                            ptl_size_t threshold);
```

Arguments

get_md_handle	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.

local_get_offset	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
put_md_handle	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
local_put_offset	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
length	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
target_id	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
pt_index	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
match_bits	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
remote_offset	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
user_ptr	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
hdr_data	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
operation	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
datatype	**input**	See **PtlFetchAtomic()** description in Section 3.15.6.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.16.5 PtlTriggeredSwap

The **PtlTriggeredSwap()** function adds triggered operation semantics to the **PtlSwap()** function described in Section 3.15.7.

Function Prototype for PtlTriggeredSwap

```
int PtlTriggeredSwap(ptl_handle_md_t get_md_handle,
                     ptl_size_t local_get_offset,
                     ptl_handle_md_t put_md_handle,
                     ptl_size_t local_put_offset,
                     ptl_size_t length,
                     ptl_process_t target_id,
                     ptl_pt_index_t pt_index,
                     ptl_match_bits_t match_bits,
                     ptl_size_t remote_offset,
                     void *user_ptr,
                     ptl_hdr_data_t hdr_data,
                     const void *operand,
                     ptl_op_t operation,
                     ptl_datatype_t datatype,
                     ptl_handle_ct_t trig_ct_handle,
                     ptl_size_t threshold);
```

Arguments

get_md_handle	**input**	See **PtlSwap()** description in Section 3.15.7.
local_get_offset	**input**	See **PtlSwap()** description in Section 3.15.7.
put_md_handle	**input**	See **PtlSwap()** description in Section 3.15.7.
local_put_offset	**input**	See **PtlSwap()** description in Section 3.15.7.
length	**input**	See **PtlSwap()** description in Section 3.15.7.
target_id	**input**	See **PtlSwap()** description in Section 3.15.7.
pt_index	**input**	See **PtlSwap()** description in Section 3.15.7.
match_bits	**input**	See **PtlSwap()** description in Section 3.15.7.
remote_offset	**input**	See **PtlSwap()** description in Section 3.15.7.
user_ptr	**input**	See **PtlSwap()** description in Section 3.15.7.
hdr_data	**input**	See **PtlSwap()** description in Section 3.15.7.
operand	**input**	See **PtlSwap()** description in Section 3.15.7.
operation	**input**	See **PtlSwap()** description in Section 3.15.7.
datatype	**input**	See **PtlSwap()** description in Section 3.15.7.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.

PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.16.6 PtlTriggeredCTInc

The triggered counting event increment extends the counting event increment (**PtlCTInc()**) with the triggered operation semantics. It is a convenient mechanism to provide chaining of dependencies between counting events. This allows a relatively arbitrary ordering of operations. For example, a **PtlTriggeredPut()** and a **PtlTriggeredCTInc()** could be dependent on *ct_handle* A with the same threshold. If the **PtlTriggeredCTInc()** is set to increment *ct_handle* B and a second **PtlTriggeredPut()** is dependent on *ct_handle* B, the second **PtlTriggeredPut()** will occur after the first.

Function Prototype for PtlTriggeredCTInc

```
int PtlTriggeredCTInc(ptl_handle_ct_t ct_handle,
                      ptl_ct_event_t increment,
                      ptl_handle_ct_t trig_ct_handle,
                      ptl_size_t threshold);
```

Arguments

ct_handle	**input**	See **PtlCTInc()** description in Section 3.14.9.
increment	**input**	See **PtlCTInc()** description in Section 3.14.9.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.16.7 PtlTriggeredCTSet

The triggered counting event increment extends the counting event set (**PtlCTSet()**) with the triggered operation semantics. It is a convenient mechanism to provide reinitialization of counting events between invocations of an algorithm.

Function Prototype for PtlTriggeredCTSet

```
int PtlTriggeredCTSet(ptl_handle_ct_t ct_handle,
                      ptl_ct_event_t new_ct,
                      ptl_handle_ct_t trig_ct_handle,
                      ptl_size_t threshold);
```

Arguments

ct_handle	**input**	See **PtlCTSet()** description in Section 3.14.8.
new_ct	**input**	See **PtlCTSet()** description in Section 3.14.8.
trig_ct_handle	**input**	Handle used for triggering the operation.
threshold	**input**	Threshold at which the operation triggers.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.17 Deferred Communication Operations

Frequently, upper layer protocols and applications generate a stream of operations with loose synchronization requirements between operations. For example, an MPI implementation may need to start a large number of operations to implement the fan-out portion of a collective operation. The portals deferred communication operations provide a mechanism for allowing the Portals implementation to optimize for these situations.

3.17.1 PtlStartBundle

The **PtlStartBundle()** function is used by the application to indicate to the implementation that a group of communication operations is about to start. **PtlStartBundle()** takes an *ni_handle* as an argument and only impacts operations on that *ni_handle*. **PtlStartBundle()** can be called multiple times, and each call to **PtlStartBundle()** increments a reference count and must be matched by a call to **PtlEndBundle()**. After a call to **PtlStartBundle()**, the implementation may begin deferring communication operations until a call to **PtlEndBundle()**.

Function Prototype for PtlStartBundle

```
int PtlStartBundle(ptl_handle_ni_t ni_handle);
```

Arguments

ni_handle	**input**	An interface handle to start bundling operations.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

Discussion: *Layered libraries and heavily nested **PtlStartBundle()** calls can yield unexpected results. The **PtlStartBundle()** and **PtlEndBundle()** interface was designed for use in short periods of high activity (e.g. during the setup of a collective operation or during an inner loop for PGAS languages). The interval between **PtlStartBundle()** and the corresponding **PtlEndBundle()** should be kept short.*

IMPLEMENTATION NOTE 17:	Purpose of Bundling
	The **PtlStartBundle()** and **PtlEndBundle()** interface was designed to allow the implementation to avoid unnecessary sfence()/memory barrier operations during periods that the application expects high message rate. A quality implementation will attempt to minimize latency while maximizing message rate. For example, an implementation that requires writes into "write-combining" space may require sfence() operations with every message to have relatively deterministic latency. Between a **PtlStartBundle()** and **PtlEndBundle()**, the implementation might simply omit the sfence() operations.

3.17.2 PtlEndBundle

The **PtlEndBundle()** function is used by the application to indicate to the implementation that a group of communication operations has ended. **PtlEndBundle()** takes an *ni_handle* as an argument and only impacts operations on that *ni_handle*. **PtlEndBundle()** must be called once for each **PtlStartBundle()** call. At each call to **PtlEndBundle()**, the implementation must initiate all communication operations that have been deferred; however, the implementation is not required to cease bundling future operations until the reference count reaches zero.

Function Prototype for PtlEndBundle

```
int PtlEndBundle(ptl_handle_ni_t ni_handle);
```

Arguments

ni_handle	**input**	An interface handle to end bundling operations.

Return Codes

PTL_OK	Indicates success.
PTL_NO_INIT	Indicates that the Portals API has not been successfully initialized.
PTL_ARG_INVALID	Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

3.18 Operations on Handles

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

3.18.1 PtlHandleIsEqual

The **PtlHandleIsEqual()** function compares two handles to determine if they represent the same object. **PtlHandleIsEqual()** does not check whether the two handles are valid, but only whether they are equal.

Function Prototype for PtlHandleIsEqual

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

Arguments

handle1	**input**	An object handle. May be the constant value `PTL_INVALID_HANDLE`, which represents the value of an invalid handle.
handle2	**input**	An object handle. May be the constant value `PTL_INVALID_HANDLE`, which represents the value of an invalid handle.

Return Codes

zero	Indicates that the two handles are not equivalent.
non-zero	Indicates that the two handles are equivalent.

Discussion: ***PtlHandleIsEqual()*** *returns a value suitable for direct evaluation in a conditional expression. While different from all other Portals functions and previous Portals versions, it does greatly simplify usage of* ***PtlHandleIsEqual()****.*

3.19 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 3.5 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 and structures with members of this type.

Table 3.5. Portals Data Types: Data Types Defined by the Portals API.

Name	Meaning	Definition	Functions/Data Structures
ptl_ack_req_t	acknowledgment request types	3.15.1	PtlAtomic(), PtlPut(), PtlTriggeredAtomic(), PtlTriggeredPut()
ptl_ct_event_t	counting event structure	3.14.1	PtlCTGet(), PtlCTInc(), PtlCTPoll(), PtlCTSet(), PtlTriggeredCTInc(), PtlTriggeredCTSet(), PtlCTWait()
ptl_datatype_t	datatype for atomic operation	3.15.4	PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap(), ptl_event_t
ptl_event_kind_t	event kind	3.13.1	ptl_event_t
ptl_event_t	event queue entry	3.13.4	PtlEQGet(), PtlEQWait(), PtlEQPoll()
ptl_handle_any_t	any object handles	3.3.2	PtlHandleIsEqual(), PtlNIHandle()
ptl_handle_ct_t	counting event handles	3.3.2	PtlCTAlloc(), PtlCTCancelTriggered(), PtlCTFree(), PtlCTGet(), PtlCTInc(), PtlCTPoll(), PtlCTSet(), PtlCTWait(), PtlTriggeredAtomic(), PtlTriggeredCTInc(), PtlTriggeredCTSet(), PtlTriggeredFetchAtomic(), PtlTriggeredGet(), PtlTriggeredPut(), PtlTriggeredSwap(), ptl_le_t, ptl_md_t, ptl_me_t
ptl_handle_eq_t	event queue handles	3.3.2	PtlEQAlloc(), PtlEQFree(), PtlEQGet(), PtlEQPoll(), PtlEQWait(), PtlPTAlloc(), ptl_md_t
ptl_handle_le_t	list entry handles	3.3.2	PtlLEAppend(), PtlLEUnlink()
ptl_handle_md_t	memory descriptor handles	3.3.2	PtlAtomic(), PtlFetchAtomic(), PtlGet(), PtlMDBind(), PtlMDRelease(), PtlPut(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredGet(), PtlTriggeredPut(), PtlTriggeredSwap()
ptl_handle_me_t	match list entry handles	3.3.2	PtlMEAppend(), PtlMEUnlink()

continued on next page

continued from previous page

Name	Meaning	Definition	Functions/Data Structures
ptl_handle_ni_t	network interface handles	3.3.2	PtlCTAlloc(), PtlEQAlloc(), PtlEndBundle(), PtlGetId(), PtlGetMap(), PtlGetPhysId(), PtlGetUid(), PtlLEAppend(), PtlLESearch(), PtlMDBind(), PtlMEAppend(), PtlMESearch(), PtlNIFini(), PtlNIHandle(), PtlNIInit(), PtlNIStatus(), PtlPTAlloc(), PtlPTDisable(), PtlPTEnable(), PtlPTFree(), PtlSetMap(), PtlStartBundle()
ptl_hdr_data_t	user header data	3.15.2	PtlAtomic(), PtlFetchAtomic(), PtlPut(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredPut(), PtlTriggeredSwap(), ptl_event_t
ptl_interface_t	network interface identifiers	3.3.5	PtlNIInit()
ptl_iovec_t	scatter/gather buffer descriptors	3.10.2	
ptl_le_t	list entries	3.11.1	PtlLEAppend(), PtlLESearch()
ptl_list_t	type of list attached to a portal table entry	3.12.2	PtlLEAppend(), PtlLEAppend(), ptl_event_t
ptl_match_bits_t	match (and ignore) bits	3.3.4	PtlAtomic(), PtlFetchAtomic(), PtlGet(), PtlPut(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredGet(), PtlTriggeredPut(), PtlTriggeredSwap(), ptl_event_t, ptl_me_t
ptl_md_t	memory descriptors	3.10.1	PtlMDBind()
ptl_me_t	match list entries	3.12.1	PtlMEAppend(), PtlMESearch()
ptl_ni_fail_t	network interface specific failures	3.13.3	ptl_event_t
ptl_ni_limits_t	implementation dependent limits	3.6.1	PtlNIInit()
ptl_nid_t	node identifiers	3.3.6	ptl_process_t
ptl_op_t	atomic operation type	3.15.4	PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap(), ptl_event_t
ptl_pid_t	process identifier	3.3.6	PtlNIInit(), ptl_process_t
ptl_process_t	process identifiers	3.9.1	PtlAtomic(), PtlFetchAtomic(), PtlGet(), PtlGetId(), PtlGetMap(), PtlGetPhysId(), PtlPut(), PtlSetMap(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredGet(), PtlTriggeredPut(), PtlTriggeredSwap(), ptl_event_t, ptl_me_t

continued on next page

continued from previous page

Name	Meaning	Definition	Functions/Data Structures
ptl_pt_index_t	portal table indexes	3.3.3	PtlAtomic(), PtlFetchAtomic(), PtlGet(), PtlLEAppend(), PtlLESearch(), PtlMEAppend(), PtlMESearch(), PtlPTAlloc(), PtlPTDisable(), PtlPTEnable(), PtlPTFree(), PtlPut(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredGet(), PtlTriggeredPut(), PtlTriggeredSwap(), ptl_event_t
ptl_rank_t	rank within a group of communicating processes	3.3.6	ptl_process_t
ptl_search_op_t	operation performed by list search	3.12.4	PtlLESearch(), PtlMESearch()
ptl_size_t	sizes	3.3.1	PtlAtomic(), PtlCTPoll(), PtlCTWait(), PtlEQAlloc(), PtlFetchAtomic(), PtlGet(), PtlGetMap(), PtlPut(), PtlSetMap(), PtlSwap(), PtlTriggeredAtomic(), PtlTriggeredCTInc(), PtlTriggeredCTSet(), PtlTriggeredFetchAtomic(), PtlTriggeredGet(), PtlTriggeredPut(), PtlTriggeredSwap(), ptl_ct_event_t, ptl_event_t, ptl_iovec_t, ptl_le_t, ptl_md_t, ptl_me_t, ptl_ni_limits_t
ptl_sr_index_t	status register indexes	3.3.7	PtlNIStatus()
ptl_sr_value_t	status register values	3.3.7	PtlNIStatus()
ptl_time_t	time in milliseconds	3.13.9	PtlCTPoll(), PtlEQPoll()
ptl_uid_t	user identifier	3.3.6	PtlGetUid(), ptl_event_t, ptl_le_t, ptl_me_t

Table 3.6 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 3.6. Portals Functions: Functions Defined by the Portals API.

Name	Meaning	Definition
PtlAtomic()	perform an atomic operation	3.15.5
PtlAtomicSync()	synchronize results of atomic operations with the host	3.15.8
PtlCTAlloc()	create a counting event	3.14.2
PtlCTCancelTriggered()	cancel pending triggered operations	3.14.4
PtlCTFree()	free a counting event	3.14.3
PtlCTGet()	get the current value of a counting event	3.14.5
PtlCTInc()	increment a counting event by a certain value	3.14.9
PtlCTPoll()	wait for an array of counting events to reach certain values	3.14.7
PtlCTSet()	set a counting event to a certain value	3.14.8
PtlCTWait()	wait for a counting event to reach a certain value	3.14.6
PtlEndBundle()	end a communications bundle	3.17.2
PtlEQAlloc()	create an event queue	3.13.5
PtlEQFree()	release the resources for an event queue	3.13.6

continued on next page

continued from previous page

Name	Meaning	Definition
PtlEQGet()	get the next event from an event queue	3.13.7
PtlEQPoll()	poll for a new event on multiple event queues	3.13.9
PtlEQWait()	wait for a new event in an event queue	3.13.8
PtlFetchAtomic()	perform an fetch and atomic operation	3.15.6
PtlFini()	shut down the Portals API	3.5.2
PtlGet()	perform a *get* operation	3.15.3
PtlGetId()	get the identifier for the current process	3.9.2
PtlGetMap()	retrieve a rank to physical mapping	3.6.7
PtlGetPhysId()	get the physical identifier for the current process	3.9.3
PtlGetUid()	get the network interface specific user identifier	3.8.1
PtlHandleIsEqual()	compares two handles to determine if they represent the same object	3.18.1
PtlInit()	initialize the Portals API	3.5.1
PtlLEAppend()	create a list entry and append it to a portal table	3.11.2
PtlLESearch()	search an unexpected header	3.11.4
PtlLEUnlink()	remove a list entry from a list and release its resources	3.11.3
PtlMDBind()	create a free-floating memory descriptor	3.10.3
PtlMDRelease()	release resources associated with a memory descriptor	3.10.4
PtlMEAppend()	create a match list entry and append it to a portal table	3.12.2
PtlMESearch()	search an unexpected header	3.12.4
PtlMEUnlink()	remove a match list entry from a list and release its resources	3.12.3
PtlNIFini()	shut down a network interface	3.6.3
PtlNIHandle()	get the network interface handle for an object	3.6.5
PtlNIInit()	initialize a network interface	3.6.2
PtlNIStatus()	read a network interface status register	3.6.4
PtlPTAlloc()	allocate a free portal table entry	3.7.1
PtlPTFree()	free a portal table entry	3.7.2
PtlPTDisable()	disable a portal table entry	3.7.3
PtlPTEnable()	enable a portal table entry that has been disabled	3.7.4
PtlPut()	perform a *put* operation	3.15.2
PtlSetMap()	initialize a rank to physical mapping	3.6.6
PtlStartBundle()	start a communications bundle	3.17.1
PtlSwap()	perform a swap operation	3.15.7
PtlTriggeredAtomic()	perform a triggered atomic operation	3.16.3
PtlTriggeredCTInc()	a triggered increment of a counting event by a certain value	3.16.6
PtlTriggeredCTSet()	a triggered set of a counting event by a certain value	3.16.7
PtlTriggeredFetchAtomic()	perform a triggered fetch and atomic operation	3.16.4
PtlTriggeredGet()	perform a triggered *get* operation	3.16.2
PtlTriggeredPut()	perform a triggered *put* operation	3.16.1
PtlTriggeredSwap()	perform a triggered swap operation	3.16.5

Table 3.7 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 3.7. Portals Return Codes: Function Return Codes for the Portals API.

Name	Meaning	Functions
PTL_ARG_INVALID	invalid argument passed	*all*, except **PtlAtomicSync()**, **PtlFini()**, **PtlHandleIsEqual()**, **PtlInit()**
PTL_CT_NONE_REACHED	timeout reached before any counting event reached the test	**PtlCTPoll()**

continued on next page

continued from previous page

Name	Meaning	Functions
PTL_EQ_DROPPED	at least one event has been dropped	**PtlEQGet()**, **PtlEQPoll()**, **PtlEQWait()**
PTL_EQ_EMPTY	no events available in an event queue	**PtlEQGet()**, **PtlEQPoll()**
PTL_FAIL	error during initialization	**PtlInit()**
PTL_IGNORED	Logical map set failed	**PtlSetMap()**
PTL_IN_USE	MD, ME, or LE has pending operations	**PtlLEUnlink()**, **PtlMEUnlink()**
PTL_INTERRUPTED	wait/get operation was interrupted	**PtlCTPoll()**, **PtlCTWait()**, **PtlEQPoll()**, **PtlEQWait()**
PTL_LIST_TOO_LONG	list too long	**PtlLEAppend()**, **PtlMEAppend()**
PTL_NO_INIT	uninitialized API	*all*, except **PtlFini()**, **PtlHandleIsEqual()**, **PtlInit()**
PTL_NO_SPACE	insufficient memory	**PtlCTAlloc()**, **PtlEQAlloc()**, **PtlGetMap()**, **PtlLEAppend()**, **PtlMDBind()**, **PtlMEAppend()**, **PtlNIInit()**, **PtlSetMap()**
PTL_OK	success	*all*, except **PtlFini()**, **PtlHandleIsEqual()**
PTL_PID_IN_USE	pid is in use	**PtlNIInit()**
PTL_PT_EQ_NEEDED	EQ must be attached when flow control is enabled	**PtlPTAlloc()**
PTL_PT_FULL	portal table is full	**PtlPTAlloc()**
PTL_PT_IN_USE	portal table index is busy	**PtlPTAlloc()**, **PtlPTFree()**

Table 3.8 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 section in which the constant is introduced or described.

Table 3.8. Portals Constants: Other Constants Defined by the Portals API.

Name	Meaning	Base Type	Definition
PTL_ACK_REQ	request an acknowledgment	**ptl_ack_req_t**	3.15
PTL_BAND	Compute and return the bitwise AND of the initiator and target value	**ptl_op_t**	3.15.4
PTL_BOR	Compute and return the bitwise OR of the initiator and target value	**ptl_op_t**	3.15.4
PTL_BXOR	Compute and return the bitwise XOR of the initiator and target value	**ptl_op_t**	3.15.4
PTL_COHERENT_ATOMICS	a flag to indicate that the implementation provides atomic operations which are coherent with processor atomic operations	int	3.15.4
PTL_CSWAP	Conditional swap if target and operand equal	**ptl_op_t**	3.15.4
PTL_CSWAP_GE	Conditional swap if the operand is greater than or equal to the target	**ptl_op_t**	3.15.4

continued on next page

continued from previous page

Name	Meaning	Base Type	Definition
PTL_CSWAP_GT	Conditional swap if the operand is greater than the target	**ptl_op_t**	3.15.4
PTL_CSWAP_LE	Conditional swap if the operand is less than or equal to the target	**ptl_op_t**	3.15.4
PTL_CSWAP_LT	Conditional swap if the operand is less than the target	**ptl_op_t**	3.15.4
PTL_CSWAP_NE	Conditional swap if the operand and target are not equal	**ptl_op_t**	3.15.4
PTL_CT_ACK_REQ	request a counting acknowledgment	**ptl_ack_req_t**	3.15
PTL_CT_NONE	a NULL count handle	**ptl_handle_ct_t**	3.3.2
PTL_DOUBLE	64-bit floating-point number	**ptl_op_t**	3.15.4
PTL_DOUBLE_COMPLEX	64-bit floating-point complex number	**ptl_op_t**	3.15.4
PTL_EQ_NONE	a NULL event queue handle	**ptl_handle_eq_t**	3.3.2
PTL_EVENT_ACK	acknowledgment event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_ATOMIC	atomic event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_ATOMIC_OVERFLOW	atomic overflow event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_AUTO_FREE	automatic free event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_AUTO_UNLINK	automatic unlink event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_FETCH_ATOMIC	fetching atomic event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_FETCH_ATOMIC_OVERFLOW	fetching atomic overflow event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_GET	get event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_GET_OVERFLOW	get overflow event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_LINK	event generated when a list entry links **ptl_event_kind_t**	3.13.1	
PTL_EVENT_PT_DISABLED	portal table entry disabled event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_PUT	put event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_PUT_OVERFLOW	put overflow event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_REPLY	reply event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_SEARCH	search event	**ptl_event_kind_t**	3.13.1
PTL_EVENT_SEND	send event	**ptl_event_kind_t**	3.13.1
PTL_FLOAT	32-bit floating-point number	**ptl_op_t**	3.15.4
PTL_FLOAT_COMPLEX	32-bit floating-point complex number	**ptl_op_t**	3.15.4
PTL_IFACE_DEFAULT	default interface	**ptl_interface_t**	3.3.5
PTL_INT16_T	16-bit signed integer	**ptl_op_t**	3.15.4
PTL_INT32_T	32-bit signed integer	**ptl_op_t**	3.15.4
PTL_INT64_T	64-bit signed integer	**ptl_op_t**	3.15.4
PTL_INT8_T	8-bit signed integer	**ptl_op_t**	3.15.4
PTL_INVALID_HANDLE	invalid handle	**ptl_handle_any_t**	3.3.2

continued on next page

continued from previous page

Name	Meaning	Base Type	Definition
PTL_IOVEC	a flag to enable scatter/gather memory descriptors	int	3.12.1
PTL_LAND	Compute and return the logical AND of the initiator and target	**ptl_op_t**	3.15.4
PTL_LE_ACK_DISABLE	a flag to disable acknowledgments	int	3.11.1
PTL_LE_EVENT_COMM_DISABLE	a flag to disable events associated with new communications	int	3.11.1
PTL_LE_EVENT_CT_BYTES	a flag to count bytes instead of operations	int	3.11.1
PTL_LE_EVENT_CT_COMM	a flag to count communication events	int	3.11.1
PTL_LE_EVENT_CT_OVERFLOW	a flag to count overflow events	int	3.11.1
PTL_LE_EVENT_FLOWCTRL_DISABLE	a flag to disable events associated with flow control	int	3.11.1
PTL_LE_EVENT_LINK_DISABLE	a flag to disable link events	int	3.11.1
PTL_LE_EVENT_OVER_DISABLE	a flag to disable overflow events	int	3.11.1
PTL_LE_EVENT_SUCCESS_DISABLE	a flag to disable events that indicate success	int	3.11.1
PTL_LE_EVENT_UNLINK_DISABLE	a flag to disable unlink events	int	3.11.1
PTL_LE_IS_ACCESSIBLE	a flag to indicate the entire LE is accessible	int	3.11.1
PTL_LE_OP_GET	a flag to enable *get* operations	int	3.11.1
PTL_LE_OP_PUT	a flag to enable *put* operations	int	3.11.1
PTL_LE_UNEXPECTED_HDR_DISABLE	a flag to disable adding headers to the unexpected headers list	int	3.11.1
PTL_LE_USE_ONCE	a flag to indicate that the list entry will only be used once	int	3.11.1
PTL_LONG_DOUBLE	System defined long double type	**ptl_op_t**	3.15.4
PTL_LONG_DOUBLE_COMPLEX	System defined long double complex type	**ptl_op_t**	3.15.4
PTL_LOR	Compute and return the logical OR of the initiator and target	**ptl_op_t**	3.15.4
PTL_LXOR	Compute and return the logical XOR of the initiator and target	**ptl_op_t**	3.15.4

continued on next page

continued from previous page

Name	Meaning	Base Type	Definition
PTL_MAX	Compute and return the maximum of the initiator and target	ptl_op_t	3.15.4
PTL_MD_EVENT_CT_ACK	a flag to count acknowledgment events	int	3.10.1
PTL_MD_EVENT_CT_BYTES	a flag to count bytes instead of operations	int	3.10.1
PTL_MD_EVENT_CT_REPLY	a flag to count reply events	int	3.10.1
PTL_MD_EVENT_CT_SEND	a flag to count send events	int	3.10.1
PTL_MD_EVENT_SEND_DISABLE	a flag to disable send events	int	3.10.1
PTL_MD_EVENT_SUCCESS_DISABLE	a flag to disable events that indicate success	int	3.10.1
PTL_MD_UNORDERED	a flag to indicate that messages from this MD do not need to be ordered	int	3.10.1
PTL_MD_VOLATILE	a flag to indicate that the application will modify the put buffer immediately upon operation return, before receiving a send event.	int	3.10.1
PTL_ME_ACK_DISABLE	a flag to disable acknowledgments	int	3.12.1
PTL_ME_EVENT_COMM_DISABLE	a flag to disable events associated with new communications	int	3.12.1
PTL_ME_EVENT_CT_BYTES	a flag to count bytes instead of operations	int	3.12.1
PTL_ME_EVENT_CT_COMM	a flag to count communication events	int	3.12.1
PTL_ME_EVENT_CT_OVERFLOW	a flag to count overflow events	int	3.12.1
PTL_ME_EVENT_FLOWCTRL_DISABLE	a flag to disable events associated with flow control	int	3.12.1
PTL_ME_EVENT_LINK_DISABLE	a flag to disable link events	int	3.12.1
PTL_ME_EVENT_OVER_DISABLE	a flag to disable overflow events	int	3.12.1
PTL_ME_EVENT_SUCCESS_DISABLE	a flag to disable events that indicate success	int	3.12.1
PTL_ME_EVENT_UNLINK_DISABLE	a flag to disable unlink events	int	3.12.1
PTL_ME_IS_ACCESSIBLE	a flag to indicate the entire ME is accessible	int	3.12.1
PTL_ME_MANAGE_LOCAL	a flag to enable the use of local offsets	int	3.12.1
PTL_ME_MAY_ALIGN	a flag to indicate that the implementation may align an incoming message to a natural boundary to enhance performance	int	3.12.1

continued on next page

continued from previous page

Name	Meaning	Base Type	Definition
PTL_ME_NO_TRUNCATE	a flag to disable truncation of a request	int	3.12.1
PTL_ME_OP_GET	a flag to enable *get* operations	int	3.12.1
PTL_ME_OP_PUT	a flag to enable *put* operations	int	3.12.1
PTL_ME_UNEXPECTED_HDR_DISABLE	a flag to disable adding headers to the unexpected headers list	int	3.12.1
PTL_ME_USE_ONCE	a flag to indicate that the match list entry will only be used once	int	3.12.1
PTL_MIN	Compute and return the minimum of the initiator and target	**ptl_op_t**	3.15.4
PTL_MSWAP	A masked version of the swap operation	**ptl_op_t**	3.15.4
PTL_NI_DROPPED	message was dropped	**ptl_ni_fail_t**	3.13.3
PTL_NI_LOGICAL	a flag to indicate that the network interface must provide logical addresses for network endpoints	int	3.6.2
PTL_NI_MATCHING	a flag to indicate that the network interface must provide matching portals addressing	int	3.6.2
PTL_NI_NO_MATCH	search did not find an entry in the unexpected list	**ptl_ni_fail_t**	3.13
PTL_NI_NO_MATCHING	a flag to indicate that the network interface must provide non-matching portals addressing	int	3.6.2
PTL_NI_OK	successful event	**ptl_ni_fail_t**	3.13.3
PTL_NI_OP_VIOLATION	message encountered an operation violation	**ptl_ni_fail_t**	3.13.3
PTL_NI_PERM_VIOLATION	message encountered a permissions violation	**ptl_ni_fail_t**	3.13.3
PTL_NI_PHYSICAL	a flag to indicate that the network interface must provide physical addresses for network endpoints	int	3.6.2
PTL_NI_PT_DISABLED	message encountered a disabled portal table entry	**ptl_ni_fail_t**	3.13.3
PTL_NI_SEGV	message attempted to access inaccessible memory	**ptl_ni_fail_t**	3.13.3
PTL_NI_UNDELIVERABLE	message could not be delivered	**ptl_ni_fail_t**	3.13.3
PTL_NID_ANY	wildcard for node identifier fields	**ptl_nid_t**	3.3.6
PTL_NO_ACK_REQ	request no acknowledgment	**ptl_ack_req_t**	3.15

continued on next page

continued from previous page

Name	Meaning	Base Type	Definition
PTL_OC_ACK_REQ	request an operation completed acknowledgment	**ptl_ack_req_t**	3.15
PTL_OVERFLOW_LIST	specifies the overflow list attached to a portal table entry	int	3.12.2
PTL_PID_ANY	wildcard for process identifier fields	**ptl_pid_t**	3.3.6
PTL_PID_MAX	Maximum legal process identifier	**ptl_pid_t**	3.6.2
PTL_PRIORITY_LIST	specifies the priority list attached to a portal table entry	int	3.12.2
PTL_PROD	Compute and return the product of the initiator and target value	**ptl_op_t**	3.15.4
PTL_PT_ANY	wildcard for portal table entry identifier fields	**ptl_pt_index_t**	3.7.1
PTL_PT_FLOWCTRL	a flag to request flow control	int	3.7.1
PTL_PT_ONLY_TRUNCATE	a flag to indicate that the priority list on this portal table entry will only have entries without the PTL_ME_NO_TRUNCATE option set	int	3.7.1
PTL_PT_ONLY_USE_ONCE	a flag to indicate that the priority list on this portal table entry will only have entries with the PTL_ME_USE_ONCE or PTL_LE_USE_ONCE option set	int	3.7.1
PTL_RANK_ANY	wildcard for rank fields	**ptl_rank_t**	3.3.6
PTL_SEARCH_DELETE	specifies that the unexpected list should be searched and the matching item should be deleted	int	3.12.4
PTL_SEARCH_ONLY	specifies that the unexpected list should only be searched	int	3.12.4
PTL_SIZE_MAX	maximum value of a **ptl_size_t**	**ptl_size_t**	3.3.1
PTL_SR_DROP_COUNT	index for the dropped count register	**ptl_sr_index_t**	3.3.7
PTL_SR_OPERATION_VIOLATIONS	index for the operation violations register	**ptl_sr_index_t**	3.3.7
PTL_SR_PERMISSION_VIOLATIONS	index for the permission violations register	**ptl_sr_index_t**	3.3.7

continued on next page

continued from previous page

Name	Meaning	Base Type	Definition
PTL_SUM	Compute and return the sum of the initiator and target	**ptl_op_t**	3.15.4
PTL_SWAP	Swap the initiator and target value	**ptl_op_t**	3.15.4
PTL_TARGET_BIND_INACCESSIBLE	A flag to indicate that the implementation should allow LEs to be bound over ranges of memory that are not allocated	int	3.6.1
PTL_TIME_FOREVER	a flag to indicate unbounded time	**ptl_time_t**	3.13.9
PTL_TOTAL_DATA_ORDERING	A flag to indicate that the implementation should attempt to provide total data ordering	int	3.6.1
PTL_UID_ANY	wildcard for user identifier	**ptl_uid_t**	3.3.6
PTL_UINT16_T	16-bit unsigned integer	**ptl_op_t**	3.15.4
PTL_UINT32_T	32-bit unsigned integer	**ptl_op_t**	3.15.4
PTL_UINT64_T	64-bit unsigned integer	**ptl_op_t**	3.15.4
PTL_UINT8_T	8-bit unsigned integer	**ptl_op_t**	3.15.4

Chapter 4

Guide to Implementors

In this chapter, we provide a number of notes and clarifications useful to implementors of the Portals specification. This chapter is not normative; that is, this chapter only seeks to clarify and raise subtle points in the standard. Should any statement in this chapter conflict with statements in another chapter, the other chapter is correct.

4.1 Run-time Support

The Portals API does not include a run-time interface; this is assumed to be provided by other sources, such as the machine system software or as part of an upper-layer protocol. This is similar to Open Fabrics, Myrinet/MX, and TCP/IP, which provide communication semantics, but say little about process lifespan or interaction. Interaction with a run-time is clearly unavoidable due to logically addressed network interfaces, but the proper interaction between the run-time and **PtlSetMap()**/**PtlGetMap()** is the responsibility of the upper layer protocol.

Many implementations of the Portals specification (both Portals 4.0 and earlier specifications) were tightly coupled with a specific run-time. It is expected that such coupling will continue on tightly integrated platforms in which Portals is the lowest layer communication interface. While the user of the portals library must always call **PtlSetMap()** before using a logically addressed interface, the implementation is free to ignore the requested mapping and provide it's own by returning **PTL_IGNORED**.

4.2 Data Transfer

The Portals API uses five types of messages: *put*, *acknowledgment*, *get*, *reply*, and *atomic*. 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. The Portals specification does not enforce a given wire protocol or in what order and what manner information is passed along the communication path.

4.2.1 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.

It may not be necessary for the implementation to transmit all fields listed in Table 4.1. For example, portals semantics require that an *acknowledgment* event contains the *user_ptr* and it must be placed in the event queue referenced by the *eq_handle* found in the MD referenced by the *md_handle* associated with the *put*; i.e., the *acknowledgment* event provides a pointer that the application can use to identify the operation and must be placed the in the right memory descriptor's event queue. One approach would be to send the *user_ptr* and *md_handle* to the

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

Information	Type	PtlPut() Argument	Notes
operation	int		indicates a *put* request
ack type	**ptl_ack_req_t**	*ack_req*	
options	unsigned int	*md_handle*	*options* field from NI associated with MD
initiator	**ptl_process_t**		local information
user	**ptl_uid_t**		local information
target	**ptl_process_t**	*target_id*	
portal index	**ptl_pt_index_t**	*pt_index*	
match bits	**ptl_match_bits_t**	*match_bits*	opt. if *options*.PTL_NI_NO_MATCHING
offset	**ptl_size_t**	*remote_offset*	
memory desc	**ptl_handle_md_t**	*md_handle*	opt. if *ack_req* =PTL_NO_ACK_REQ
header data	**ptl_hdr_data_t**	*hdr_data*	user data in header
put user pointer	void *	*user_ptr*	opt. if *ack_req* =PTL_NO_ACK_REQ or *ack_req* =PTL_CT_ACK_REQ or *ack_req* =PTL_OC_ACK_REQ
length	**ptl_size_t**	*length*	*length* argument
data	*bytes*	*md_handle*	user data

target in the *put* and back again in the *acknowledgment* message. If an implementation has another way of tracking the *user_ptr* and *md_handle* at the initiator, then sending the *user_ptr* and *md_handle* should not be necessary.

Notice that the *match_bits*, *md_handle* and *user_ptr* fields in the *put* operation are optional. If the *put* is originating from a non-matching network interface, there is no need for the *match_bits* to be transmitted since the destination will ignore them. Similarly, if no acknowledgment was requested, *md_handle* and *user_ptr* do not need to be sent. If an acknowledgment is requested (either PTL_ACK_REQ, PTL_CT_ACK_REQ, or PTL_OC_ACK_REQ), then the *md_handle* may be sent in the *put* message so that the *target* can send it back to the *initiator* in the *acknowledgment* message. The *md_handle* is needed by the *initiator* to find the right event queue for the acknowledgment event. The *user_ptr* is only required in the case of a full acknowledgment (PTL_ACK_REQ). PTL_CT_ACK_REQ and PTL_OC_ACK_REQ requests do not require the *user_ptr* field to generate the acknowledgment event at the *initiator* of the *put* operation.

A portals header contains 8 bytes of user supplied data specified by the *hdr_data* argument passed to **PtlPut()**. This is useful for out-of-band data transmissions with or without bulk data. The header bytes are stored in the event generated at the *target*. (See Section 3.15.2 on page 90.)

Tables 4.2 and 4.3 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 pieces of information in the *acknowledgment* are the manipulated length, which is determined as the *put* request is satisfied, and the actual offset used.

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 3.10.4 for how pending operations affect when memory descriptors may be unlinked.

If the target match list entry has the PTL_ME_MANAGE_LOCAL flag set, the offset local to the *target* match list entry is used. If the flag is not set, the offset requested by the *initiator* is used. An *acknowledgment* message returns the actual value used.

Lightweight "counting" acknowledgments do not require the actual offset used or user pointer since they do not generate a **ptl_event_t** at the *put* operation *initiator*.

Table 4.2. Acknowledgment: Information Passed in an Acknowledgment.

Information	Type	PtlPut() Argument	Notes
operation	int		indicates an *acknowledgment*
options	unsigned int	*put_md_handle*	*options* field from NI associated with MD
initiator	**ptl_process_t**	*target_id*	echo *target* of *put*
target	**ptl_process_t**	*initiator*	echo *initiator* of *put*
memory descriptor	**ptl_handle_md_t**	*md_handle*	echo *md_handle* of *put*
put user pointer	void *	*user_ptr*	echo *user_ptr* of *put*
offset	**ptl_size_t**	*remote_offset*	obtained from the operation
manipulated length	**ptl_size_t**		obtained from the operation
matched list	**ptl_list_t**		obtained from the operation

Table 4.3. Acknowledgment: Information Passed in a "Counting" Acknowledgment.

Information	Type	PtlPut() Argument	Notes
operation	int		indicates an *acknowledgment*
options	unsigned int	*put_md_handle*	*options* field from NI associated with MD
initiator	**ptl_process_t**	*target_id*	local information on *put* target
target	**ptl_process_t**	*initiator*	echo *initiator* of *put*
memory descriptor	**ptl_handle_md_t**	*md_handle*	echo *md_handle* of *put*
manipulated length	**ptl_size_t**		obtained from the operation

Table 4.4 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 4.4. Get Request: Information Passed in a Get Request — **PtlGet()** and **PtlGetRegion()**.

Information	Type	PtlGet() Argument	Notes
operation	int		indicates a *get* operation
options	unsigned int	*md_handle*	*options* field from NI associated with MD
initiator	**ptl_process_t**		local information
user	**ptl_uid_t**		local information
target	**ptl_process_t**	*target_id*	
portal index	**ptl_pt_index_t**	*pt_index*	
match bits	**ptl_match_bits_t**	*match_bits*	optional if the `PTL_NI_NO_MATCHING` option is set.
offset	**ptl_size_t**	*remote_offset*	
memory descriptor	**ptl_handle_md_t**	*md_handle*	destination of *reply*
length	**ptl_size_t**	*length*	
initiator offset	**ptl_size_t**	*local_offset*	
get user pointer	void *	*user_ptr*	

Table 4.5 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 4.5. Reply: Information Passed in a Reply.

Information	Type	PtlGet() Argument	Notes
operation	int		indicates an *reply*
options	unsigned int	*get_md_handle*	*options* field from NI associated with MD
initiator	**ptl_process_t**	*target_id*	local information on *get* target
target	**ptl_process_t**	*initiator*	echo *initiator* of *get*
memory descriptor	**ptl_handle_md_t**	*md_handle*	echo *md_handle* of *get*
initiator offset	**ptl_size_t**	*local_offset*	echo *local_offset* of *get*
get user pointer	void *	*user_ptr*	echo *user_ptr* of *get*
manipulated length	**ptl_size_t**		obtained from the operation
offset	**ptl_size_t**	*remote_offset*	obtained from the operation
matched list	**ptl_list_t**		obtained from the operation
data	*bytes*		obtained from the operation

Table 4.6 presents the information that needs to be transmitted from the *initiator* to the *target* for an *atomic* operation. The result of an *atomic* operation is a *reply* and (optionally) an *acknowledgment* as described in Table 4.5.

Table 4.6. Atomic Request: Information Passed in an Atomic Request.

Information	Type	PtlAtomic() Argument	Notes
operation	int		indicates the type of *atomic* operation and datatype
options	unsigned int	*put_md_handle*	*options* field from NI associated with MD
ack type	**ptl_ack_req_t**	*ack_req*	
initiator	**ptl_process_t**		local information
user	**ptl_uid_t**		local information
target	**ptl_process_t**	*target_id*	
portal index	**ptl_pt_index_t**	*pt_index*	
memory descriptor	**ptl_handle_md_t**	*put_md_handle*	opt. if *ack_req* =PTL_NO_ACK_REQ
user pointer	void *	*user_ptr*	opt. if *ack_req* =PTL_NO_ACK_REQ or *ack_req* =PTL_CT_ACK_REQ or *ack_req* =PTL_OC_ACK_REQ
match bits	**ptl_match_bits_t**	*match_bits*	optional if the PTL_NI_NO_MATCHING option is set.
offset	**ptl_size_t**	*remote_offset*	
memory descriptor	**ptl_handle_md_t**	*get_md_handle*	destination of *reply*
length	**ptl_size_t**	*put_md_handle*	*length* member
operand	*bytes*	operand	Used in CSWAP and MSWAP operations
data	*bytes*	*put_md_handle*	user data

4.2.2 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 *atomic* messages go through portals address translation (searching a list) and must then pass an access control test. In contrast, *acknowledgment* and *reply* messages bypass the access control checks and the translation step.

Acknowledgment messages include the memory descriptor handle used in the original **PtlPut()** 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 memory descriptor handle. 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 *atomic* request involves mapping the request to a match list entry (or list entry). This step starts by using the portal index in the incoming request to identify a list of match list entries (or list entries). On a matching interface, the list of match list entries is searched in sequential order until a match list entry is found whose match criteria matches the match bits in the incoming request and that accepts the request. On a non-matching interface, the first item on the list is used and a permissions check is performed.

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 *atomic* messages are generated by remote processes and the checks performed for these messages are more extensive. Incoming *put*, *get*, or *atomic* messages may be rejected because:

- the portal index supplied in the request is not valid;
- the match bits supplied in the request do not match any of the match list entries that accepts the request, or
- the access control information provided in the list entry does not match the information provided in the message.

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

A list entry or match list entry may reject an incoming request if the `PTL_ME_OP_PUT` or `PTL_ME_OP_GET` option has not been enabled and the operation is *put*, *get*, or *atomic* (Table 4.7). In addition, a match list entry may reject an incoming request if the length specified in the request is too long for the match list entry and the `PTL_ME_NO_TRUNCATE` option has been enabled. Truncation is always enabled on standard list entries; thus, a message cannot be rejected for this reason on a non-matching network interface.

Also see Sections 2.4 and Figure 2.11.

4.3 Event Generation and Error Reporting

The types of events and when they are generated is discussed in Chapter 3.13. Operations related to memory descriptors, list entries, and match list entries may both generate a full event (of type **ptl_event_t**) and update a

Table 4.7. Portals Operations and ME/LE Flags: A - indicates that the operation will be rejected, and a • indicates that the operation will be accepted.

Target ME/LE Flags	Operation		
	put	*get*	*atomic*
none	-	-	-
PTL_ME_OP_PUT/PTL_LE_OP_PUT	•	-	-
PTL_ME_OP_GET/PTL_LE_OP_GET	-	•	-
both	•	•	•

counting event. There is no implied ordering between the generation of a full event and updating of a counting event, although if the user requests both a full event and a counting event, the implementation must deliver both in a timely fashion.

Acknowledgement events require special attention due the the flexibility Portals provides the user in controlling acknowledgments. An acknowledgment event is only generated if the *initiator* requests an acknowledgement and either the *target* enables sending an acknowledgment in the list entry or an error occurs during the operation. Requesting a full acknowledgement (PTL_ACK_REQ) without an event queue on the associated memory descriptor (or with success events disabled) still results in the generation of a counting event.

Bibliography

[1] N.R. Adiga and et. al. An Overview of the BlueGene/L Supercomputer. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[2] Robert Alverson. Red Storm. In *Invited Talk, Hot Chips 15*, August 2003.

[3] Christian Bell and Dan Bonachea. A new dma registration strategy for pinning-based high performance networks. *Parallel and Distributed Processing Symposium, International*, 0:198a, 2003.

[4] Ron Brightwell, David S. Greenberg, Arthur B. Maccabe, and Rolf Riesen. Massively Parallel Computing with Commodity Components. *Parallel Computing*, 26:243–266, February 2000.

[5] Ron Brightwell, Tramm Hudson, Rolf Riesen, and Arthur B. Maccabe. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[6] Ron Brightwell and Arthur B. Maccabe. Scalability limitations of VIA-based technologies in supporting MPI. In *Fourth MPI Developers' and Users' Conference*, March 2000.

[7] Ron Brightwell and Lance Shuler. Design and implementation of MPI on Puma portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.

[8] Compaq, Microsoft, and Intel. Virtual Interface Architecture Specification Version 1.0. Technical report, Compaq, Microsoft, and Intel, December 1997.

[9] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.

[10] Infiniband Trade Association. *http://www.infinibandta.org*, 1999.

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

[12] Mario Lauria, Scott Pakin, and Andrew Chien. Efficient Layering for High Speed Communication: Fast Messages 2.x. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 1998.

[13] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.

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

[15] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[16] Myricom, Inc. The GM Message Passing System. Technical report, Myricom, Inc., 1997.

[17] Rolf Riesen, Ron Brightwell, and Arthur B. Maccabe. The evolution of Portals, an API for high performance communication. *To be published*, 2005.

[18] Rolf Riesen, Ron Brightwell, Arthur B. Maccabe, Trammell Hudson, and Kevin Pedretti. The Portals 3.3 message passing interface: Document revision 2.0. Technical report SAND2006-0420, Sandia National Laboratories, January 2006.

[19] Lance Shuler, Chu Jong, Rolf Riesen, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[20] Task Group of Technical Committee T11. Information Technology - Scheduled Transfer Protocol - Working Draft 2.0. Technical report, Accredited Standards Committee NCITS, July 1998.

Appendix A

Portals Design Guidelines

Early versions of Portals were based on the idea of using 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. While Portals 3.x supported MPI well for kernel level implementations, more advanced offloading network interfaces and the rising importance of PGAS models exposed several weaknesses. This led to several enhancements that became Portals 4.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 4.x could provide them.

A.1 Mandatory Requirements

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

Partitioned Global Address Space (PGAS) Support. Portals *must* support efficient implementations of typical PGAS languages and programming interfaces.

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 millions of nodes.

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

Multiprocess support. Portals *must* support use of the communication interface by tens of processes 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.

A.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.

PGAS. It *will* be possible to write an efficient implementation of the one-sided and atomic operations found in PGAS models using Portals.

Network Interfaces. It *will* be possible to write an efficient implementation of Portals using a network interface that provides offload support.

Operating Systems. It *will* be possible to write an efficient implementation of Portals using a lightweight kernel *or* Linux as the host OS.

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.

A.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).

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.

Appendix B

README Definition

Each Portals implementation should provide a README file that details implementation-specific choices. This appendix describes 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 3.6.1

Resource Usage. The implementation will be required to consume some user memory for the limits specified in **PtlNIInit()**. The README should document the memory resources required by the implementation and should enumerate the relationship between the memory resources consumed and the limits requested in the desired set of limits passed into **PtlNIInit()**. See Section 3.6.1

Status Registers. Portals define a set of status registers (Section 3.3.7). The type **ptl_sr_index_t** defines the mandatory `PTL_SR_DROP_COUNT`, `PTL_SR_PERMISSION_VIOLATIONS`, and `PTL_SR_OPERATION_VIOLATIONS`, as well as 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 3.3.5 and 3.6.2). 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 64 entries per portal table (Section 3.6). The README file should state how many entries will actually be provided.

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 (Sections 3.10.1 and 3.12.1). Furthermore, if the implementation supports unaligned atomic operations, it should be documented.

Appendix C

Summary of Changes

C.1 Portals 4.0.1

- Specify that PTL_EVENT_AUTO_UNLINK must come after all other events on a list entry / match list entry.
- PTL_EVENT_PT_DISABLED, PTL_EVENT_LINK, PTL_EVENT_AUTO_UNLINK, and PTL_EVENT_AUTO_FREE should provide a *user_ptr* and *ni_fail_type* and not a *start* and *hdr_data* field.
- For clarity, remove "and return" from the description of PTL_MIN, PTL_MAX, PTL_SUM, PTL_PROD, PTL_LOR, PTL_LAND, PTL_BOR, PTL_BAND, PTL_LXOR, PTL_BXOR from Section 3.15.4.

C.2 Portals 4.0

The most recent version of this document described Portals version 3.3 [18]. 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 version 3.3 and the current 4.0 version. Many of the fundamental changes were driven by the desire to reduce the tight coupling required between the application processor and the portals processor, but some additions were made to better support lighter weight communications models such as PGAS.

Foremost, Portals version 4.0 was substantially enhanced to better support the various PGAS programming models. Communication operations that do not include matching were added along with key atomic operations. In addition, the ordering definition was substantially strengthened relative to Portals version 3.3 for small messages. In support of the lightweight communication semantics required by PGAS models, lightweight "counting" events and acknowledgments were added. A **PtlAtomic()** function was added to support functionality commonly provided in PGAS models. Finally, the Portals ordering model was substantially expanded to better support some PGAS models.

An equally fundamental change in Portals version 4.0 adds a mechanism to cope better with the concept of unexpected messages in MPI. Whereas version 3.3 used **PtlMDUpdate()** to atomically insert items into the match list so that the MPI implementation could manage unexpected messages, version 4.0 adds an overflow list where the application provides buffer space that the implementation can use to store unexpected messages. The implementation is then responsible for matching new list insertions to items that have arrived and are resident in the overflow list space. This change was necessary to eliminate round trips between the processor and the NIC for each item that was added to the match list (now named the priority list).

A third major change separated all resources for initiators and targets. Memory descriptors are used by the initiator to describe memory regions while list entries are used by targets to describe the memory region *and* matching criteria (in the case of match list entries). This separation of resources was also extended to events, where the number of event types was significantly reduced and only required fields for a given event type must be defined.

To better offload collective operations, a set of *triggered* operations were added. These operations allow an application to build non-blocking, offloaded collective operations with independent progress. They include variants of both the data movement operations (get and put) as well as the atomic operations.

Another set of changes arise from a desire to simplify hardware implementations. The threshold value was removed from the target and was replaced by the ability to specify that a match list entry is “use once” or “persistent”. List insertions occur *only* at the tail of the list, since unexpected message handling has been separated out into a separate list.

Access control entries were found to be a non-scalable resource, so they have been eliminated. At the same time, it was recognized that the `PTL_LE_OP_PUT` and `PTL_LE_OP_GET` semantics required a form of matching. These two options along with the ability to include user ID based authentication were moved to *permissions fields* on the respective list entry or match list entry.

Ordering only at the message level was found to be insufficient for many PGAS models, which often require ordering of data. Unfortunately, uniformly requiring data ordering could create unnecessary performance constraints. As such, the ordering definition has been expanded to include data ordering and to let the user disable that ordering and message ordering.

Index

D

E

F

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

[*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 Trammell Hudson
c/o OS Research
1527 16th NW #5
Washington, DC 20036
1 Arthur B. Maccabe
Oak Ridge National Laboratory
PO Box 2008
Oak Ridge, TN 37831-6164
1 Neil Pundit
1354 Plumosa Way
Weston, FL 33327
1 Keith Underwood
Oak Ridge National Laboratory
PO Box 2008
Oak Ridge, TN 37831-6164

1 MS 0806 Jim Schutt, 9336
1 MS 1319 Brian Barrett, 1423
1 MS 1319 Ron Brightwell, 1423
1 MS 1319 Doug Doerfler, 1422
1 MS 1319 K. Scott Hemmert, 1422
1 MS 1319 Sue Kelly, 1422
1 MS 1319 Mike Levenhagen, 1422
1 MS 1319 Ron Oldfield, 1423
1 MS 1319 Kevin Pedretti, 1423
1 MS 1319 Lee Ward, 1423
1 MS 0899 Technical Library, 9536 (electronic copy)

Sandia National Laboratories