The Portals 4.0 Network Programming Interface
June 14, 2012 DRAFT

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.

Approved for public release; further dissemination unlimited.
Abstract

This report presents a specification for the Portals 4.0 network programming interface. Portals 4.0 are intended to allow scalable, high-performance network communication between nodes of a parallel computing system. Portals 4.0 are well suited to massively parallel processing and embedded systems. Portals 4.0 represent 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 to support enhanced offload capabilities.
Acknowledgments

Over the years, many people have helped shape, design, and write portals code. 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 contributor(s).
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>9</td>
</tr>
<tr>
<td>List of Tables</td>
<td>10</td>
</tr>
<tr>
<td>List of Implementation Notes</td>
<td>11</td>
</tr>
<tr>
<td>Preface</td>
<td>12</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>13</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>15</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>15</td>
</tr>
<tr>
<td>1.2 Purpose</td>
<td>15</td>
</tr>
<tr>
<td>1.3 Background</td>
<td>16</td>
</tr>
<tr>
<td>1.4 Scalability</td>
<td>17</td>
</tr>
<tr>
<td>1.5 Communication Model</td>
<td>17</td>
</tr>
<tr>
<td>1.6 Zero Copy, OS Bypass, and Application Bypass</td>
<td>18</td>
</tr>
<tr>
<td>1.7 Faults</td>
<td>18</td>
</tr>
<tr>
<td><strong>2 An Overview of the Portals API</strong></td>
<td>19</td>
</tr>
<tr>
<td>2.1 Data Movement</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Completion Events</td>
<td>23</td>
</tr>
<tr>
<td>2.3 Portals Addressing</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Lists and List Entries</td>
<td>26</td>
</tr>
<tr>
<td>2.3.2 Match Lists and Match List Entries</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Modifying Data Buffers</td>
<td>27</td>
</tr>
<tr>
<td>2.5 Ordering</td>
<td>29</td>
</tr>
<tr>
<td>2.5.1 Short Message Ordering Semantics</td>
<td>29</td>
</tr>
<tr>
<td>2.5.2 Long Message Ordering Semantics</td>
<td>30</td>
</tr>
<tr>
<td>2.5.3 Relative Ordering of Overlapping MD and ME Operations</td>
<td>30</td>
</tr>
<tr>
<td>2.5.4 Ordering of Unexpected Messages</td>
<td>30</td>
</tr>
<tr>
<td>2.5.5 Relaxing Message Ordering</td>
<td>30</td>
</tr>
<tr>
<td>2.6 Flow Control</td>
<td>31</td>
</tr>
<tr>
<td>2.7 Multi-Threaded Applications</td>
<td>32</td>
</tr>
<tr>
<td>2.8 Usage</td>
<td>32</td>
</tr>
<tr>
<td><strong>3 The Portals API</strong></td>
<td>35</td>
</tr>
<tr>
<td>3.1 Naming Conventions and Typeface Usage</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Constants</td>
<td>36</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>3.3</td>
<td>Base Types</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Sizes</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Handles</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Indexes</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Match Bits</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Network Interfaces</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Identifiers</td>
</tr>
<tr>
<td>3.3.7</td>
<td>Status Registers</td>
</tr>
<tr>
<td>3.4</td>
<td>Function Arguments and Return Codes</td>
</tr>
<tr>
<td>3.5</td>
<td>Initialization and Cleanup</td>
</tr>
<tr>
<td>3.5.1</td>
<td>PtlInit</td>
</tr>
<tr>
<td>3.5.2</td>
<td>PtlFini</td>
</tr>
<tr>
<td>3.6</td>
<td>Network Interfaces</td>
</tr>
<tr>
<td>3.6.1</td>
<td>The Network Interface Limits Type</td>
</tr>
<tr>
<td>3.6.2</td>
<td>PtlNIInit</td>
</tr>
<tr>
<td>3.6.3</td>
<td>PtlNIFini</td>
</tr>
<tr>
<td>3.6.4</td>
<td>PtlNIStatus</td>
</tr>
<tr>
<td>3.6.5</td>
<td>PtlNIHandle</td>
</tr>
<tr>
<td>3.6.6</td>
<td>PtlSetMap</td>
</tr>
<tr>
<td>3.6.7</td>
<td>PtlGetMap</td>
</tr>
<tr>
<td>3.7</td>
<td>Portal Table Entries</td>
</tr>
<tr>
<td>3.7.1</td>
<td>PtlPTAlloc</td>
</tr>
<tr>
<td>3.7.2</td>
<td>PtlPTFree</td>
</tr>
<tr>
<td>3.7.3</td>
<td>PtlPTDisable</td>
</tr>
<tr>
<td>3.7.4</td>
<td>PtlPTEnable</td>
</tr>
<tr>
<td>3.8</td>
<td>User Identification</td>
</tr>
<tr>
<td>3.8.1</td>
<td>PtlGetUid</td>
</tr>
<tr>
<td>3.9</td>
<td>Process Identification</td>
</tr>
<tr>
<td>3.9.1</td>
<td>The Process Identification Type</td>
</tr>
<tr>
<td>3.9.2</td>
<td>PtlGetId</td>
</tr>
<tr>
<td>3.9.3</td>
<td>PtlGetPhysId</td>
</tr>
<tr>
<td>3.10</td>
<td>Memory Descriptors</td>
</tr>
<tr>
<td>3.10.1</td>
<td>The Memory Descriptor Type</td>
</tr>
<tr>
<td>3.10.2</td>
<td>The I/O Vector Type</td>
</tr>
<tr>
<td>3.10.3</td>
<td>PtlMDBind</td>
</tr>
<tr>
<td>3.10.4</td>
<td>PtlMDBRelease</td>
</tr>
<tr>
<td>3.11</td>
<td>List Entries and Lists</td>
</tr>
<tr>
<td>3.11.1</td>
<td>The List Entry Type</td>
</tr>
<tr>
<td>3.11.2</td>
<td>PtlLEAppend</td>
</tr>
</tbody>
</table>
3.16 Triggered Operations .................................................. 105
3.16.1 PtlTriggeredPut ................................................. 104
3.16.2 PtlTriggeredGet ................................................. 105
3.16.3 PtlTriggeredAtomic ....................................................... 106
3.16.4 PtlTriggeredFetchAtomic ................................................... 107
3.16.5 PtlTriggeredSwap ............................................................. 108
3.16.6 PtlTriggeredCTInc .......................................................... 109
3.16.7 PtlTriggeredCTSet ......................................................... 109
3.17 Deferred Communication Operations ........................................... 110
  3.17.1 PtlStartBundle ............................................................. 110
  3.17.2 PtlEndBundle ............................................................. 111
3.18 Operations on Handles .......................................................... 112
  3.18.1 PtlHandleIsEqual ......................................................... 112
3.19 Summary ................................................................. 112

4 Guide to Implementation .......................................................... 121
  4.1 Run-time Support .............................................................. 121
  4.2 Data Transfer ................................................................. 121
    4.2.1 Sending Messages ......................................................... 121
    4.2.2 Receiving Messages ....................................................... 125
  4.3 Event Generation and Error Reporting ..................................... 125

References ................................................................. 127

Appendix ................................................................. 129
A Portals Design Guidelines ..................................................... 129
  A.1 Mandatory Requirements ................................................... 129
  A.2 The Will Requirements ..................................................... 130
  A.3 The Should Requirements .................................................. 130
B README Definition .......................................................... 133
C Summary of Changes .......................................................... 135
Index ................................................................. 137
List of Figures

2.1 Graphical Conventions ................................................................. 19
2.2 Portals Put (Send) ................................................................. 20
2.3 Portals Get (Receive) from a match list entry .................................. 21
2.4 Portals Get (Receive) from a list entry .......................................... 22
2.5 Portals Atomic Swap Operation ...................................................... 22
2.6 Portals Atomic Sum Operation ....................................................... 23
2.7 Portals LE Addressing Structures .................................................. 24
2.8 Portals ME Addressing Structures .................................................. 25
2.9 Non-Matching Portals Address Translation ........................................ 26
2.10 Matching Portals Address Translation ............................................. 28
2.11 Simple Put Example ................................................................. 32
3.1 Portals Operations and Event Types ................................................. 75
## List of Tables

3.1 Object Type Codes ......................................................... 35  
3.2 Event Type Summary .................................................... 76  
3.3 Event Field Definition .................................................. 80  
3.4 Legal Atomic Operation, Datatype, and Function Combinations ......................................................... 98  
3.5 Portals Data Types .......................................................... 112  
3.6 Portals Functions ........................................................... 114  
3.7 Portals Return Codes ...................................................... 115  
3.8 Portals Constants ............................................................ 116  
4.1 Information Passed in a Send Request ............................... 122  
4.2 Information Passed in an Acknowledgment ......................... 123  
4.3 Information Passed in a “Counting” Acknowledgment ............ 123  
4.4 Information Passed in a Get Request ................................. 123  
4.5 Information Passed in a Reply ............................................ 124  
4.6 Information Passed in an Atomic Request ......................... 124  
4.7 Portals Operations and ME/LE Permission Flags .................. 126
List of Implementation Notes

1. No wire protocol ................................................................. 17
2. Location of event queues and counting events .............................. 21
3. Protected space ................................................................. 21
4. README and portals4.h ....................................................... 35
5. Ordering of Data Structures .................................................. 36
6. Size of handle types ............................................................ 36
7. Unique handles ...................................................................... 37
8. Logical network interfaces ...................................................... 39
9. Multiple calls to `PtlNIInit()` .................................................. 43
10. Memory descriptors that bind inaccessible memory ....................... 52
11. Optimization for Duplicate Memory Descriptors ......................... 55
12. List entries that bind inaccessible memory .................................. 57
13. `PtlLEUnlink()` and unlinked handles ....................................... 62
14. Match list entries that bind inaccessible memory ......................... 64
15. Checking `match_id` ................................................................ 70
16. `PtlMEUnlink()` and unlinked handles ....................................... 71
17. Pending operations and `acknowledgment` ................................... 75
18. Completion of portals operations ............................................ 77
19. Size of event queue and reserved space ..................................... 81
20. PTL_INTERRUPTED return code ............................................. 83
21. Fairness of `PtlEQPoll()` ....................................................... 84
22. Macros using `PtlEQPoll()` ..................................................... 85
23. Filling in the `ptl_event_t` structure ........................................ 85
24. Counting Event Handles ....................................................... 85
25. Minimizing cost of counting events ......................................... 87
26. Fairness of `PtlCTPoll()` ....................................................... 89
27. Macros using `PtlCTPoll()` ..................................................... 90
28. Functions that require communication ...................................... 92
29. Portals Atomic Synchronization ............................................. 103
30. Ordering of Triggered Operations .......................................... 103
31. Implementation of Triggered Operations .................................. 104
32. Triggered Operations Reaching the Threshold ............................ 104
33. Purpose of Bundling ............................................................ 111
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™ 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 proved 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 were born.

We defined the Version 3.0 API in 5. Since then, Portals have gone through three revisions. The latest was Version 3.3 19. 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 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 a atomic 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) or it may act as a gateway between networks.
OS   Operating System.
PM   Message passing layer for SCoreD [12].
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.
<table>
<thead>
<tr>
<th>SMP</th>
<th>Shared Memory Processor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNMOS</td>
<td>Sandia national laboratories/University of New Mexico Operating System.</td>
</tr>
<tr>
<td>Target</td>
<td>A process that is acted upon by a message operation.</td>
</tr>
<tr>
<td>Thread</td>
<td>A context of execution that shares a virtual memory context with other threads.</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol.</td>
</tr>
<tr>
<td>UNIX</td>
<td>A multiuser, multitasking, portable OS.</td>
</tr>
<tr>
<td>VIA</td>
<td>Virtual Interface Architecture.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Overview

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

This document is divided into several sections:

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

Section 2 – An Overview of the Portals 4.0 API.
This section gives a brief overview of the portals API. The goal is to introduce the key concepts and terminology used in the description of the API.

Section 3 – The Portals 4.0 API.
This section describes the functions and semantics of the portals API in detail.

Section 4 – Guide to Implementation.
This section provides a guide to implementors, highlighting subtleties of the standard that are critical to an implementation design. The information transmitted in each type of message and the processing of incoming messages is discussed.

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

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

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

1.2 Purpose

Existing message passing technologies available for supercomputer network hardware do not meet the scalability goals required by emerging massively parallel processing platforms that will have as many as two million processor cores.

---

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

2We use the lower case portals when it is used as an adjective; e.g., portals document, a (generic) portals address, or portals operations. We use the singular when we refer to a specific portal or its attributes; e.g., portal index, portal table, or a (specific) portal address.
cores. This greatly exceeds the capacity for which existing message passing technologies have been designed and implemented.

In addition to the scalability requirements of the network, these technologies must also be able to support a scalable, high performance implementation of the Message Passing Interface (MPI) [15] standard as well as the various partitioned global address space (PGAS) models, such as unified parallel C (UPC), Co-Array Fortran (CAF), and SHMEM [10]. While neither MPI nor PGAS models impose specific scalability limitations, many message passing technologies do not provide the functionality needed to allow implementations of MPI to meet our scalability or performance goals.

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

- Connectionless – Many connection-oriented architectures, such as InfiniBand [11], VIA [9] 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 2.6), 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 avoids scalability limitations for an implementation of MPI:

- Receiver-managed – Sender-managed message passing implementations require a persistent block of memory to be available for every process, requiring memory resources to increase with job size.

- User-level bypass (application bypass) – While OS bypass is necessary for high performance, it alone is not sufficient to support the progress rule of MPI asynchronous operations. After an application has posted a receive, data must be delivered and acknowledged without further intervention from the application.

- Unexpected messages – Few communication systems have support for receiving messages for which there is no prior notification. Support for these types of messages is necessary to avoid flow control and protocol overhead.

### 1.3 Background

Portals were originally designed for and implemented on the nCUBE-2 machine as part of the SUNMOS (Sandia/UNM OS) [14] and Puma [20] lightweight kernel development projects. Portals went through three design phases [18], 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 have been very successful in meeting the needs of such large machines, not only as a layer for a high-performance MPI implementation [8], 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 concept of unexpected messages to Portals. 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 data that experience has shown to be unnecessary.

1.4 Scalability

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

Similarly, a quality implementation is needed for Portals to be scalable. 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 provide reliable, ordered delivery of messages between pairs of processes. Portals are connectionless: a process is not required to explicitly establish a point-to-point connection with another process in order to communicate. Moreover, all buffers used in the transmission of messages are maintained in user space. The target process determines how to respond to incoming messages, and messages for which there are no buffers are discarded.

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>No wire protocol</td>
</tr>
<tr>
<td>This document does not specify a wire protocol. Portals require a reliable communication layer. Whether that is achieved through software or hardware is up to the implementation. For example, for Red Storm two reliability protocols were implemented; one by Cray and one by Sandia [6].</td>
</tr>
</tbody>
</table>

1.5 Communication Model

Portals combine 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 allow 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 [21], VIA [9], FM [13], GM [17], PM [12], and Portals are examples of OS bypass mechanisms.

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

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 successful 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, it is possible that a reply will be generated by another component in the system, bypassing the application.

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.](image)

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 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 (Section 4). When the data has been put into the remote memory descriptor (or been discarded), the target optionally sends an acknowledgment message.
Figure 2.2. Portals Put (Send): Note that the put request ➀ is part of the header and the data ➁ 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 there may be multiple logical network interfaces associated with a single physical network interface. 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, even if they share a physical layer. 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.
**IMPLEMTATION 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.

**IMPLEMTATION 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.

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.](image-url)

Portals address translations (matching and permissions checks) are 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_.

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

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

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

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 used PtlAtomic() instead of PtlFetchAtomic().
2.2 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.

2.3 Portals Addressing

One-sided data movement models (e.g., SHMEM [10], ST [21], and MPI-2 [16]) 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 [7]. 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 2.3.2.

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 once the correct buffer has been identified. The permissions check has two components: the target of the message must allow the initiator to access this buffer and must allow the specified operation type. Each list entry and match list entry has specifiers of 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

Figure 2.6. Portals Atomic (sum is shown). An atomic sum operation in memory described by a list entry.
modify the Portals state in any way, except to update the status registers (see Section 3.3.7).

Figure 2.7. 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.

Figures 2.7 and 2.8 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.

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 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.

\(^1\) A logical rank can be substituted for the combination of node ID and process ID when logical endpoint addressing is used.
The third list that is associated with each portal index is transparent to the user. Each time a message is delivered into the overflow list, its header is linked into the unexpected list. The unexpected list cannot be directly manipulated by the user, however it may be searched for matching entries. 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 \texttt{PtlLEUnlink()} or \texttt{PtlMEUnlink()} successfully unlinking the 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., \texttt{PTL\_EVENT\_PUT}) has a corresponding overflow event (e.g., \texttt{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 \texttt{mlength} in the full event is less than the \texttt{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, the message is discarded and the \texttt{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 \texttt{PTL\_PT\_FLOWCTRL} full event is generated in the event queue associated with the portal table entry (see Section 2.6).

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).

### 2.3.1 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.

![Diagram of Non-Matching Portals Address Translation](image)

**Figure 2.9. Non-Matching Portals Address Translation.**

Figure 2.9 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; however, 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 unlinked 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.

### 2.3.2 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 “don’t care” bits and a set of “must 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. 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 2.10 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\(^2\), 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, node identifier, and process identifier.

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. In contrast, 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. If the end of the priority list has been reached, the address translation continues with the overflow list.

### 2.4 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

\(^2\)Even if an incoming message matches the match criteria of a match list entry, the match list entry can reject operations because the memory region does not have sufficient space or the wrong operation is attempted. See Section 3.10.
Figure 2.10. Matching Portals Address Translation.

After node, process, matching NI selection, and selecting correct Portals table entry.

- Priority list empty?
  - yes: discard message
  - no: get next matching list entry
  - match?: yes: permissions pass?
    - yes: perform operation
    - no: unlink ME? (min free, or use once)
    - yes: unlink ME
    - no: discard message
    - increment permission violations count
  - no: overflow list empty?
    - yes: post dropped message event
    - no: get next matching list entry
    - match?: yes: increment drop count
    - no: increment drop count

- End
implementation can utilize user memory as scratch space and staging buffers for operations larger than this threshold. When the operation is larger than \textit{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 \textit{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 \textit{put}. Similarly, an implementation must not alter data in a user buffer that is used in a \textit{reply} operation. This is independent of whether the operation succeeds or fails.

2.5 Ordering

There are two types of ordering typically defined by user 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 user level languages and message passing APIs. As such, Portals has a variety of options to control message and data ordering. As a general overview, Portals has an option to request data ordering between a pair of end-points when targeting a specific list entry or match list entry up to some message size (\textit{max\_waw\_ordered\_size} and \textit{max\_war\_ordered\_size} in \texttt{ptl\_ni\_limits\_t} as an argument to \texttt{PtlNIInit()}) to support semantics such as the relaxed ordering semantics of UPC. This ordering can be strengthened by an implementation, and the application can request \texttt{PTL\_TOTAL\_DATA\_ORDERING} in the \texttt{features} field of \texttt{PtlNIInit()}. For all messages, including those longer than the \textit{max\_waw\_ordered\_size} (or \textit{max\_war\_ordered\_size}, as appropriate), message ordering is provided unless it is disabled using the \texttt{PTL\_MD\_UNORDERED} option in the \texttt{ptl\_md\_t}. This supports the MPI two-sided message ordering requirements while providing the flexibility to disable ordering when it is not needed.

2.5.1 Short Message Ordering Semantics

The default ordering semantics for Portals messages differ for short and long messages. When one message that stores data (\textit{put}, \textit{atomic}) is followed by a message that stores data or retrieves data (\textit{put}, \textit{atomic}, \textit{get}) from the same initiator to the same target have a length that is less than or equal to the \textit{max\_waw\_ordered\_size} field of the \texttt{ptl\_ni\_limits\_t} structure returned in the \texttt{actual} argument of \texttt{PtlNIInit()}, 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 (\textit{get}) is followed by a second message that store data (\textit{put}, \textit{atomic}) from the same initiator to the same target have a length that is less than or equal to the \textit{max\_war\_ordered\_size} field of the \texttt{ptl\_ni\_limits\_t} structure returned in the \texttt{actual} argument of \texttt{PtlNIInit()}, 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. Non-overlapping bytes within or between messages are not ordered relative to each other unless \texttt{PTL\_TOTAL\_DATA\_ORDERING} is requested in \texttt{PtlNIInit()} and the implementation indicates that request is honored through the \texttt{actual} limits field. Support for the ordering of bytes within or between messages is an optional feature, since some implementations may be unable to provide it. Furthermore, the completion for the two messages is not guaranteed to occur in the order the messages were issued. This ordering requirement means that a \textit{get} following a \textit{put} to the same LE (or ME) is guaranteed to see the result of the \textit{put} for all of the overlapping bytes if and only if the length of both the \textit{get} and the \textit{put} is less than or equal to \textit{max\_waw\_ordered\_size}. In addition, when a \textit{put} follows a \textit{get} to the same LE (or ME), the \textit{get} is guaranteed to see the state of the data before the \textit{put} if the \textit{get} is smaller than \textit{max\_war\_ordered\_size}. These ordering semantics can be loosened by using the \texttt{PTL\_MD\_UNORDERED} option on the memory descriptor at the initiator (see Section 3.10). Note that replies and acknowledgments do not require ordering.
2.5.2 Long Message Ordering Semantics

The default ordering semantics for Portals that have a length that is longer than the `max_waw_ordered_size` (or `max_war_ordered_size`, as appropriate) field of the `ptl_ni_limits_t` structure returned in the `actual` argument of `PtlNIInit()` are much weaker. For long messages, the ordering semantics only require that message headers for a pair of processes arrive in order at the target and that message matching order is maintained. 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 network can use a retransmission protocol on the wire that 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. An initiator may explicitly indicate that a message does not have to be ordered at the target using an option on the MD (see Section 3.10). 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`. To simplify the implementation of `shmem_fence()`, an implementation may choose to provide a `max_waw_ordered_size` that is equal to the `max_msg_size`.

2.5.3 Relative Ordering of Overlapping MD and ME Operations

There is also an issue with the ordering of data. When data arrives in a region described by a list entry that happens to overlap with a region described by a memory descriptor with an active operation, the ordering of data operations is undefined. Data is only available for transmit after the event corresponding to the arriving message has been delivered. Thus, triggered operations are safe, since they do not trigger until the counting event is delivered.

2.5.4 Ordering of Unexpected Messages

Unexpected messages pose a particular challenge for ordering semantics. The unexpected list maintains insertion ordering, although headers in the middle of the list may be removed first based on matching criteria. Data delivery into the overflow list entry which generated the header 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.

2.5.5 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 `ptl_md_t` used to send the data (see Section 3.10). This turns off both header and data ordering.
2.6 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 and buffer 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 adopts the philosophy that such behavior will lead to slow application performance anyway. Thus, if the application causes exhaustion of resources, recovery from this condition can be very slow. It must, however, be possible to recover.

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 implementation must disable the portal table entry and deliver a PTL_EVENT_PT_DISABLED full event to the application. At this point, all messages targeting that portal table entry for that process must be dropped until PtiPTEnable() 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. Note that a reply does not target a portal table entry and is not dropped. In addition, the PTL_EVENT_ACK event associated with that message (and subsequent in flight messages) fails. If the PTL_EVENT_ACK event is a full event, it will set the ni_fail_type variable to PTL_NI_PT_DISABLED. The application (e.g. MPI library) must then use a second portal table entry to recover from the overflow. The user must quiesce the library (e.g. MPI), ensure that resources are available, re-enable the portal table entry, and restart communications. 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.

Discussion: It is important to note that flow control events are only returned to the initiator in the PTL_EVENT_ACK; thus, it is necessary for a user to request acknowledgments at the initiator and allow acknowledgments at the target to receive notice of a flow control event at the initiator.

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 does not find a match in either the priority list or the overflow list, flow control must be invoked. Essentially, by enabling flow control, the user is indicating that all incoming messages should match in one of the two lists. 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. None of these scenarios apply if flow control is disabled.

Note that flow control applies to incoming message processing. Local operations which generate events (such as a call to PtlEAppend() or PtlMEAppend() which result in the generation of link or overflow events) are not required to trigger flow control and may cause the event queue to overflow, resulting in dropped events.

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 of the event queue, but are not required to do so.
2.7 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.

2.8 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 2.11 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.

![Diagram of Portals structures for one-sided put operation.](image)

**Figure 2.11. 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.
Another example is Figure 2.6 which is simpler than Figure 2.5 and probably more likely to be used. 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.
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 **Ptl** 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 **Ptl**, 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>
<thead>
<tr>
<th>yy</th>
<th>xx</th>
<th>Name</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI</td>
<td>ni</td>
<td>Network Interface</td>
<td>3.6</td>
</tr>
<tr>
<td>PT</td>
<td>pt</td>
<td>Portal Table Entry</td>
<td>3.7</td>
</tr>
<tr>
<td>LE</td>
<td>le</td>
<td>List Entry</td>
<td>3.11</td>
</tr>
<tr>
<td>ME</td>
<td>me</td>
<td>Matching list Entry</td>
<td>3.12</td>
</tr>
<tr>
<td>MD</td>
<td>md</td>
<td>Memory Descriptor</td>
<td>3.10</td>
</tr>
<tr>
<td>EQ</td>
<td>eq</td>
<td>Event Queue</td>
<td>3.13</td>
</tr>
<tr>
<td>CT</td>
<td>ct</td>
<td>Count</td>
<td>3.14</td>
</tr>
</tbody>
</table>

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.

**IMPLEMENTATION NOTE 4:**

README and portals4.h

Each implementation must supply an include file named **portals4.h** with the definitions specified in this document. There should also be a README file that explains implementation specific details. For example, it should list the limits (Section 3.6.1) for this implementation and provide a list of status registers that are provided (Section 3.3.7). See Appendix B for a template.
3.2 Constants

The portals API defines a number of constants. Constants defined in this specification must be compile time constants and must be valid labels for switch statements. Constants are generally associated with a base type in which constants are stored. Implementations are given implementation freedom regarding constants and their associated base types, constrained only by the compile time link requirement.

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 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.

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.

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

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.


### 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 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 types of indexes that can be used to access the status registers. Only three indexes are defined for all implementations: `PTL_SR_DROP_COUNT`, which identifies the status register that counts the dropped requests for the interface, `PTL_SR_PERMISSIONS_VIOLATIONS`, which counts the number of attempted permission violations, and `PTL_SR_OPERATIONS_VIOLATIONS`, which 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 types of values held in status registers. This is a signed integer type. The size is implementation dependent but must be at least 32 bits.

### 3.4 Function Arguments and Return Codes

Unless otherwise noted, an implementation is not required to check the validity of any arguments to a portals 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 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 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()` which 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 but 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**

```c
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(). PtlInit() must not be called after the portals library has been finalized.

Function Prototype for PtlFini

```c
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 implemented by the process or embedded in the underlying network. Interfaces are treated as independent entities to make it easier to cache information on individual network interface cards. In addition to supporting physical interfaces, each 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 (providing the full cross-product of possibilities) and must be provided for each physical interface such that a physical interface can be opened as four logical interfaces.

**IMPLEMENTATION NOTE 8:**

A logical interface is very similar to a physical interface. Like a physical interface, a logical interface is a "well known" interface; i.e. it is a specific physical interface with a specific set of properties. One additional burden placed on the implementation is the need for the initiator to place 2 bits in the message header to identify to the target the logical interface on which this message was sent. In addition, all logical interfaces within a single process that are associated with a single physical interface must share a single node ID and Portals process ID.

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 3.12 for a discussion of updating portal table entries using the PtlMEAppend() function. 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 match list entry) is associated with a specific logical network interface. The association to a logical network interface is established when the object is created and is encoded in the handle for the object.

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 PtlNIFin() 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.

**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.

The portals API also defines the `PtlNIStatus()` function (Section 3.6.4) to query the status registers for a logical network interface, and the `PtlNIHandle()` function (Section 3.6.5) to determine the logical network interface with which an object is associated.

### 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. The two lists are of type `ptl_ni_limits_t` and include the following members:

```c
typedef struct {
    int max_entries;
    int max_unexpected_headers;
    int max_mds;
    int max_eqs;
    int max_cts;
    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 have a `max_pt_index` of at least 63.
**max_iovecs**
Maximum number of I/O vectors for a single memory descriptor 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 2.5, 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. For subsequent calls to **PtlNIInit()** from within the same process (either by different threads or the same thread), the desired limits will be ignored and the call will return the existing network interface handle and the actual limits. Calls to **PtlNIInit()** increment a reference count on the network interface and must be matched by a call to **PtlNIFini()**.

**Function Prototype for PtlNIInit**

```c
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.

**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.

**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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_NI_MATCHING</td>
<td>Request that the interface specified in iface be opened with matching enabled.</td>
</tr>
<tr>
<td>PTL_NI_NO_MATCHING</td>
<td>Request that the interface specified in iface be opened with matching disabled. PTL_NI_MATCHING and PTL_NI_NO_MATCHING are mutually exclusive.</td>
</tr>
<tr>
<td>PTL_NI_LOGICAL</td>
<td>Request that the interface specified in iface be opened with logical endpoint addressing (e.g. MPI communicator and rank or SHMEM PE).</td>
</tr>
<tr>
<td>PTL_NI_PHYSICAL</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

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

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

**Return Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_OK</td>
<td>Indicates success.</td>
</tr>
<tr>
<td>PTL_NO_INIT</td>
<td>Indicates that the portals API has not been successfully initialized.</td>
</tr>
<tr>
<td>PTL_ARG_INVALID</td>
<td>Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.</td>
</tr>
</tbody>
</table>
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.

### IMPLEMENTATION NOTE 9:

Multiple calls to `PtlNIInit()`

If `PtlNIInit()` gets called more than once per logical interface, then the implementation should fill in `actual` and `ni_handle`. It should ignore `pid`. `PtlGetId()` or `PtlGetPhysId()` (Section 3.9) can be used to retrieve the `pid`.

#### 3.6.3 PtlNIFinin

The `PtlNIFinin()` 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 `PtlNIFinin()`. 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 PtlNIFinin**

```c
int PtlNIFinin(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

```c
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.

Discussion: Only three status registers are currently required: a drop count register (PTL_SR_DROP_COUNT), an attempted permissions violation register (PTL_SR_PERMISSIONS_VIOLATIONS), and an attempted operation violation register (PTL_SR_OPERATIONS_VIOLATIONS). Implementations may define additional status registers. Identifiers for the indexes associated with these registers should start with the prefix PTL_SR_.

Return Codes

- `PTL_OK` Indicates success.
- `PTL_NO_INIT` Indicates that the portals API has not been successfully initialized.
- `PTL_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

```c
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 a logical identifiers (rank) to physical 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() and PtlGetMap(). 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

```c
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 logical 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_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.
- **PTL_IGNORED** Indicates that the implementation does not support dynamic changing of the logical identifier map, likely due to integration with a static run-time system.
3.6.7 PtlGetMap

The PtlGetMap() function retrieves the mapping from a 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 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

```c
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 2.6) 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(
    pti_handle_ni_t ni_handle,
    unsigned int options,
    pti_handle_eq_t eq_handle;
    pti_pt_index_t pt_index_req,
    pti_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 operations performed on list entries attached to the portal table entry. If this argument is PTL_EQ_NONE, operations performed on 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 2.6).

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, it may be helpful to know that the entries are only used once and that size is not part of the matching criteria. For a non-matching logical network interface, the implementation may be able to provide better optimizations with a persistent LE, and truncation is required.

### 3.7.2 PtlPTFree

The **PtlPTFree()** function releases the resources associated with a portal table entry.

**Function Prototype for PtlPTFree**

```c
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 that 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**.

**Function Prototype for PtlPTDisable**

```c
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 disable.
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: After successful completion, no other messages will be accepted on this portal table entry and no more events associated with this portal table entry will be delivered. Replies arriving at this initiator will continue to succeed.

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

<table>
<thead>
<tr>
<th>ni_handle</th>
<th>input</th>
<th>The interface handle to use.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt_index</td>
<td>input</td>
<td>The value of the portal index to enable.</td>
</tr>
</tbody>
</table>

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.

3.8 User Identification

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

```c
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 \textit{pid}. In turn, a logical address uses a logical index within a translation table specified by the application (the \textit{rank}) to identify another process.

\begin{verbatim}
typedef union {
    struct {
        pti_nid_t nid;
        pti_pid_t pid;
    } phys;
    pti_rank_t rank;
} pti_process_t;
\end{verbatim}

### 3.9.2 PtlGetId

**Function Prototype for PtlGetId**

\begin{verbatim}
int PtlGetId(ptl_handle_ni_t ni_handle,
              pti_process_t *id);
\end{verbatim}

**Arguments**

- \textit{ni_handle} input: A network interface handle.
- \textit{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.

**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.

**Return Codes**

- \texttt{PTL\_OK} Indicates success.
- \texttt{PTL\_ARG\_INVALID} Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.
- \texttt{PTL\_NO\_INIT} Indicates that the portals API has not been successfully initialized.

### 3.9.3 PtlGetPhysId

**Function Prototype for PtlGetPhysId**

\begin{verbatim}
int PtlGetPhysId(ptl_handle_ni_t ni_handle,
                 pti_process_t *id);
\end{verbatim}
Arguments

\texttt{ni\_handle} \hspace{1cm} \textbf{input} \hspace{1cm} A network interface handle.

\texttt{id} \hspace{1cm} \textbf{output} \hspace{1cm} 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.

\textbf{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.

Return Codes

\texttt{PTL\_OK} \hspace{1cm} Indicates success.

\texttt{PTL\_ARG\_INVALID} \hspace{1cm} Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.

\texttt{PTL\_NO\_INIT} \hspace{1cm} Indicates that the portals API has not been successfully initialized.

3.10 Memory Descriptors

A memory descriptor contains information about a region of a process’ memory and optionally points to an event queue where information about the operations performed on the memory descriptor are recorded. Memory descriptors are initiator side resources that are used to encapsulate an association with 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). \texttt{PtlMDBind()} is used to create a memory descriptor and \texttt{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, even though the entire region is not currently allocated. If the application issues a portals operation (e.g. \texttt{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 \texttt{ni\_fail\_type} to \texttt{PTL\_NI\_SEGV}. If the memory descriptor sets the \texttt{PTL\_IOVEC} option, the memory region(s) described by the \texttt{ptl\_iovec\_t} must all be accessible within the application.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{IMPLEMENTATION NOTE 10:} & Memory descriptors that bind inaccessible memory \\
\hline
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 on the transmit side for some implementations. \\
\hline
\end{tabular}
\end{table}
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. As noted in Section 3.1, the ordering of the data structure can be optimized by the implementation.

```c
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 alignment restrictions on the starting address or the length of the region; although unaligned messages may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

**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_SUCCESS_DISABLE` Specifies that this memory descriptor should not generate full events that indicate success. This is useful in scenarios where the application does not need normal full events, but does require failure information to enhance reliability.
- `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 until the message completes (e.g., a PTL_EVENT_SEND full event is received or a corresponding counting event has been incremented).

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.

c_t_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 match list entry or memory descriptor 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 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 is unlikely to perform well in most implementations.

```c
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
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 would invoke the appropriate memory registration functions.

**Function Prototype for PtlMDBind**

```c
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 API.
- **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 11:** 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 should support this usage model and may desire to optimize for it.

### 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**

```c
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). A list is created using the `PtlLEAppend()` function, which 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. If the `PTL_TARGET_BIND_INACCESSIBLE` bit is set in the `features` field (Section 3.6.1) of the `desired` limits argument passed to `PtlNIInit()` (Section 3.6.2), and the implementation honors the request by returning `PTL_TARGET_BIND_INACCESSIBLE` bit in the `actual` limits, 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, 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.
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 simply 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 or PTL_ME_USE_ONCE option set—in the priority list. Thus, overflow list entries will not be required.

**Discussion:** 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. In contrast, when PtlMEUnlink() is called, the application is instructing the implementation to discard any state associated with the buffer.

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. As noted in Section 3.1, the ordering of the data structure can be optimized by the implementation.

**Discussion:** The list entry (LE) has a number of fields in common with the memory descriptor (MD). The overlapping fields have the same meaning in the LE as in the MD; however, since initiator and target resources are decoupled, the MD is not a proper subset of the LE, and the options field has a different meaning based on whether it is used at an initiator or target, it was deemed undesirable and cumbersome to include a “target MD” structure that would be included as an entry in the LE.
**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.

```c
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 match list entry. The `start` member specifies the starting address for the memory region and the `length` member specifies the length of the region. The `start` member can be NULL provided that the `length` member is zero. Zero-length buffers (NULL 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). There are no alignment 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 `PtlNIStatus()` register indexed by PTL_SR_PERMISSIONS_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_OPERATIONS_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_OPERATIONS_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: Every list entry responds to reply operations. Nor is it considered an error to have a list entry that responds to both put and get operations. In fact, it is often desirable for a list entry used in an atomic operation to be configured to respond to both put and get operations.

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. 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 ptilovec_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 ptilovec_t elements referred to by the start argument cannot be changed or released until the message completes (e.g. a PTL_EVENT_SEND full event is received or a corresponding counting event has been incremented).

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 that indicate success. This is useful in scenarios where the application does not need normal full events, but does require failure information to enhance reliability.
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. If the list is currently uninitialized, the PtlLEAppend() function creates the first entry in the list.

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,
                pti_index_t pt_index,
                const pti_le_t *le,
                pti_list_t pti_list,
                void *user_ptr,
                pti_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.
- `pti_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\(^1\).
- `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.

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

\(^1\)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.
entry is on the overflow list and 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, 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 13:**

`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**

```c
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 `ptlssearchop` is set to `PTL_SEARCH_ONLY`, the unexpected
list is searched to support the MPI_Probe functionality. If \texttt{ptlsearchop} is set to \texttt{PTL\_SEARCH\_DELETE}, the unexpected list is searched and any matching items are deleted. When used with \texttt{PTL\_SEARCH\_ONLY}, a \texttt{PTL\_EVENT\_SEARCH} event with \texttt{ni\_fail\_type PTL\_NI\_OK} is generated when a matching message is found in the unexpected list. When used with \texttt{PTL\_SEARCH\_DELETE}, the event that is generated corresponds to the type of operation that is found (e.g. \texttt{PTL\_EVENT\_PUT\_OVERFLOW}, \texttt{PTL\_EVENT\_GET\_OVERFLOW}, \texttt{PTL\_EVENT\_ATOMIC\_OVERFLOW}, or \texttt{PTL\_EVENT\_FETCH\_ATOMIC\_OVERFLOW}). In either case, if no matching message is found, a \texttt{PTL\_EVENT\_SEARCH} event is generated with a failure indication of \texttt{PTL\_NI\_NO\_MATCH}. If the list entry specified in the \texttt{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 \texttt{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 \texttt{PtlLESearch()} that uses a persistent LE can cause multiple matches.

**List Entry Search Operation Constants (ptl_search_op_t)**

- \texttt{PTL\_SEARCH\_ONLY} Use the LE/ME to search the overflow list, without consuming an item in the list.
- \texttt{PTL\_SEARCH\_DELETE} Use the LE/ME to search the overflow list and delete the item from the list.

**Function Prototype for PtlLESearch**

```c
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**

- \texttt{ni\_handle} input The interface handle to use.
- \texttt{pt\_index} input The portal table index that should be searched.
- \texttt{le} input Provides values for the user-visible parts of a list entry to use for searching.
- \texttt{ptlsearchop} input Determines whether the function only searches the list or searches the list and deletes the matching entries from the list.
- \texttt{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\textsuperscript{2}.

\textsuperscript{2}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 \texttt{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.
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.

### 3.12 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. A match list (priority list or overflow list) is created using the `PtlMEAppend()` function, which 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 memory descriptor, a match list entry describes a memory region using a base address and length. If the `PTL_TARGET_BIND_INACCESSIBLE` bit is set in the `features` field (Section 3.6.1) of the `desired` limits argument passed to `PtlNIInit()` (Section 3.6.2), and the implementation honors the request by returning `PTL_TARGET_BIND_INACCESSIBLE` bit in the `actual` limits, 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, even though the entire region is not currently allocated. 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_ME_IS_ACCESSIBLE` option to indicate that the entire memory space described by the ME 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.

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 simply 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 delivered.

**IMPLEMENTATION NOTE 14:** Match 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).
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.

Discussion: 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.

Matching list entries can be appended to a network interface without the PTL_NI_NO_MATCHING option set; however, an NI with the PTL_NI_LOGICAL option set changes the interpretation of the match_id.

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. As noted in Section 3.1, the ordering of the data structure can be optimized by the implementation.

Discussion: The match list entry (ME) has a number of fields in common with the memory descriptor (MD). The overlapping fields have the same meaning in the ME as in the MD; however, since initiator and target resources are decoupled, the MD is not a proper subset of the ME, and the options field has different meaning based on whether it is used at an initiator or target, it was deemed undesirable and cumbersome to include a “target MD” structure that would be included as an entry in the ME.

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. The **start** member can be NULL provided that the **length** member is zero. 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 LE). There are no alignment restrictions on buffer alignment, the starting address or the length of the region; although unaligned messages may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

**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 **PtlStatus()** register indexed by **PTL_SR_PERMISSIONS_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_OPERATIONS_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_OPERATIONS_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, it is often desirable for a match list entry used in an **atomic** operation to be configured to respond to both **put** and **get** operations.
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 unlinked. 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. 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 that indicate success. This is useful in scenarios where the application does not need normal full events, but does require failure information to enhance reliability.

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: 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.

Discussion: 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.

Discussion: 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: 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. If the list is currently uninitialized, the PtlMEAppend() function creates the first entry in the list.

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. Inserts 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.

**Match List 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 PtlMEAppend**

```c
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.

### IMPLEMENTATION NOTE 15:

Checking **match_id**

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. The presence of unexpected message state associated with an ME in the overflow list should not cause **PTL_IN_USE** to be returned. Instead, this state is discarded. 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. 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**.

---

3Tying 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 16: **PtlMEUnlink()** and unlinked handles

**PtlMEUnlink()** may be used to unlink match list entries which are use-once. In this case, there is a race condition between a network operation causing the match list entry to unlink and the match list entry being explicitly unlinked. Requiring the handle to remain valid until the next call to **PtlMEAppend()** 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 **PtlMEUnlink()** for significantly longer than specified.

**Function Prototype for PtlMEUnlink**

```c
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 **ptlsearchop** is set to **PTL_SEARCH_ONLY**, the unexpected list is searched to support the MPI_Probe functionality. If **ptlsearchop** 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 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 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 ptlsearchop 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.
- `ptlsearchop` 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 entry⁴.

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.

3.13 Events and Event Queues

Event queues are used to log operations performed on local list entries, match list entries, memory descriptors, or portal table entries. In particular, they signal the end of a data transmission into or out of a memory region. They can

⁴Tying 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.
also be used to hold acknowledgments for completed put operations and indicate when a match 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 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 a 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.

- **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 2.6.

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.

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.

Discussion: 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 2.3). 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 relength and mlength of the full event may be used to determine whether the message was fully delivered or truncated.

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 67, also in Section 3.12.1.)

**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. (Also see implementation note 17.)

**IMPLEMENTATION NOTE 17:** Pending operations and acknowledgment

If a user attempts to unlink a list entry or match list entry while it has operations pending, the implementation should return PTL_IN_USE until the operation has completed. Since users cannot know when events occur, the implementer has a certain amount of freedom honoring unlink requests or returning PTL_IN_USE.
Table 3.2 summarizes the portals event types. In the table we use the word local to describe the location where the event is delivered; it can be the initiator or the target of an operation.

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

<table>
<thead>
<tr>
<th>Event Type</th>
<th>initiator</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_EVENT_GET</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_GET_OVERFLOW</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_PUT</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_PUT_OVERFLOW</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC_OVERFLOW</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_FETCH_ATOMIC</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_FETCH_ATOMIC_OVERFLOW</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_REPLY</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_SEND</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_ACK</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_PT_DISABLED</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_LINK</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_AUTO_UNLINK</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_AUTO_FREE</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>PTL_EVENT_SEARCH</td>
<td></td>
<td>•</td>
</tr>
</tbody>
</table>

### 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 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 remote errors 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, unless: 1) the message is successfully delivered at the target and the remote target has disabled event generation, or 2) a type of failure is delivered in the PTL_EVENT_SEND that would indicate that a PTL_EVENT_ACK event would never be received. 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 18:**

<table>
<thead>
<tr>
<th>Completion of portals operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports 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.</td>
</tr>
</tbody>
</table>

### 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. As noted in Section 3.1, the ordering of the data structure can be optimized by the implementation.

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.
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;

These fields are included in a structure:

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 the 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 PtILEAppend() or PtIMEAppend(). 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.
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_SEARCH events, the manipulated length is the number of bytes which would have been manipulated if the list entry had been passed to `PtlLEAppend()` or `PtlMEAppend()`. For all other operations, the manipulated length is the number of bytes of memory manipulated (delivered into or read from memory) at the target, which may be less than `rlength` in the case of truncated operations.

**remote_offset**

The offset requested/used by the other end of the link. 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` (`ptl_process_t`).

**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**

Is 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.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>type</th>
<th>initiator</th>
<th>pt_index</th>
<th>ptl_list</th>
<th>tid</th>
<th>match_bits</th>
<th>length</th>
<th>remote_offset</th>
<th>start</th>
<th>user_ptr</th>
<th>hdr_data</th>
<th>nil_fail_type</th>
<th>atomic_operation</th>
<th>atomic_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_EVENT_GET</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_GET_OVERFLOW</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_PUT</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_PUT_OVERFLOW</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC_OVERFLOW</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_FETCH_ATOMIC</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_FETCH_ATOMIC_OVERFLOW</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_REPLY</td>
<td>•</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_SEND</td>
<td>•</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_ACK</td>
<td>•</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_PT_DISABLED</td>
<td>•</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_LINK</td>
<td>•</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_AUTO_UNLINK</td>
<td>•</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_AUTO_FREE</td>
<td>•</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_EVENT_SEARCH</td>
<td>•••••</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.13.5 PtlEQAlloc

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

Function Prototype for PtlEQAlloc

```c
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.

Discussion:  
An event queue has room for at least `count` number of full events. The event queue is circular. If flow control is not enabled on the portal table entry (Sections 3.7.1 and 2.6), then older events will be overwritten by new ones if they are not removed in time by the user, using the functions PtlEQGet(), PtlEQWait(), or PtlEQPoll(). It is up to the user to determine the appropriate size of the event queue to prevent this loss of events.

Return Codes

- **PTL_OK**  
  Indicates success.

- **PTL_NO_INIT**  
  Indicates that the portals API has not been successfully initialized.

- **PTL_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 19:

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 once it is freed.
Function Prototype for PtlEQFree

```c
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

```c
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**

```c
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()`.

**Implementation Note 20:**

The return code of **PTL_INTERRUPTED** adds an unfortunate degree of complexity to the `PtlEQWait()` and `PtlEQPoll()` functions; however, it was deemed necessary to be able to interrupt waiting functions for the sake of applications that need to tolerate failures. Hence, this approach to dealing with the conflict of reading and freeing full events was chosen.

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 have been retained in the API for backward compatibility.

---

**IMPLEMENTATION NOTE 21:**

Fairness of `PtlEQPoll()`

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

---

Function Prototype for `PtlEQPoll`

```c
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 Section 3.13.7 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 enabled by attaching an `ptl_handle_ct_t` to a memory descriptor or match list entry and by specifying which operations are to be counted in the options field. By default, counting events count the total number of operations; however, they can be set to count the number of bytes transferred for the associated operations (if they are “success” events) using an option in the structure they are attached to.

Counting events mirror full events in virtually every way. They can be used to log the same set of operations performed on local match list entries or memory descriptors that event queues log. Counting events introduce an additional type, the counting event handle: `ptl_handle_ct_t`. A `ptl_handle_ct_t` refers two unsigned 64-bit integral type variables that are allocated through a `PtlCTAlloc()`, queried through a `PtlCTGet()`, `PtlCTWait()`, or `PtlCTPoll()`, set through a `PtlCTSet()`, incremented through a `PtlCTInc()`, and freed through a `PtlCTFree()`. To mirror the failure semantics of the full events, one variable counts the successful events and the second variable counts the events that failed.

**IMPLEMENTATION NOTE 24:**

A high performance implementation could choose to make a `ptl_handle_ct_t` a simple pointer to a structure in the address space of the application; however, in some cases, it may be desirable, or even necessary, to allocate these pointers in a special part of the address space (e.g. low physical addresses to facilitate accesses by particular hardware).

Semantics for event occurrence match those described in Sections 3.13.2. They can be independently enabled/disabled with options on the memory descriptor or match list entry analogous to those used for event queues.
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 associated with failed events that counts events or bytes.

3.14.2 PtlCTAlloc

The `PtlCTAlloc()` function is used to allocate a counting event that counts either operations on the memory descriptor (match list entry) or bytes that flow out of (into) a memory descriptor (match list entry). While a `PtlCTAlloc()` call could be as simple as a malloc of a structure holding the counting event and a network interface handle, 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; thus, an allocation routine is provided. A newly allocated count is 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.
IMPLEMENTATION NOTE 25: Minimizing cost of counting events
A quality implementation will attempt to minimize the cost of counting events. This can be done by translating the simple functions (PtlCTGet(), PtlCTWait(), PtlCTSet(), and PtlCTInc()) into simple macros that directly access a structure in the applications memory unless otherwise required by the hardware.

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 once 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

```c
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 destroyed. The operations are not triggered, and will not modify any application-visible state. The other state associated with `ct_handle` is left unchanged.

Function Prototype for PtlCTCancelTriggered

```c
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. Access made by `PtlCTGet()` are not atomic relative to modifications made by the `PtlCTSet()` and `PtlCTInc()` functions in a separate thread. Calling `PtlCTFree()` in a separate thread while `PtlCTGet()` is executing may yield undefined results in the returned value.

**Function Prototype for PtlCTGet**

```c
int PtlCTGet(ptl_handle_ct_t ct_handle,
             pty_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 `pty_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.

**Discussion:** `PtlCTGet()` must be as close to the speed of a simple variable access as possible; hence, `PtlCTGet()` is not atomic relative to `PtlCTSet()` or `PtlCTInc()` operations that occur in a separate thread and is undefined if `PtlCTFree()` or `PtlNIFini()` is called during the execution of the function.

### 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 a test value or when the failure field is non-zero. All processes that are waiting on a single counting event with a given test value will return from `PtlCTWait()` when that test is reached.
Function Prototype for PtlCTWait

```c
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 Section 3.13.7 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() is sufficiently general to implement both PtlCTGet() and PtlCTWait(), but these functions have been retained in the API, since the can be implemented in a substantially lighter weight manner.

**IMPLEMENTATION NOTE 26:**

Fairness of PtlCTPoll()

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

```c
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 `ctHandles` 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 PtlNIFin() was called by another thread while this thread was waiting in PtlCTPoll(). See Section 3.13.7 for more information.

**IMPLEMENTATION NOTE 27:**

Macros using PtlCTPoll()

Implementations are discouraged from providing macros for PtlCTGet() and PtlCTWait() that use PtlCTPoll() instead of providing these functions. The usage scenario for PtlCTGet() and PtlCTWait() is expected to depend on minimizing the computational cost of these routines.

### 3.14.8 PtlCTSet

Periodically, it is desirable to reinitialize or adjust the value of a counting event. This must be done atomically relative to other modifications, so a functional interface is provided. The PtlCTSet() function is used to set the value of a counting event. The entire ptl_ct_event_t is updated atomically relative to other modifications of the counting
event; however, it is not atomic relative to read accesses of the counting event that are made from another thread. The results of a PtlCTSet() operation are visible to subsequent accesses by PtlCTGet() within the same thread.

**Function Prototype for PtlCTSet**

```c
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

In some scenarios, the counting event will need to be incremented by the application. This must be done atomically relative to other modifications of the counting event, so a functional interface is provided. The PtlCTInc() function is used to increment the value of a counting event. The entire `ptl_ct_event_t` is updated atomically relative to other modifications of the counting event; however, it is not atomic relative to read accesses of the counting event from a different thread. The results of a PtlCTSet() operation are visible to subsequent accesses by PtlCTGet() within the same thread. The `increment` field can only be non-zero for either the success or failure field in a given call to PtlCTInc().

**Discussion:** As an example, a counting event may need to be incremented at the completion of a message that is received. If the message arrives in the overflow list, it may be desirable to delay the counting event increment until the application can place the data in the correct buffer.

**Function Prototype for PtlCTInc**

```c
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()`.

**IMPLEMENTATION NOTE 28:** Functions that require communication

Other than `PtlPut()`, `PtlGet()`, `PtlAtomic()`, `PtlFetchAtomic()`, and `PtlSwap()` (and their triggered variants), no function in the portals API requires communication with other nodes in the system.

### 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 modified 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`)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PTL_ACK_REQ</code></td>
<td>An acknowledgement capable of generating both a full event and counting event is requested.</td>
</tr>
<tr>
<td><code>PTL_CT_ACK_REQ</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>PTL_OC_ACK_REQ</code></td>
<td>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 <em>target</em>. <code>PTL_OC_ACK_REQ</code> does not support the <code>PTL_MD_EVENT_CT_BYTES</code> 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.</td>
</tr>
<tr>
<td><code>PTL_NO_ACK_REQ</code></td>
<td>No acknowledgement is requested.</td>
</tr>
</tbody>
</table>

|
**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).

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**

```c
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. Allowed constants: PTL_ACK_REQ, PTL_NO_ACK_REQ, PTL_CT_ACK_REQ, PTL_OC_ACK_REQ.

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.\footnote{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.}

hdr_data | input | 64 bits of user data that can be included in the message header. This data is written to an event queue entry at the target if an event queue is present on the match list entry that matches the message.

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.

3.15.3 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

```c
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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>md_handle</td>
<td>input</td>
<td>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.</td>
</tr>
<tr>
<td>local_offset</td>
<td>input</td>
<td>Offset from the start of the memory descriptor.</td>
</tr>
<tr>
<td>length</td>
<td>input</td>
<td>Length of the memory region for the reply.</td>
</tr>
<tr>
<td>target_id</td>
<td>input</td>
<td>A process identifier for the target process.</td>
</tr>
<tr>
<td>pt_index</td>
<td>input</td>
<td>The index in the target portal table.</td>
</tr>
<tr>
<td>match_bits</td>
<td>input</td>
<td>The match bits to use for message selection at the target process.</td>
</tr>
<tr>
<td>remote_offset</td>
<td>input</td>
<td>The offset into the target match list entry (used unless the target match list entry has the PTL_ME_MANAGE_LOCAL option set).</td>
</tr>
<tr>
<td>user_ptr</td>
<td>input</td>
<td>See the discussion for PtlPut().</td>
</tr>
</tbody>
</table>

Return Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_OK</td>
<td>Indicates success.</td>
</tr>
<tr>
<td>PTL_NO_INIT</td>
<td>Indicates that the portals API has not been successfully initialized.</td>
</tr>
<tr>
<td>PTL_ARG_INVALID</td>
<td>Indicates that an invalid argument was passed. The definition of which arguments are checked is implementation dependent.</td>
</tr>
</tbody>
</table>

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 passed in the put memory descriptor. The PtlFetchAtomic() function extends PtlAtomic() to be an atomic fetch-and-update operation; thus, 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 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 in all implementations.

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. All three atomic functions are required to be natively aligned at the target to the size of the ptl_datatype_t used.

**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 swap under mask. Update the bits of the target value that are set to 1 in the operand and 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>
<thead>
<tr>
<th>Integral Types</th>
<th>Floating-Point Types</th>
<th>Complex Types</th>
<th>PtlAtomic()</th>
<th>PtlFetchAtomic()</th>
<th>PtlSwap()</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_MIN</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_MAX</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_SUM</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_PROD</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_LOR</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_BAND</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_SWAP</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_CSWAP</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_CSWAP_NE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_CSWAP_LE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_CSWAP_LT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_CSWAP_GE</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_CSWAP_GT</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>PTL_MSWAP</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

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

```c
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. Allowed constants: PTL_ACK_REQ, PTL_NO_ACK_REQ, PTL_CT_ACK_REQ, PTL_OC_ACK_REQ.

- 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. 64 bits of user data that can be included in the message header. This data is written to an event queue entry at the target if an event queue is present on the match list entry that the message matches.

- 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. This means that both PtlPut() and PtlGet() style events can be delivered. 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. Note that receiving a PTL_EVENT_REPLY inherently implies that the flow control check has passed on the target node. In addition, 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**

```c
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: 64 bits of user data that can be included in the message header. This data is written to an event queue entry at the target if an event queue is present on the match list entry that the message matches.

- **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. In addition, 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

```c
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.
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.

Offset from the start of the memory descriptor referenced by the put_md_handle to use for transmitted data.

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.

A process identifier for the target process.

The index in the target portal table.

The match bits to use for message selection at the target process.

The offset into the target memory region (used unless the target match list entry has the PTL_ME_MANAGE_LOCAL option set).

See the discussion for PtlPut().

64 bits of user data that can be included in the message header. This data is written to an event queue entry at the target if an event queue is present on the match list entry that the message matches.

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.

The operation to be performed using the initiator and target data.

The type of data being operated on at the initiator and target.

Indicates success.

Indicates that the portals API has not been successfully initialized.

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

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 29:**

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**

```c
int PtlAtomicSync();
```

**Return Codes**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_OK</td>
<td>Indicates success.</td>
</tr>
<tr>
<td>PTL_NO_INIT</td>
<td>Indicates that the portals API has not been successfully initialized.</td>
</tr>
</tbody>
</table>

---

### 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()`.

**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.

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.

---

**IMPLEMENTATION NOTE 30:**

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.
IMPLEMENTATION NOTE 31: Implementation of Triggered Operations

The most straightforward way to implement triggered operations is to associate a list of dependent operations with the structure referenced by a trig_ct_handle. Operations depending on the same trig_ct_handle with the same threshold should proceed in the order that they were issued; thus, the list of operations associated with a trig_ct_handle may be sorted for faster searching.

IMPLEMENTATION NOTE 32: Triggered Operations Reaching the Threshold

The triggered operation is released when the counter referenced by the trig_ct_handle reaches or exceeds the threshold. This means that the triggered operation must check the value of the trig_ct_handle in an atomic way when it is first associated with the trig_ct_handle.

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

```c
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`, `local_offset`, `length`, `ack_req`, `target_id`, `pt_index`, `match_bits`, `remote_offset`, `user_ptr`, `hdr_data`, `trig_ct_handle`, `threshold`

  - **input** See description in Section 3.15.2.
  - **input** Handle used for triggering the operation.
  - **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

```c
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**, **target_id**, **pt_index**, **match_bits**, **user_ptr**, **remote_offset**, **local_offset**, **length**
  - **input**: See the discussion for **PtlGet**.

- **trig_ct_handle**, **threshold**
  - **input**: Handle used for triggering the operation.
  - **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 triggered atomic operations extend the Portals atomic operations (PtlAtomic(), PtlFetchAtomic(), and PtlSwap()) with the triggered operation semantics. 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

```c
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`, `local_offset`, `length`, `ack_req`, `target_id`, `pt_index`, `match_bits`, `remote_offset`, `user_ptr`, `hdr_data`, `operation`, `datatype` input  See the discussion of PtlAtomic().

- `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

Function Prototype for PtlTriggeredFetchAtomic

```c
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, local_get_offset, put_md_handle, local_put_offset, length, target_id, pt_index, match_bits, remote_offset, user_ptr, hdr_data, operation, datatype

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

Function Prototype for PtlTriggeredSwap

```c
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,
                      ptt_op_t operation,
                      ptt_datatype_t datatype,
                      ptt_handle_ct_t trig_ct_handle,
                      ptt_size_t threshold);
```

Arguments

- `get_md_handle`, `local_get_offset`, `put_md_handle`, `local_put_offset`, `length`, `target_id`, `pt_index`, `match_bits`, `remote_offset`, `user_ptr`, `hdr_data`, `operand`, `operation`, `datatype` are input arguments.
  - See the discussion of `PtlSwap()`.
- `trig_ct_handle` is input.
  - Handle used for triggering the operation.
- `threshold` is 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**

```c
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 the discussion of **PtlCTInc()**.
- `increment` 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**

```c
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 the discussion of **PtlCTSet()**.
- `new_ct` input: Handle used for triggering the operation.
- `threshold` input: Threshold at which the operation triggers.
ct_handle, new_ct input See the discussion of PtlCTSet().

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

In many cases, the application has knowledge of its intended usage model that could be used to improve the performance of the implementation if there was a way to convey that knowledge. When an application informs the implementation of an intended usage model, it is binding on the application: the application cannot violate the usage model conveyed to the implementation. In contrast, such information is only a hint to the implementation: the implementation is not required to change its behavior based on the usage model the application describes. One prevalent usage model that many implementations could optimize for is a stream of operations in close temporal proximity. Informing the implementation of an impending stream of operations may allow it to optimize the conveyance of those operations through the messaging system (e.g. across the host bus or even across the network).

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

```c
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.

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 33:</th>
<th>Purpose of Bundling</th>
</tr>
</thead>
</table>
| The PtlStartBundle() and PtlEndBundle() interface was designed to allow the implementation to avoid unnecessary sfunce() memory barrier operations during periods that the application expects high message rate usage. 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 sfunce() operations with every message to have relatively deterministic latency. Between a PtlStartBundle() and PtlEndBundle(), the implementation might simply omit the sfunce() 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

```c
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

\[
\text{int PtlHandleIsEqual(\text{ptl\_handle\_any\_t handle1}, \\
\text{ptl\_handle\_any\_t handle2});}
\]

Arguments

\begin{itemize}
  \item \text{handle1, handle2} \quad \text{input} \quad \text{An object handle. Either of these handles is allowed to be the constant value PTL\_INVALID\_HANDLE, which represents the value of an invalid handle.}
\end{itemize}

Return Codes

\begin{itemize}
  \item \text{zero} \quad \text{Indicates that the two handles are not equivalent.}
  \item \text{non-zero} \quad \text{Indicates that the two handles are equivalent.}
\end{itemize}

Discussion: PtlHandleIsEqual() returns a value suitable for direct evaluation in a conditional expression. While different from 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Sec</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptl_ack_req_t</td>
<td>acknowledgment request</td>
<td>3.15.2</td>
<td>PtlPut(), PtlAtomic(), PtlTriggeredPut(),</td>
</tr>
<tr>
<td></td>
<td>types</td>
<td></td>
<td>PtlTriggeredAtomic()</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Sec</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptl_ct_event_t</td>
<td>counting event structure</td>
<td>3.14.2</td>
<td>PtlCTAlloc()</td>
</tr>
<tr>
<td>ptl_datatype_t</td>
<td>datatype for atomic operation</td>
<td>3.15.4</td>
<td>PtlAtomic(), PtlFetchAtomic(), PtlSwap()</td>
</tr>
<tr>
<td>ptl_event_kind_t</td>
<td>event kind</td>
<td>3.13.1</td>
<td>PtlEQGet(), PtlEQWait(), PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_event_t</td>
<td>event queue entry</td>
<td>3.13.4</td>
<td>PtlEQGet(), PtlEQWait(), PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_handle_any_t</td>
<td>any object handles</td>
<td>3.3.2</td>
<td>PtlNIHandle(), PtlHandleIsEqual()</td>
</tr>
<tr>
<td>ptl_handle_eq_t</td>
<td>event queue handles</td>
<td>3.3.2</td>
<td>PtlEQAlloc(), PtlEQFree(), PtlEQGet(), PtlEQWait(), PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_handle_md_t</td>
<td>memory descriptor handles</td>
<td>3.3.2</td>
<td>PtlMDRelease(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetch Atomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()</td>
</tr>
<tr>
<td>ptl_handle_me_t</td>
<td>match list entry handles</td>
<td>3.3.2</td>
<td>PtlMEAppend(), PtlMESearch(), PtlMEUnlink()</td>
</tr>
<tr>
<td>ptl_handle_ni_t</td>
<td>network interface handles</td>
<td>3.3.2</td>
<td>PtlNIInit(), PtlNIFini(), PtlNIStatus(), PtlEQAlloc()</td>
</tr>
<tr>
<td>ptl_hdr_data_t</td>
<td>user header data</td>
<td>3.15.2</td>
<td>PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()</td>
</tr>
<tr>
<td>ptl_interface_t</td>
<td>network interface identifiers</td>
<td>3.3.5</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>ptl_list_t</td>
<td>type of list attached to a portal table entry</td>
<td>3.12.2</td>
<td>PtlMEAppend()</td>
</tr>
<tr>
<td>ptl_match_bits_t</td>
<td>match (and ignore) bits</td>
<td>3.3.4</td>
<td>PtlMEAppend(), PtlMESearch(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()</td>
</tr>
<tr>
<td>ptl_iovec_t</td>
<td>scatter/gather buffer descriptors</td>
<td>3.10.2</td>
<td>PtlMEAppend(), PtlMDBind(), PtlMDRelease()</td>
</tr>
<tr>
<td>ptl_md_t</td>
<td>memory descriptors</td>
<td>3.10.1</td>
<td>PtlMDRelease(), PtlMDBind()</td>
</tr>
<tr>
<td>ptl_me_t</td>
<td>match list entries</td>
<td>3.12.1</td>
<td>PtlMEAppend(), PtlMESearch()</td>
</tr>
<tr>
<td>ptl_nid_t</td>
<td>node identifiers</td>
<td>3.3.6</td>
<td>PtlGetId(), PtlGetPhysId()</td>
</tr>
<tr>
<td>ptl_ni_fail_t</td>
<td>network interface specific failures</td>
<td>3.13.3</td>
<td>PtlEQGet(), PtlEQWait(), PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_ni_limits_t</td>
<td>implementation dependent limits</td>
<td>3.6.1</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>ptl_op_t</td>
<td>atomic operation type</td>
<td>3.15.4</td>
<td>PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggered Atomic(), Ptl TriggeredFetch Atomic(), PtlTriggeredSwap()</td>
</tr>
<tr>
<td>ptl_pid_t</td>
<td>process identifier</td>
<td>3.3.6</td>
<td>PtlGetId(), PtlGetPhysId</td>
</tr>
</tbody>
</table>

continued on next page
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PtlCTAlloc()</td>
<td>create a counting event</td>
<td>3.14.2</td>
</tr>
<tr>
<td>PtlICTCancelTriggered()</td>
<td>cancel pending triggered operations</td>
<td>3.14.4</td>
</tr>
<tr>
<td>PtlICTFree()</td>
<td>free a counting event</td>
<td>3.14.3</td>
</tr>
<tr>
<td>PtlICTInc()</td>
<td>increment a counting event by a certain value</td>
<td>3.14.9</td>
</tr>
<tr>
<td>PtlICTGet()</td>
<td>get the current value of a counting event</td>
<td>3.14.5</td>
</tr>
<tr>
<td>PtlICTPoll()</td>
<td>wait for an array of counting events to reach certain values</td>
<td>3.14.7</td>
</tr>
<tr>
<td>PtlICTWait()</td>
<td>wait for a counting event to reach a certain value</td>
<td>3.14.6</td>
</tr>
<tr>
<td>PtlICTSet()</td>
<td>set a counting event to a certain value</td>
<td>3.14.8</td>
</tr>
<tr>
<td>PtlIEQAlloc()</td>
<td>create an event queue</td>
<td>3.13.5</td>
</tr>
<tr>
<td>PtlIEQFree()</td>
<td>release the resources for an event queue</td>
<td>3.13.6</td>
</tr>
<tr>
<td>PtlIEQGet()</td>
<td>get the next event from an event queue</td>
<td>3.13.7</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_ARG_INVALID</td>
<td>invalid argument passed</td>
<td>PtlCTFree(), PtlCTGet(), PtlCTWait()</td>
</tr>
<tr>
<td>PTL_CT_NONE_REACHED</td>
<td>timeout reached before any counting event reached the test</td>
<td>PtlCTPoll()</td>
</tr>
</tbody>
</table>

continued on next page
### Table 3.8. Portals Constants: Other Constants Defined by the Portals API.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Base Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_ACK_REQ</td>
<td>request an acknowledgment</td>
<td>ptl_ack_req_t</td>
<td>3.15, 3.15.2</td>
</tr>
<tr>
<td>PTL_CT_ACK_REQ</td>
<td>request a counting acknowledgment completed</td>
<td>ptl_ack_req_t</td>
<td>3.15, 3.15.2</td>
</tr>
<tr>
<td>PTL_OC_ACK_REQ</td>
<td>request an operation completed</td>
<td>ptl_ack_req_t</td>
<td>3.15, 3.15.2</td>
</tr>
<tr>
<td>PTL_CT_NONE</td>
<td>a NULL count handle</td>
<td>ptl_handle_ct_t</td>
<td>3.3.2, 3.10.1</td>
</tr>
<tr>
<td>PTL_EQ_NONE</td>
<td>a NULL event queue handle</td>
<td>ptl_handle_eq_t</td>
<td>3.3.2, 3.10.1</td>
</tr>
<tr>
<td>PTL_EVENT_ACK</td>
<td>acknowledgment event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.2</td>
</tr>
<tr>
<td>PTL_EVENT_GET</td>
<td>get event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.3</td>
</tr>
<tr>
<td>PTL_EVENT_GET_OVERFLOW</td>
<td>get overflow event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.3</td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC</td>
<td>atomic event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.5</td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC_OVERFLOW</td>
<td>atomic overflow event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.3</td>
</tr>
<tr>
<td>PTL_EVENT_FETCH_ATOMIC</td>
<td>fetching atomic event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.5</td>
</tr>
<tr>
<td>PTL_EVENT_FETCH_ATOMIC_OVERFLOW</td>
<td>fetching atomic overflow event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.3</td>
</tr>
<tr>
<td>PTL_EVENT_PUT</td>
<td>put event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.2</td>
</tr>
<tr>
<td>PTL_EVENT_PUT_OVERFLOW</td>
<td>put overflow event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.2</td>
</tr>
</tbody>
</table>

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 sections in which the constant is mentioned. (A boldface section indicates the place the constant is introduced or described.)
<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Base Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_EVENT_REPLY</td>
<td>reply event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.3,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.15.5</td>
</tr>
<tr>
<td>PTL_EVENT_SEND</td>
<td>send event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.15.2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.15.5</td>
</tr>
<tr>
<td>PTL_EVENT_AUTO_UNLINK</td>
<td>automatic unlink event</td>
<td>ptl_event_kind_t</td>
<td>3.12.1, 3.12.3,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.13.1</td>
</tr>
<tr>
<td>PTL_EVENT_AUTO_FREE</td>
<td>automatic free event</td>
<td>ptl_event_kind_t</td>
<td>3.12.1, 3.12.3,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.13.1</td>
</tr>
<tr>
<td>PTL_EVENT_PT_DISABLED</td>
<td>portal table entry disabled event</td>
<td>ptl_event_kind_t</td>
<td>3.13.1, 3.12.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.6</td>
</tr>
<tr>
<td>PTL_EVENT_SEARCH</td>
<td>search event</td>
<td>ptl_event_kind_t</td>
<td>3.12.1, 3.12.3,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.13.1</td>
</tr>
<tr>
<td>PTL_IFACE_DEFAULT</td>
<td>default interface</td>
<td>ptl_interface_t</td>
<td>3.3.5</td>
</tr>
<tr>
<td>PTL_INVALID_HANDLE</td>
<td>invalid handle</td>
<td>ptl_handle_any_t</td>
<td>3.3.2, 3.18.1</td>
</tr>
<tr>
<td>PTL_PRIORITY_LIST</td>
<td>specifies the priority list attached to a portal table entry</td>
<td>int</td>
<td>3.12.2</td>
</tr>
<tr>
<td>PTL_MD_EVENT_SUCCESS_DISABLE</td>
<td>a flag to disable events that indicate success</td>
<td>int</td>
<td>3.10.1</td>
</tr>
<tr>
<td>PTL_LE_ACK_DISABLE</td>
<td>a flag to disable acknowledgments</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_TARGET_BIND_INACCESSIBLE</td>
<td>A flag to indicate that the implementation should allow LEs to be bound over ranges of memory that are not allocated</td>
<td>int</td>
<td>3.6.1</td>
</tr>
<tr>
<td>PTL_TOTAL_DATA_ORDERING</td>
<td>A flag to indicate that the implementation should attempt to provide total data ordering</td>
<td>int</td>
<td>3.6.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_COMM_DISABLE</td>
<td>a flag to disable events associated with new communications</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_FLOWCTRL_DISABLE</td>
<td>a flag to disable events associated with flow control</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_SUCCESS_DISABLE</td>
<td>a flag to disable events that indicate success</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_CT_COMM</td>
<td>a flag to count communication events</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_CT_OVERFLOW</td>
<td>a flag to count overflow events</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_CT_BYTES</td>
<td>a flag to count bytes instead of operations</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_UNLINK_DISABLE</td>
<td>a flag to disable unlink events</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_IS_ACCESSIBLE</td>
<td>a flag to indicate the entire LE is accessible</td>
<td>int</td>
<td>3.11.1, 4.2.2</td>
</tr>
<tr>
<td>PTL_LE_OP_GET</td>
<td>a flag to enable get operations</td>
<td>int</td>
<td>3.11.1, 4.2.2</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Base Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_LE_OP_PUT</td>
<td>a flag to enable put operations</td>
<td>int</td>
<td>3.11.1, 4.2.2</td>
</tr>
<tr>
<td>PTL_LE_USE_ONCE</td>
<td>a flag to indicate that the list entry will only be used once</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_ME_ACK_DISABLE</td>
<td>a flag to disable acknowledgments</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_COMM_DISABLE</td>
<td>a flag to disable events associated with new communications</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_FLOWCTRL_DISABLE</td>
<td>a flag to disable events associated with flow control</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_SUCCESS_DISABLE</td>
<td>a flag to disable events that indicate success</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_CT_COMM</td>
<td>a flag to count communication events</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_CT_OVERFLOW</td>
<td>a flag to count overflow events</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_CT_BYTES</td>
<td>a flag to count bytes instead of operations</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_MD_EVENT_CT_SEND</td>
<td>a flag to count send events</td>
<td>int</td>
<td>3.10.1</td>
</tr>
<tr>
<td>PTL_MD_EVENT_CT_REPLY</td>
<td>a flag to count reply events</td>
<td>int</td>
<td>3.10.1</td>
</tr>
<tr>
<td>PTL_MD_EVENT_CT_ACK</td>
<td>a flag to count acknowledgment events</td>
<td>int</td>
<td>3.10.1</td>
</tr>
<tr>
<td>PTL_MD_EVENT_CT_BYTES</td>
<td>a flag to count bytes instead of operations</td>
<td>int</td>
<td>3.10.1</td>
</tr>
<tr>
<td>PTL_MD_UNORDERED</td>
<td>a flag to indicate that messages from this MD do not need to be ordered</td>
<td>int</td>
<td>3.10.1</td>
</tr>
<tr>
<td>PTL_MD_VOLATILE</td>
<td>a flag to indicate that the application will modify the put buffer immediately upon operation return, before receiving a send event.</td>
<td>int</td>
<td>3.10.1</td>
</tr>
<tr>
<td>PTL_IOVEC</td>
<td>a flag to enable scatter/gather memory descriptors</td>
<td>int</td>
<td>3.12.1, 3.10.2</td>
</tr>
<tr>
<td>PTL_ME_EVENT_UNLINK_DISABLE</td>
<td>a flag to disable unlink events</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_IS_ACCESSIBLE</td>
<td>a flag to indicate the entire ME is accessible</td>
<td>int</td>
<td>3.12.1, 4.2.2</td>
</tr>
<tr>
<td>PTL_ME_MANAGE_LOCAL</td>
<td>a flag to enable the use of local offsets</td>
<td>int</td>
<td>3.12.1, 3.15.2, 3.15.3</td>
</tr>
<tr>
<td>PTL_ME_OP_GET</td>
<td>a flag to enable get operations</td>
<td>int</td>
<td>3.12.1, 4.2.2</td>
</tr>
<tr>
<td>PTL_ME_OP_PUT</td>
<td>a flag to enable put operations</td>
<td>int</td>
<td>3.12.1, 4.2.2</td>
</tr>
<tr>
<td>PTL_ME_NO_TRUNCATE</td>
<td>a flag to disable truncation of a request</td>
<td>int</td>
<td>3.12.1, 4.2.2</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Base Type</td>
<td>Reference</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>----------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>PTL_ME_USE_ONCE</td>
<td>a flag to indicate that the match list entry will only be used once</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_MAY_ALIGN</td>
<td>a flag to indicate that the implementation may align an incoming message to a natural boundary to enhance performance</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_NID_ANY</td>
<td>wildcard for node identifier fields</td>
<td>ptl_nid_t</td>
<td>3.3.6, 3.12.2, 3.12</td>
</tr>
<tr>
<td>PTL_NI_OK</td>
<td>successful event</td>
<td>ptl_ni_fail_t</td>
<td>3.13.3, 3.13.4</td>
</tr>
<tr>
<td>PTL_NI_UNDELIVERABLE</td>
<td>message could not be delivered</td>
<td>ptl_ni_fail_t</td>
<td>3.13.3, 3.13.4</td>
</tr>
<tr>
<td>PTL_NI_DROPPED</td>
<td>message was dropped</td>
<td>ptl_ni_fail_t</td>
<td>3.13.3, 3.13.4</td>
</tr>
<tr>
<td>PTL_NI_SEGV</td>
<td>message attempted to access inaccessible memory</td>
<td>ptl_ni_fail_t</td>
<td>3.13.3, 3.13.4</td>
</tr>
<tr>
<td>PTL_NI_PT_DISABLED</td>
<td>message encountered a disabled portal table entry</td>
<td>ptl_ni_fail_t</td>
<td>3.13.3, 3.13.4</td>
</tr>
<tr>
<td>PTL_NI_NO_MATCH</td>
<td>search did not find an entry in the unexpected list</td>
<td>ptl_ni_fail_t</td>
<td>3.13, 3.12.4, 3.11.4</td>
</tr>
<tr>
<td>PTL_NI_OP_ViolATION</td>
<td>message encountered an operation violation</td>
<td>ptl_ni_fail_t</td>
<td>3.13.3, 3.13.4</td>
</tr>
<tr>
<td>PTL_NI_PERM_ViolATION</td>
<td>message encountered a permissions violation</td>
<td>ptl_ni_fail_t</td>
<td>3.13.3, 3.13.4</td>
</tr>
<tr>
<td>PTL_NI_MATCHING</td>
<td>a flag to indicate that the network interface must provide matching portals addressing</td>
<td>int</td>
<td>3.6.2</td>
</tr>
<tr>
<td>PTL_NI_NO_MATCHING</td>
<td>a flag to indicate that the network interface must provide non-matching portals addressing</td>
<td>int</td>
<td>3.6.2</td>
</tr>
<tr>
<td>PTL_NI_LOGICAL</td>
<td>a flag to indicate that the network interface must provide logical addresses for network endpoints</td>
<td>int</td>
<td>3.6.2</td>
</tr>
<tr>
<td>PTL_NI_PHYSICAL</td>
<td>a flag to indicate that the network interface must provide physical addresses for network endpoints</td>
<td>int</td>
<td>3.6.2</td>
</tr>
<tr>
<td>PTL_NO_ACK_REQ</td>
<td>request no acknowledgment</td>
<td>ptl_ack_req_t</td>
<td>3.15, 3.15.2, 4.2.1</td>
</tr>
<tr>
<td>PTL_OVERFLOW_LIST</td>
<td>specifies the overflow list attached to a portal table entry</td>
<td>int</td>
<td>3.12.2</td>
</tr>
<tr>
<td>PTL_PID_ANY</td>
<td>wildcard for process identifier fields</td>
<td>ptl_pid_t</td>
<td>3.3.6, 3.6.2, 3.12.2, 3.12</td>
</tr>
<tr>
<td>PTL_PID_MAX</td>
<td>Maximum legal process identifier</td>
<td>ptl_pid_t</td>
<td>3.6.2</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Base Type</td>
<td>Reference</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>PTL_PT_ANY</td>
<td>wildcard for portal table entry identifier fields</td>
<td>ptl_pt_index_t</td>
<td>3.7.1</td>
</tr>
<tr>
<td>PTL_PT_FLOWCTRL</td>
<td>a flag to request flow control</td>
<td>int</td>
<td>3.7.1</td>
</tr>
<tr>
<td>PTL_PT_ONLY_USE_ONCE</td>
<td>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</td>
<td>int</td>
<td>3.7.1</td>
</tr>
<tr>
<td>PTL_PT_ONLY_TRUNCATE</td>
<td>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</td>
<td>int</td>
<td>3.7.1</td>
</tr>
<tr>
<td>PTL_SEARCH_ONLY</td>
<td>specifies that the unexpected list should only be searched</td>
<td>int</td>
<td>3.12.4</td>
</tr>
<tr>
<td>PTL_SEARCH_DELETE</td>
<td>specifies that the unexpected list should be searched and the matching item should be deleted</td>
<td>int</td>
<td>3.12.4</td>
</tr>
<tr>
<td>PTL_RANK_ANY</td>
<td>wildcard for rank fields</td>
<td>ptl_rank_t</td>
<td>3.3.6, 3.12.2, 3.12</td>
</tr>
<tr>
<td>PTL_SR_DROP_COUNT</td>
<td>index for the dropped count register</td>
<td>ptl_sr_index_t</td>
<td>3.3.7, 3.6.4</td>
</tr>
<tr>
<td>PTL_SR_OPERATIONS_VIOLATIONS</td>
<td>index for the operation violations register</td>
<td>ptl_sr_index_t</td>
<td>3.3.7, 3.6.4</td>
</tr>
<tr>
<td>PTL_SR_PERMISSIONS_VIOLATIONS</td>
<td>index for the permission violations register</td>
<td>ptl_sr_index_t</td>
<td>3.3.7, 3.6.4</td>
</tr>
<tr>
<td>PTL_TIME_FOREVER</td>
<td>a flag to indicate unbounded time</td>
<td>ptl_time_t</td>
<td>3.13.9</td>
</tr>
<tr>
<td>PTL_UID_ANY</td>
<td>wildcard for user identifier</td>
<td>ptl_uid_t</td>
<td>3.3.6, 3.12.2, 3.11, 3.12</td>
</tr>
</tbody>
</table>
Chapter 4

Guide to Implementation

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 its 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 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().

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

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 94.)

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.

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

Table 4.3. **Acknowledgment**: Information Passed in a “Counting” Acknowledgment.

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

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().

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

<table>
<thead>
<tr>
<th>Information</th>
<th>Type</th>
<th>PtlGet() Argument</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td>get_md_handle</td>
<td>indicates an reply</td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td>options field from NI associated with MD options</td>
<td></td>
</tr>
<tr>
<td>initiator</td>
<td>ptr_process_t</td>
<td>target_id</td>
<td>local information on get target</td>
</tr>
<tr>
<td>target</td>
<td>ptr_process_t</td>
<td>initiator</td>
<td>echo initiator of get</td>
</tr>
<tr>
<td>memory descriptor</td>
<td>ptr_handle_md_t</td>
<td>md_handle</td>
<td>echo md_handle of get</td>
</tr>
<tr>
<td>get user pointer</td>
<td>void *</td>
<td>local_offset</td>
<td>echo local_offset of get</td>
</tr>
<tr>
<td>manipulated length</td>
<td>ptr_size_t</td>
<td>user_ptr</td>
<td>echo user_ptr of get</td>
</tr>
<tr>
<td>offset</td>
<td>ptr_size_t</td>
<td>remote_offset</td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>matched list</td>
<td>ptr_list_t</td>
<td></td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>data</td>
<td>bytes</td>
<td></td>
<td>obtained from the operation</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Information</th>
<th>Type</th>
<th>PtlAtomic() Argument</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td>put_md_handle</td>
<td>indicates the type of atomic operation and datatype</td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td>put_md_handle</td>
<td>options field from NI associated with MD options</td>
</tr>
<tr>
<td>ack type</td>
<td>ptrl_ack_req_t</td>
<td>ack_req</td>
<td></td>
</tr>
<tr>
<td>initiator</td>
<td>ptrl_process_t</td>
<td>target_id</td>
<td>local information</td>
</tr>
<tr>
<td>user</td>
<td>ptrl_uid_t</td>
<td></td>
<td>local information</td>
</tr>
<tr>
<td>target</td>
<td>ptrl_process_t</td>
<td>pt_index</td>
<td></td>
</tr>
<tr>
<td>portal index</td>
<td>ptrl_pt_index_t</td>
<td>put_md_handle</td>
<td>opt. if ack_req = PTL_NO_ACK_REQ</td>
</tr>
<tr>
<td>memory descriptor</td>
<td>ptrl_handle_md_t</td>
<td>user_ptr</td>
<td>opt. if ack_req = PTL_NO_ACK_REQ or ack_req = PTL_CT_ACK_REQ or ack_req = PTL_OC_ACK_REQ</td>
</tr>
<tr>
<td>user pointer</td>
<td>void *</td>
<td>match_bits</td>
<td>optional if the PTL_NI_NO_MATCHING option is set.</td>
</tr>
<tr>
<td>match bits</td>
<td>ptrl_match_bits_t</td>
<td>match_bits</td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td>ptr_size_t</td>
<td>remote_offset</td>
<td></td>
</tr>
<tr>
<td>memory descriptor</td>
<td>ptrl_handle_md_t</td>
<td>get_md_handle</td>
<td>destination of reply</td>
</tr>
<tr>
<td>length</td>
<td>ptr_size_t</td>
<td>put_md_handle</td>
<td>length member</td>
</tr>
<tr>
<td>operand</td>
<td>bytes</td>
<td>operand</td>
<td>Used in CSWAP and MSWAP operations</td>
</tr>
<tr>
<td>data</td>
<td>bytes</td>
<td>put_md_handle</td>
<td>user data</td>
</tr>
</tbody>
</table>
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 NI.

Also see Sections 2.3 and Figure 2.10.

4.3 Event Generation and Error Reporting
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.

<table>
<thead>
<tr>
<th>Target ME/LE Flags</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>put</td>
</tr>
<tr>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>PTL_ME_OP_PUT/PTL_LE_OP_PUT</td>
<td>•</td>
</tr>
<tr>
<td>PTL_ME_OP_GET/PTL_LE_OP_GET</td>
<td>-</td>
</tr>
<tr>
<td>both</td>
<td>•</td>
</tr>
</tbody>
</table>
References


Appendix A

Portals Design Guidelines

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

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

Virtual Interface Architecture. It should be possible to write an efficient implementation of Portals based on the Virtual Interface Architecture (VIA).
Internetwork consistency. Portals should not impose any consistency requirements across multiple networks/interfaces. In particular, there will not be any memory consistency/coherency requirements when messages arrive on independent paths.

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

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

One exception to this is if a non-native implementation would suffer in scalability or take a large performance penalty.
Appendix 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_PERMISSIONS_VIOLATIONS`, and `PTL_SR_OPERATIONS_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

The most recent version of this document described Portals version 3.3 [19]. 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 initiator and target events were separated into different types with different accessor functions.

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.
PTL_LAND ............................... 96, 98
PTL_LE_ACK_DISABLE ............... 59, 117
PTL_LE_EVENT_COMM_DISABLE .... 59, 117
PTL_LE_EVENT_CT_BYTES .......... 60, 117
PTL_LE_EVENT_CT_COMM .......... 60, 117
PTL_LE_EVENT_CT_OVERFLOW ..... 60, 117
PTL_LE_EVENT_FLOWCTRL_DISABLE 59, 60, 117
PTL_LE_EVENT_LINK_DISABLE ...... 59
PTL_LE_EVENT_OVER_DISABLE ..... 60
PTL_LE_EVENT_SUCCESS_DISABLE .. 59, 117
PTL_LE_EVENT_UNLINK_DISABLE ... 60, 117
PTL_LE_OP_GET .................. 59, 117, 126, 135
PTL_LE_OP_PUT .................. 58, 118, 126, 135
PTL_LE_UNEXPECTED_HDR_DISABLE 59
PTL_LE_USE_ONCE . 47, 57, 59, 60, 74, 76, 118, 120
PTL_LONG_DOUBLE . 96, 97
PTL_LONG_DOUBLE_COMPLEX ....... 96, 97
PTL_LOR ............................ 96, 98
PTL_LXOR ....................... 97, 98
PTL_MAX ............................. 96, 98
PTL_MD_EVENT_CT_ACK ............. 53, 118
PTL_MD_EVENT_CT_BYTES .......... 53, 92, 118
PTL_MD_EVENT_CT_REPLY .......... 53, 118
PTL_MD_EVENT_CT_SEND ............ 53, 118
PTL_MD_EVENT_SUCCESS_DISABLE .. 53, 117
PTL_MD_UNORDERED .......... 29, 30, 53, 118
PTL_MD_VOLATILE ................ 41, 54, 118
PTL_ME_ACK_DISABLE ............... 67, 118
PTL_ME_EVENT_COMM_DISABLE ..... 67, 75, 118
PTL_ME_EVENT_CTBYTES .......... 68, 118
PTL_ME_EVENT_CT_COMM .......... 68, 118
PTL_ME_EVENT_CT_OVERFLOW ...... 68, 118
PTL_ME_EVENT_FLOWCTRL_DISABLE .. 67, 68, 118
PTL_ME_EVENT_LINK_DISABLE ...... 67
PTL_ME_EVENT_OVER_DISABLE .... 68
PTL_ME_EVENT_SUCCESS_DISABLE .. 68, 118
PTL_ME_EVENT_UNLINK_DISABLE .. 68, 75, 118
PTL_ME_IS_ACCESSIBLE .......... 64, 67, 118
PTL_ME_MANAGE_LOCAL . 66, 67, 94, 95, 99, 100, 102, 118, 122
PTL_ME_MAY_ALIGN ................ 67, 119
PTL_ME_NO_TRUNCATE . 27, 47, 67, 118, 120, 125
PTL_ME_OP_GET ................. 66, 96, 118, 125, 126
PTL_ME_OP_PUT ................. 66, 96, 118, 125, 126
PTL_ME_UNEXPECTED_HDR_DISABLE .. 67
PTL_ME_USE_ONCE . 47, 57, 67–69, 74, 119, 120
PTL_MIN ............................ 96, 98
PTL_M_swap ...................... 95, 97, 98, 101, 102
PTL_NI_DROPPED .................. 76, 119

PTL_NI_LOGICAL .................. 37, 42, 65, 119
PTL_NI_MATCHING ................. 42, 119
PTL_NI_NO_MATCH ............... 63, 71, 74, 119
PTL_NI_NO_MATCHING ........... 42, 57, 65, 119, 122–124
PTL_NI_OK ...................... 63, 71, 74, 76, 79, 119
PTL_NI_OP_VIOLATION .......... 58, 59, 66, 77, 119
PTL_NI_PERM_VIOLATION ....... 58, 66, 77, 119
PTL_NI_PROCESS ................. 37, 42, 119
PTL_NI_PT_DISABLED .......... 31, 76, 119
PTL_NI_SEGV .................... 52, 56, 64, 119
PTL_NI_UNDELIVERABLE ...... 76, 77, 119
PTL_NICOM_ACCESSIBLE ........ 37, 68, 119
PTL_NO_ACK_REQ ... 92, 94, 99, 119, 122, 124
PTL_OC_ACK_REQ ... 92–94, 99, 116, 122, 124
PTL_OVERFLOW_LIST ............. 61, 69, 119
PTL_PID_ANY ..................... 37, 42, 68, 119
PTL_PID_MAX ........................ 42, 119
PTL_PRIORITY_LIST ............. 61, 69, 117
PTL_PROD .......................... 96, 98
PTL_PT_ANY ...................... 47, 120
PTL_PT_FLOWCTRL ................ 25, 47, 120
PTL_PT_ONLY_TRUNCATE ........ 47, 120
PTL_PT_ONLY_USE_ONCE .......... 47, 120
PTL_RANK_ANY ................... 37, 68, 120
PTL_SEARCH_DELETE ............. 63, 71, 120
PTL_SEARCH_ONLY ............... 63, 71, 120
PTL_SR_DROP_COUNT ............ 25, 37, 44, 120, 133
PTL_SR_OPERATIONS_VIOLATIONS 37, 44, 58, 59, 66, 120, 133
PTL_SR_PERMISSIONS_VIOLATIONS 37, 44, 58, 66, 120, 133
PTL_SUM ......................... 96, 98
PTL_SWAP ....................... 97, 98, 101, 102
PTL_TARGET_BIND_INACCESSIBLE .. 41, 56, 57, 64, 117
PTL_TIME_FOREVER .............. 84, 90, 120
PTL_TOTAL_DATA_ORDERING . 29, 30, 41, 117
PTL_UID_ANY ..................... 37, 58, 66, 120
PTL_UINT16_T .................. 97
PTL_UINT32_T .................. 97
PTL_UINT64_T .................. 97
PTL_UINT8_T .................... 97
summary .......................... 116

count (field) ...................... 81
counting event
allocate .......................... 86
enable ............................ 53, 60, 68
freeing ........................... 87
freeing triggered operations .. 87
get ............................... 88
handle ........................... 85
increment ........................ 91
poll ............................... 89
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>110</td>
</tr>
<tr>
<td>Deferred Communication Operations</td>
<td></td>
</tr>
<tr>
<td>end bundle</td>
<td>111</td>
</tr>
<tr>
<td>limit bundling</td>
<td>111</td>
</tr>
<tr>
<td>start bundle</td>
<td>110</td>
</tr>
<tr>
<td>design guidelines</td>
<td>129</td>
</tr>
<tr>
<td>desired (field)</td>
<td>42, 56, 64</td>
</tr>
<tr>
<td>discarded events</td>
<td>93</td>
</tr>
<tr>
<td>discarded messages</td>
<td>17, 19, 125</td>
</tr>
<tr>
<td>DMA</td>
<td>[13]</td>
</tr>
<tr>
<td>dropped message count</td>
<td>44, 120, 125</td>
</tr>
<tr>
<td>dropped messages</td>
<td>37, 82-84, 116</td>
</tr>
<tr>
<td>E</td>
<td>84</td>
</tr>
<tr>
<td>eq_handle (field)</td>
<td>47, 54, 55, 81-83, 121</td>
</tr>
<tr>
<td>eq_handles (field)</td>
<td></td>
</tr>
<tr>
<td>event</td>
<td>18, 23, 58, 72</td>
</tr>
<tr>
<td>enable</td>
<td>60, 68, 117, 118</td>
</tr>
<tr>
<td>occurrence</td>
<td>74</td>
</tr>
<tr>
<td>overflow list</td>
<td>60, 68</td>
</tr>
<tr>
<td>types</td>
<td>73, 75, 76</td>
</tr>
<tr>
<td>types (diagram)</td>
<td>75</td>
</tr>
<tr>
<td>unlink</td>
<td>60, 68</td>
</tr>
<tr>
<td>event (field)</td>
<td>82-84, 88-90</td>
</tr>
<tr>
<td>event queue</td>
<td>[13]</td>
</tr>
<tr>
<td>allocation</td>
<td>81</td>
</tr>
<tr>
<td>freeing</td>
<td>81</td>
</tr>
<tr>
<td>get</td>
<td>82</td>
</tr>
<tr>
<td>poll</td>
<td>83</td>
</tr>
<tr>
<td>type</td>
<td>77</td>
</tr>
<tr>
<td>wait</td>
<td>83</td>
</tr>
<tr>
<td>F</td>
<td>86</td>
</tr>
<tr>
<td>failure (field)</td>
<td></td>
</tr>
<tr>
<td>failure notification</td>
<td>76</td>
</tr>
<tr>
<td>faults</td>
<td>18</td>
</tr>
<tr>
<td>features (field)</td>
<td>29, 41, 56, 64, 96</td>
</tr>
<tr>
<td>fetch and atomic operation</td>
<td>115</td>
</tr>
<tr>
<td>flow control</td>
<td>81</td>
</tr>
<tr>
<td>support</td>
<td>31</td>
</tr>
<tr>
<td>user-level</td>
<td>16</td>
</tr>
<tr>
<td>function return codes</td>
<td>see return codes</td>
</tr>
<tr>
<td>PtAtomic</td>
<td>22, 92, 95, 98, 98, 99, 103, 106, 112-115, 124, 135</td>
</tr>
<tr>
<td>PtAtomicSync</td>
<td>96, 102, 103, 103, 115</td>
</tr>
<tr>
<td>PtICTAlloc</td>
<td>32, 85, 86, 86, 113, 114</td>
</tr>
<tr>
<td>PtICTCancelTriggered</td>
<td>87, 87, 103, 114</td>
</tr>
<tr>
<td>PtICTFree</td>
<td>32, 85, 87, 87-90, 114, 115</td>
</tr>
<tr>
<td>PtICTGet</td>
<td>85, 87, 88, 88-91, 114-116</td>
</tr>
<tr>
<td>PtICTInc</td>
<td>85, 87, 88, 91, 91, 109, 114</td>
</tr>
<tr>
<td>PtICTPoll</td>
<td>32, 85, 89, 90, 90, 114-116</td>
</tr>
<tr>
<td>PtICTSet</td>
<td>85, 87, 88, 90, 91, 91, 109, 110, 114</td>
</tr>
<tr>
<td>PtICTWait</td>
<td>32, 85, 87, 88, 89, 89, 90, 114-116</td>
</tr>
<tr>
<td>PtEndBundle</td>
<td>110, 111, 111, 115</td>
</tr>
<tr>
<td>PtIEQAlloc</td>
<td>32, 35, 73, 81, 81, 113, 114, 116</td>
</tr>
<tr>
<td>PtIEQFree</td>
<td>32, 73, 81, 82, 83, 84, 113, 114</td>
</tr>
<tr>
<td>PtIEQGet</td>
<td>73, 81, 82, 82-85, 113, 114, 116</td>
</tr>
<tr>
<td>PtIEQPoll</td>
<td>32, 73, 81-83, 84, 84, 85, 113, 115, 116</td>
</tr>
<tr>
<td>PtIEQWait</td>
<td>32, 73, 81, 82, 83, 83-85, 113, 115, 116</td>
</tr>
<tr>
<td>PtFetchAtomic</td>
<td>22, 73, 92, 95, 98, 99, 100, 100, 101, 106, 107, 113-115</td>
</tr>
<tr>
<td>PtFini</td>
<td>38, 39, 39, 115, 116</td>
</tr>
<tr>
<td>PtGet</td>
<td>92, 94, 95, 99, 105, 113-115, 123, 124</td>
</tr>
<tr>
<td>PtGetId</td>
<td>43, 50, 51, 113-115</td>
</tr>
<tr>
<td>PtGetMap</td>
<td>45, 46, 46, 115, 121</td>
</tr>
<tr>
<td>PtGetPhysId</td>
<td>40, 43, 50, 51, 113-115</td>
</tr>
<tr>
<td>PtGetUid</td>
<td>50, 50, 114, 115</td>
</tr>
<tr>
<td>PtHandleIsEqual</td>
<td>112, 112, 113, 115</td>
</tr>
<tr>
<td>PtInit</td>
<td>35, 38, 38, 39, 115, 116</td>
</tr>
<tr>
<td>PtILEAppend</td>
<td>31, 56, 60, 61, 62, 73, 74, 78, 79, 116</td>
</tr>
<tr>
<td>PtILESearch</td>
<td>62, 63, 63, 72, 74, 114</td>
</tr>
<tr>
<td>PtILEUnlink</td>
<td>25, 56, 61, 62, 62, 116</td>
</tr>
<tr>
<td>PtIMDBind</td>
<td>37, 52, 55, 55, 113, 115, 116</td>
</tr>
<tr>
<td>PtIMDRelase</td>
<td>37, 52, 55, 56, 113, 115, 116</td>
</tr>
<tr>
<td>PtIMEAppend</td>
<td>31, 39, 64, 69, 69-71, 73, 74, 78, 79, 113-116</td>
</tr>
<tr>
<td>PtIMESearch</td>
<td>71, 72, 72, 74, 113-115</td>
</tr>
<tr>
<td>PtIMEUnlink</td>
<td>25, 32, 57, 64, 70, 71, 71, 113, 115, 116</td>
</tr>
<tr>
<td>PtINIFini</td>
<td>39, 41, 43, 43, 83, 84, 88-90, 113, 115</td>
</tr>
<tr>
<td>PtINHandle</td>
<td>40, 44, 44, 113, 115</td>
</tr>
<tr>
<td>PtINInit</td>
<td>29, 30, 39, 40, 41, 41-43, 45, 56, 64, 96, 113, 115, 116, 133</td>
</tr>
<tr>
<td>PtINStatus</td>
<td>37, 39, 40, 43, 44, 58, 66, 113-115</td>
</tr>
<tr>
<td>PtPTAlloc</td>
<td>46, 47, 81, 114-116</td>
</tr>
<tr>
<td>PtPTDisable</td>
<td>32, 48, 48, 76, 114, 115</td>
</tr>
<tr>
<td>PtPTEnable</td>
<td>31, 32, 49, 49, 114, 115</td>
</tr>
<tr>
<td>PtPTFree</td>
<td>48, 48, 114-116</td>
</tr>
<tr>
<td>PtPut</td>
<td>92, 93, 93, 95, 98-100, 102, 104, 112-115, 122, 123, 125</td>
</tr>
<tr>
<td>PtSetMap</td>
<td>40, 45, 45, 115, 116, 121</td>
</tr>
<tr>
<td>Field Name</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
</tr>
<tr>
<td>max_entries</td>
<td>40</td>
</tr>
<tr>
<td>max_eqs</td>
<td>40</td>
</tr>
<tr>
<td>max_fetch_atomic_size</td>
<td>41, 95, 99, 101</td>
</tr>
<tr>
<td>max_iovecs</td>
<td>41</td>
</tr>
<tr>
<td>max_list_size</td>
<td>41</td>
</tr>
<tr>
<td>max_mds</td>
<td>40</td>
</tr>
<tr>
<td>max_msg_size</td>
<td>30, 41</td>
</tr>
<tr>
<td>max_index</td>
<td>40</td>
</tr>
<tr>
<td>max_triggered_ops</td>
<td>41</td>
</tr>
<tr>
<td>max_unexpected_headers</td>
<td>31, 40</td>
</tr>
<tr>
<td>max_volatility_size</td>
<td>41, 54</td>
</tr>
<tr>
<td>max_war_ordered_size</td>
<td>29, 30, 41</td>
</tr>
<tr>
<td>max_waw_ordered_size</td>
<td>29, 30, 41</td>
</tr>
<tr>
<td>MD</td>
<td>52</td>
</tr>
<tr>
<td>alignment</td>
<td>53, 133</td>
</tr>
<tr>
<td>bind</td>
<td>55</td>
</tr>
<tr>
<td>options</td>
<td>53</td>
</tr>
<tr>
<td>pending_operation</td>
<td>55, 122</td>
</tr>
<tr>
<td>release</td>
<td>52, 55, 113, 115, 116</td>
</tr>
<tr>
<td>unlink</td>
<td>123</td>
</tr>
<tr>
<td>volatile</td>
<td>41, 54</td>
</tr>
<tr>
<td>md</td>
<td>55</td>
</tr>
<tr>
<td>md_handle</td>
<td>55, 56, 93, 95, 99, 104–106, 121–124</td>
</tr>
<tr>
<td>ME</td>
<td>64</td>
</tr>
<tr>
<td>access control</td>
<td>23, 27, 49</td>
</tr>
<tr>
<td>alignment</td>
<td>66, 133</td>
</tr>
<tr>
<td>append</td>
<td>69</td>
</tr>
<tr>
<td>free</td>
<td>76, 117</td>
</tr>
<tr>
<td>ignore_bits</td>
<td>see ignore_bits</td>
</tr>
<tr>
<td>link</td>
<td>76</td>
</tr>
<tr>
<td>list_types</td>
<td>69</td>
</tr>
<tr>
<td>match_bits</td>
<td>see match bits</td>
</tr>
<tr>
<td>message_reject</td>
<td>125</td>
</tr>
<tr>
<td>options</td>
<td>66</td>
</tr>
<tr>
<td>pending_operation</td>
<td>70</td>
</tr>
<tr>
<td>permissions</td>
<td>23, 27</td>
</tr>
<tr>
<td>persistent</td>
<td>69</td>
</tr>
<tr>
<td>protection</td>
<td>23, 27</td>
</tr>
<tr>
<td>search</td>
<td>63, 71, 74, 76, 117</td>
</tr>
<tr>
<td>search_and_delete</td>
<td>63</td>
</tr>
<tr>
<td>search_operations</td>
<td>63</td>
</tr>
<tr>
<td>truncate</td>
<td>67, 118, 125</td>
</tr>
<tr>
<td>unlink</td>
<td>27, 64, 66, 70, 73–76, 113, 115–117</td>
</tr>
<tr>
<td>me</td>
<td>69, 72</td>
</tr>
<tr>
<td>me_handle</td>
<td>see also MD, [13], 29, 30</td>
</tr>
<tr>
<td>memory_descriptor</td>
<td>see also MD, [13], 29, 30</td>
</tr>
<tr>
<td>message</td>
<td>[13]</td>
</tr>
<tr>
<td>message_operation</td>
<td>[13]</td>
</tr>
<tr>
<td>message_rejection</td>
<td>125</td>
</tr>
<tr>
<td>messages, receiving</td>
<td>125</td>
</tr>
<tr>
<td>messages, sending</td>
<td>121</td>
</tr>
<tr>
<td>min_free</td>
<td>27, 66</td>
</tr>
<tr>
<td>mlength</td>
<td>25, 53, 60, 68, 74, 79, 92</td>
</tr>
<tr>
<td>MPI</td>
<td>[13], 16, 23, 61, 63, 70, 72, 94, 130</td>
</tr>
<tr>
<td>progress_rule</td>
<td>16, 18</td>
</tr>
<tr>
<td>MPI scalability</td>
<td>16</td>
</tr>
<tr>
<td>MPP</td>
<td>[13]</td>
</tr>
<tr>
<td>N</td>
<td>[13]</td>
</tr>
<tr>
<td>NAL</td>
<td>[13]</td>
</tr>
<tr>
<td>naming_conventions</td>
<td>35</td>
</tr>
<tr>
<td>network</td>
<td>[13]</td>
</tr>
<tr>
<td>network_independence</td>
<td>16</td>
</tr>
<tr>
<td>network_interface</td>
<td>see also NI, 18, 35–37, 39, 41, 57, 65, 125</td>
</tr>
<tr>
<td>network_interface_initialization</td>
<td>41</td>
</tr>
<tr>
<td>network_interfaces</td>
<td>multiple</td>
</tr>
<tr>
<td>network_scalability</td>
<td>16</td>
</tr>
<tr>
<td>new_ct</td>
<td>91, 110</td>
</tr>
<tr>
<td>NI</td>
<td>options</td>
</tr>
<tr>
<td>retrieving_logical_maps</td>
<td>46</td>
</tr>
<tr>
<td>setting_logical_maps</td>
<td>45</td>
</tr>
<tr>
<td>NI_fin</td>
<td>43</td>
</tr>
<tr>
<td>NI_handle</td>
<td>44</td>
</tr>
<tr>
<td>NI_init</td>
<td>41</td>
</tr>
<tr>
<td>NI_status</td>
<td>43</td>
</tr>
<tr>
<td>ni_fail_type</td>
<td>31, 52, 56, 58, 59, 63, 64, 66, 71, 74, 76, 79, 85</td>
</tr>
<tr>
<td>ni_handle</td>
<td>42–52, 55, 60–63, 69, 71, 72, 81, 86, 96, 110, 111</td>
</tr>
<tr>
<td>niid</td>
<td>50</td>
</tr>
<tr>
<td>node</td>
<td>[13]</td>
</tr>
<tr>
<td>nodeID</td>
<td>24, 27, 37, 50</td>
</tr>
<tr>
<td>non_matching_address_translation</td>
<td>26</td>
</tr>
<tr>
<td>NULL_LE</td>
<td>58</td>
</tr>
<tr>
<td>NULL_ME</td>
<td>66</td>
</tr>
<tr>
<td>O</td>
<td>offset</td>
</tr>
<tr>
<td>local</td>
<td>66, 67, 78, 79, 93–95, 118</td>
</tr>
<tr>
<td>remote</td>
<td>58, 66, 78, 79, 94, 95, 99, 100, 102</td>
</tr>
<tr>
<td>one_sided_operation</td>
<td>17</td>
</tr>
<tr>
<td>opening_into_address_space</td>
<td>19</td>
</tr>
<tr>
<td>operand</td>
<td>124</td>
</tr>
<tr>
<td>operand</td>
<td>95, 101, 102, 108</td>
</tr>
<tr>
<td>operation</td>
<td>99, 100, 102, 106–108</td>
</tr>
<tr>
<td>operation violations count</td>
<td>120</td>
</tr>
<tr>
<td>operations</td>
<td>acknowledgment</td>
</tr>
<tr>
<td>atomic</td>
<td>13, 19, 21, 27, 29, 41, 43, 59, 66, 73, 74, 92, 98, 98, 121, 124–126</td>
</tr>
<tr>
<td>atomic sync</td>
<td>102</td>
</tr>
<tr>
<td>atomics</td>
<td>95</td>
</tr>
<tr>
<td>fetch_and_atomic</td>
<td>99</td>
</tr>
<tr>
<td>get</td>
<td>13, 19, 21, 22, 27, 29, 41, 43, 58, 59, 66, 67, 73, 74, 76, 94, 95, 96, 115, 117, 118, 121, 123–126</td>
</tr>
</tbody>
</table>
one-sided ................................................ 17
put 13, 17, 19, 21, 27, 29, 32, 41, 43, 45, 52, 54, 56, 58,
59, 62, 64, 66, 67, 70, 73, 74, 76, 92, 93,
93–96, 115, 118, 121–123, 125, 126
reply ................................................. 21, 29, 31, 41, 43, 59, 67, 73, 76, 95, 121,
123–125
swap .................................................. 101
two-sided ............................................ 17, 27
options (field) ...................... 42, 47, 53, 58, 66, 122–124
Ordering ............................ 29
adaptive .................................. 30
long messages ......................... 30
overlapping regions ................ 30
short messages .......................... 29
unexpected messages ............ 30
ordering semantics ................. 17, 29, 30, 53
OS bypass ............................. 16, 18, 18, 130
overflow list .......................... 20, 24, 27, 31, 56, 57, 63, 64, 74, 78, 91,
135

P
parallel job .................................... 17
pending operation ............... see MD performance .......................... 129
permission violations count ...... 44, 120
PGAS ........................................... 16, 130
pid (field) ..................... 42, 43, 51, portability ........................ 39
portal
index ........................................ 37
table ......................................... 39, 133
table index .......................... 47–49, 56, 64, 122–125
portal table entry ............ 35, 46
allocation ............................ 46
disable .................................. 48
enable ....................................... 49
freeing .................................. 48
portal table entry disabled event 117
Portals
early versions ..................... 12
Version 2.0 .......................... 12
Version 3.0 .......................... 12
portals
addressing .......................... see address translation
constants .......................... see constants, 36
constants summary ............. 116
data buffers .......................... 27
data types .......................... 36, 112
design ................................. 129
functions .......................... see functions
functions summary ............... 114
handle .................................. 36
multi-threading ................. 32
naming conventions ........... 35
operations .......................... see operations
ordering ................................. 29
return codes ........................ see return codes
return codes summary .......... see 115
scalability .......................... 17
sizes ........................................ 36
portals.h .................................. 35
priority list .......................... [13], 24, 26, 56, 57, 62, 71
process .................................. 32
process ID .......................... 23, 24, 27, 37, 42, 50, 50–52, 64, 68, 70, 94,
99, 100, 102, 113
well known .......................... 42
progress .................................. 18
progress rule ........................ 16, 18
protected space ................. 20, 21
PT
options .................................. 47
pt_index (field) .............. 47–49, 60–63, 69, 71, 72, 79, 94, 95, 99,
100, 102, 104–108, 122–124
pt_index_req (field) ........ 35, 92–94, 99, 116, 122
PTL_ACK_REQ (const) .... 35, 92–94, 99, 116, 122
PTL_ARG_INVALID (return code) .. 38, 42–52, 55, 56,
61, 62, 64, 70–72, 81–84, 86–92, 94, 95, 99,
101, 102, 105–111, 115
PTL_BAND (const) .......... 96, 98
PTL_BOR (const) ................. 96, 98
PTL_BXOR (const) .......... 97, 98
PTL_COHERENT_ATOMICS (const) 41, 96
PTL_CSWAP (const) ...... 95, 97, 98, 101, 102
PTL_CSWAP_GE (const) .... 97, 98
PTL_CSWAP_GT (const) .... 97, 98
PTL_CSWAP_LE (const) .... 97, 98
PTL_CSWAP_LT (const) .... 97, 98
PTL_CSWAP_NE (const) ... 97, 98
PTL_CT_ACK_REQ (const) . 92–94, 99, 116, 122, 124
PTL_CT_NONE (const) .... 36, 54, 58, 86, 116
PTL_CT_NONE_REACHED (return code) .. 90, 115
PTL_DOUBLE (const) .... 97, 117
PTL_DOUBLE_COMPLEX (const) ... 97
PTL_EQ_DROPPED (return code) .. 82–84, 116
PTL_EQ_EMPTY (return code) .. 82, 84, 116
PTL_EQ_NONE (const) ...... 36, 47, 54, 75, 116
PTL_EVENT_ACK (const) ........ 31, 53, 56–59, 65, 66,
74–77, 79, 80, 93, 96, 99, 116
PTL_EVENT_ATOMIC (const) .... 59, 60, 67, 68, 73, 76,
80, 96, 98, 116
PTL_EVENT_ATOMIC_OVERFLOW (const) .. 60, 63,
68, 71, 73, 76, 80, 98, 116
PTL_EVENT_AUTO_FREE (const) .. 57, 60, 64, 65, 68,
74, 76, 80, 117
PTL_EVENT_AUTO_UNLINK (const) .. 57, 60, 64, 65,
68, 74–76, 80, 117
PTL_EVENT_FETCH_ATOMIC (const) 73, 76, 80, 99,
116
PtlMDBind (func) 37, 52, 55, 55, 113, 115, 116
PtlMDRelease (func) 37, 52, 55, 56, 113, 115, 116
PtlMEAppend (func) 31, 39, 64, 69, 69–71, 73, 74, 78, 79, 113–116
PtlMESearch (func) 71, 72, 72, 74, 113–115
PtlMEUnlink (func) 25, 32, 57, 64, 70, 71, 71, 113, 115, 116
PtnIFini (func) 39, 41, 43, 43, 83, 84, 88–90, 113, 115
PtnIHandle (func) 40, 44, 44, 113, 115
PtnInit (func) 29, 30, 39, 40, 41–43, 45, 56, 64, 96, 113, 115, 116
PtnStatus (func) 37, 39, 40, 43, 44, 58, 66, 113–115
PtpTAlloc (func) 46, 47, 81, 114–116
PtpTDisable (func) 32, 48, 48, 76, 114, 115
PtpTEnable (func) 31, 32, 49, 49, 114, 115
PtpTFree (func) 48, 48, 114–116
PtpPut (func) 92, 93, 93, 95, 98–100, 102, 104, 112–115, 122, 123, 125
PtpSetMap (func) 40, 45, 45, 115, 116, 121
PtpStartBundle (func) 110, 110, 111, 115
PtpTriggeredAtomic (func) 103, 106, 112–115
PtpTriggeredCTInc (func) 106, 109, 109, 115
PtpTriggeredCTSet (func) 106, 109, 115
PtpTriggeredFetchAtomic (func) 107, 113–115
PtpTriggeredGet (func) 103, 105, 105, 113–115
PtpTriggeredPut (func) 104, 104, 109, 112–115
PtpTriggeredSwap (func) 108, 113–115
ptlsearchop (field) 62, 63, 71, 72
Puma 16
purpose 15
put_md_handle (field) see operations
PTL_CT_NONE_REACHED 90, 115
PTL_EQ_DROPPED 82–84, 116
PTL_EQ_EMPTY 82, 84, 116
PTL_FAIL 38, 116
PTL_IGNORED 45, 116, 121
PTL_IN_USE 61, 62, 70, 71, 75, 116
PTL_INTERRUPTED 83, 84, 89, 90, 116
PTL_LIST_TOO_LONG 61, 70, 116
PTL_NO_SPACE 42, 43, 45, 46, 55, 61, 70, 81, 86, 116
PTL_PID_IN_USE 43, 116
PTL_PT_EQ_NEEDED 47, 116
PTL_PT_FULL 47, 116
PTL_PT_IN_USE 47, 48, 116
PTL_SIZE_INVALID 116
summary 115
rlength (field) 25, 74, 78, 79
RMPP

S
scalability 17, 129
guarantee 17
MPI 16
network 16
scatter/gather 53, 54, 58, 59, 66, 67, 113, 118
Search
event generation 63, 72
status registers 63, 72
send 19
send event 93, 96, 99, 117
Set Map 45
SHMEM 16
shmem_fence() 30
size (field) 84, 90
sizes 36
space
application 20
protected 20
split event sequence see event start/end
start (field) 53, 54, 58, 66, 74, 78
state 17
status (field) 44
status registers 37, 133
status_register (field) 44
structure fields and argument names
ack_req 94, 99, 104, 106, 122, 124
actual 29, 30, 42, 43, 56, 64
actual_map_size 46

Q
quality implementation 42
quality of implementation 17

R
rank 20, 37, 45, 46, 50–52
rank (field) 37, 51
README 35, 133
receiver-managed 16
reliable communication 17
remote offset see offset
remote_offset (field) 79, 94, 95, 99, 100, 102, 104–108, 122–124
reply see operations
return codes 38, 115

PTL_ARG_INVALID 38, 42–52, 55, 56, 61, 62, 64, 70–72, 81–84, 86–92, 94, 95, 99, 101, 102, 105–111, 115
atomic_operation ........................................... 79, 85
atomic_type .................................................. 79, 85
count .......................................................... 81
counter ........................................................ 54, 55, 58, 66, 86–93, 109, 110
counter_h .......................................................... 90
datatype ........................................................... 98–102, 106–108
desired ............................................................... 42, 56, 64
eq_handle ......................................................... 47, 54, 55, 81–83, 121
eq_handles ....................................................... 84
event ............................................................... 82–84, 88–90
failure ............................................................. 86
features .......................................................... 29, 41, 56, 64, 96
get_md_handle .................................................. 96, 99–101, 107, 108, 124
handle ............................................................ 44
handle1 ............................................................. 112
handle2 ............................................................. 112
hdr_data ......................................................... 74, 78, 94, 99, 100, 102, 104, 106–108, 122
id ................................................................. 51, 52
iface .............................................................. 42
ignore_bits ....................................................... 68
increment ....................................................... 91, 109
initiator ......................................................... 79
too_v_base ..................................................... 54
too_len ......................................................... 55
too ............................................................ 61, 63
too_handle ..................................................... 61, 62
length ........................................................... 41, 53, 58, 66, 93, 95, 96, 99, 100, 102, 104–108, 122–124
local_get_offset ............................................... 99–101, 107, 108
local_offset ..................................................... 93, 95, 99, 104–106, 123, 124
local_put_offset ............................................... 99–102, 107, 108
map_size ....................................................... 45, 46
mapping ......................................................... 45, 46
match_bits ..................................................... 68, 78, 85, 94, 95, 99, 100, 102, 104–108, 122–124
match_id ....................................................... 65, 68, 70
max_atomic_size ............................................... 27, 29, 41, 95, 98–100, 102
max_ct .......................................................... 40
max_entries .................................................... 40
max_eq .......................................................... 40
max_fetch_atomic_size ........................................ 41, 95, 99, 101
max_iovecs ..................................................... 41
max_list_size .................................................. 41
max_mds ........................................................ 40
max_msg_size .................................................. 30, 41
max_pt_index .................................................. 40
max_triggered_ops ............................................ 41
max_unexpected_headers ..................................... 31, 40
max_volatile_size ............................................. 41, 54
max_war_ordered_size ........................................ 29, 30, 41
max_waw_ordered_size ....................................... 29, 30, 41
md .............................................................. 55
md_handle ...................................................... 55, 56, 93, 95, 99, 104–106, 121–124
me .............................................................. 69, 72
me_handle ...................................................... 70, 71
min_free ......................................................... 27, 66
mlength ......................................................... 25, 53, 60, 68, 74, 79, 92
new_ct .......................................................... 91, 110
ni_fail_type .................................................... 31, 52, 56, 58, 59, 63, 64, 66, 71, 74, 76, 79, 85
ni_handle ....................................................... 42–52, 55, 60–63, 69, 71, 72, 81, 86, 96, 110, 111
nid ............................................................... 50
operand .......................................................... 95, 101, 102, 108
operation ....................................................... 99, 100, 102, 106–108
options ........................................................ 42, 47, 53, 58, 66, 122–124
pid .............................................................. 42, 43, 51
pt_index_req ................................................... 47
pt_list .......................................................... 60, 61, 69, 79
ptlsearchop .................................................... 62, 63, 71, 72
put_md_handle ................................................ 96, 99–102, 107, 108, 123, 124
rank ............................................................. 37, 51
remote_offset ................................................ 79, 94, 95, 99, 100, 102, 104–108, 122–124
rlen .............................................................. 25, 74, 78, 79
size ............................................................. 84, 90
start ........................................................... 53, 54, 58, 66, 74, 78
status .......................................................... 44
status_register ............................................... 44
success ......................................................... 86
target_id ....................................................... 94, 95, 99, 100, 102, 104–108, 122–124
test ............................................................ 89, 90
tests ............................................................ 90
threshold ...................................................... 103–110
timeout ......................................................... 84, 90
trig_ct ........................................................ 103–110
type ............................................................ 79, 85
uid ............................................................. 50, 58, 66, 79
user_ptr ....................................................... 61, 63, 70, 72, 74, 78, 79, 85, 94, 95, 99, 100, 102, 104–108, 121–124
which .......................................................... 83, 84, 89, 90
success (field) .................................................. 86
summary ....................................................... 112
SUNMOS ....................................................... [14], 16
swap operation ............................................... 115

T
target . . see also initiator, 13, [14], 17–19, 21, 23, 24, 49, 73–77, 92–96, 99, 100, 102, 122–125
target_id (field) ................................................ 94, 95, 99, 100, 102, 104–108, 122–124
TCP/IP .......................................................... 16, 130
test (field) ....................................................... 89, 90
tests (field) ..................................................... 90
thread ......................................................... [14], 32
thread ID ....................................................... 50
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)