The Portals 4.0 Message Passing Interface

Rolf Riesen, Ron Brightwell, Kevin Pedretti, and Brian Barrett, Sandia National Laboratories
Keith Underwood, Intel Corporation
Arthur B. Maccabe, University of New Mexico,
Trammell Hudson, Rotomotion

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's

Approved for public release; further dissemination unlimited.
Abstract

This report presents a specification for the Portals 4.0 message passing 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. Version 3.0 of Portals runs on the Cplant cluster at Sandia National Laboratories, and version 3.3 is running on Cray’s Red Storm system. 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, Lee Ann Fisk, David Greenberg, Eric Hoffman, Gabi Istrail, Jeanette Johnston, Chu Jong, Clint Kaul, Mike Levenhagen, Kevin McCurley, Jim Otto, David Robboy, Mark Sears, Lance Shuler, Jim Schutt, Mack Stallcup, Todd Underwood, David van Dresser, Dena Vigil, Lee Ward, and Stephen Wheat.

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

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

**List of Figures** 9  
**List of Tables** 10  
**List of Implementation Notes** 12  
**Preface** 13  
**Nomenclature** 14  

### 1 Introduction

1.1 Overview 17  
1.2 Purpose 18  
1.3 Background 18  
1.4 Scalability 19  
1.5 Communication Model 20  
1.6 Zero Copy, OS Bypass, and Application Bypass 20  
1.7 Faults 21  

### 2 An Overview of the Portals API

2.1 Data Movement 23  
2.2 Portals Addressing 27  
2.3 Flow Control 32  
2.4 Multi-Threaded Applications 33  
2.5 Usage 33  

### 3 The Portals API

3.1 Naming Conventions and Typeface Usage 35  
3.2 Base Types 36  
3.2.1 Sizes 36  
3.2.2 Handles 36  
3.2.3 Indexes 36  
3.2.4 Match Bits 36  
3.2.5 Network Interfaces 37  
3.2.6 Identifiers 37  
3.2.7 Status Registers 37  
3.3 Return Codes 37  
3.4 Initialization and Cleanup 37  
3.4.1 PtlInit 38
3.4.2 PtlFini ................................................................. 38
3.5 Network Interfaces ................................................. 38
  3.5.1 The Network Interface Limits Type ......................... 40
  3.5.2 PtlNIInit ....................................................... 40
  3.5.3 PtlNIFini ...................................................... 43
  3.5.4 PtlNIStatus ................................................... 43
  3.5.5 PtlNIHandle .................................................. 44
3.6 Portal Table Entries ............................................... 45
  3.6.1 PtlPTAlloc .................................................... 45
  3.6.2 PtlPTFree ..................................................... 46
  3.6.3 PtlPTDisable ................................................ 46
  3.6.4 PtlPTEnable .................................................. 47
3.7 User Identification ................................................... 47
  3.7.1 PtlGetUid ...................................................... 47
3.8 Process Identification .............................................. 48
  3.8.1 The Process Identification Type ............................ 48
  3.8.2 PtlGetId ...................................................... 49
3.9 Process Aggregation ................................................. 49
  3.9.1 PtlGetJid ...................................................... 50
3.10 Memory Descriptors ............................................... 50
  3.10.1 The Memory Descriptor Type ................................. 50
  3.10.2 The I/O Vector Type ....................................... 52
  3.10.3 PtlMDBind ................................................... 52
  3.10.4 PtlMDRelease ............................................... 53
3.11 List Entries and Lists ............................................. 54
  3.11.1 The List Entry Type ....................................... 54
  3.11.2 PtlLEAppend ............................................... 57
  3.11.3 PtlLEUnlink ............................................... 58
3.12 Match List Entries and Matching Lists ......................... 59
  3.12.1 The Match List Entry Type ................................. 59
  3.12.2 PtlMEAppend ............................................... 62
  3.12.3 PtlMEUnlink ............................................... 64
3.13 Events and Event Queues ......................................... 65
  3.13.1 Kinds of Events ............................................ 65
  3.13.2 Event Occurrence ......................................... 66
  3.13.3 Failure Notification ....................................... 69
  3.13.4 The Event Queue Types ................................... 69
  3.13.5 PtlEQAlloc ................................................ 71
  3.13.6 PtlEQFree .................................................. 72
B.2 The Will Requirements ................................................................. 116
B.3 The Should Requirements .......................................................... 116

C A README Template .................................................................. 119

D Implementations ........................................................................ 121
  D.1 Reference Implementation ....................................................... 121
  D.2 Portals 3.3 on the Cray XT3/XT4/XT5 Red Storm ....................... 122
    D.2.1 Generic ........................................................................... 122
    D.2.2 Accelerated ..................................................................... 122

E Summary of Changes .................................................................... 123

Index ............................................................................................. 124
List of Figures

2.1 Graphical Conventions ................................................................. 23
2.2 Portals Put (Send) ..................................................................... 24
2.3 Portals Get (Receive) from a match list entry ................................. 25
2.4 Portals Get (Receive) from a list entry ........................................ 26
2.5 Portals Atomic Swap Operation ................................................... 26
2.6 Portals Atomic Sum Operation ..................................................... 27
2.7 Portals LE Addressing Structures ................................................. 28
2.8 Portals ME Addressing Structures ............................................... 29
2.9 Matching Portals Address Translation ......................................... 30
2.10 Non-Matching Portals Address Translation .................................. 31
2.11 Simple Put Example ................................................................... 34
3.1 Portals Operations and Event Types ............................................. 67
# List of Tables

3.1 Object Type Codes ................................................................. 35
3.2 Event Type Summary .............................................................. 68
3.3 Portals Data Types ................................................................. 97
3.4 Portals Functions ................................................................. 98
3.5 Portals Return Codes .............................................................. 99
3.6 Portals Constants ................................................................. 100

4.1 Send Request ............................................................... 106
4.2 Acknowledgment .............................................................. 107
4.3 Acknowledgment .............................................................. 107
4.4 Get Request ............................................................... 108
4.5 Reply ............................................................... 108
4.6 Atomic Request .............................................................. 109
4.7 Portals Operations and ME/LE Flags ..................................... 110
# List of Implementation Notes

<table>
<thead>
<tr>
<th></th>
<th>Note</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No wire protocol</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>Weak Ordering Semantics</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>User memory as scratch space</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Don’t alter put or reply buffers</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Location of event queues and counters</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Protected space</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Overflow list</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>Non-matching address translation</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>README and portals4.h</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>Network interface encoded in handle</td>
<td>36</td>
</tr>
<tr>
<td>11</td>
<td>Size of handle types</td>
<td>36</td>
</tr>
<tr>
<td>12</td>
<td>Supporting fork()</td>
<td>38</td>
</tr>
<tr>
<td>13</td>
<td>Logical network interfaces</td>
<td>39</td>
</tr>
<tr>
<td>14</td>
<td>Multiple calls to <code>PtlNIInit()</code></td>
<td>42</td>
</tr>
<tr>
<td>15</td>
<td>Object encoding in handle</td>
<td>44</td>
</tr>
<tr>
<td>16</td>
<td>Support of I/O Vector Type and Offset</td>
<td>52</td>
</tr>
<tr>
<td>17</td>
<td>Unique memory descriptor handles</td>
<td>53</td>
</tr>
<tr>
<td>18</td>
<td>Checking <code>match_id</code></td>
<td>64</td>
</tr>
<tr>
<td>19</td>
<td>Overflow Events</td>
<td>66</td>
</tr>
<tr>
<td>20</td>
<td>Pending operations and buffer modifications</td>
<td>67</td>
</tr>
<tr>
<td>21</td>
<td>Pending operations and acknowledgment</td>
<td>68</td>
</tr>
<tr>
<td>22</td>
<td>Completion of portals operations</td>
<td>69</td>
</tr>
<tr>
<td>23</td>
<td>Location of event queue</td>
<td>72</td>
</tr>
<tr>
<td>24</td>
<td>Size of event queue and reserved space</td>
<td>72</td>
</tr>
<tr>
<td>25</td>
<td>Fairness of <code>PtlIEQPoll()</code></td>
<td>74</td>
</tr>
<tr>
<td>26</td>
<td>Macros using <code>PtlIEQPoll()</code></td>
<td>75</td>
</tr>
<tr>
<td>27</td>
<td>Filling in the <code>ptl_event_t</code> and <code>ptl_target_event_t</code> structures</td>
<td>75</td>
</tr>
<tr>
<td>28</td>
<td>Counting Event Handles</td>
<td>76</td>
</tr>
<tr>
<td>29</td>
<td>Minimizing cost of counting events</td>
<td>77</td>
</tr>
<tr>
<td>30</td>
<td>Functions that require communication</td>
<td>80</td>
</tr>
<tr>
<td>31</td>
<td>Ordering of Triggered Operations</td>
<td>89</td>
</tr>
<tr>
<td>32</td>
<td>Implementation of Triggered Operations</td>
<td>89</td>
</tr>
<tr>
<td>33</td>
<td>Triggered Operations Reaching the Threshold</td>
<td>89</td>
</tr>
<tr>
<td>34</td>
<td>Information on the wire</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---</td>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>35</td>
<td>Size of data on the wire</td>
<td>106</td>
</tr>
<tr>
<td>36</td>
<td>Acknowledgment requests</td>
<td>107</td>
</tr>
<tr>
<td>37</td>
<td>Implementations of Portals 3.3</td>
<td>121</td>
</tr>
</tbody>
</table>
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$^\text{tm}$ blocks to create more complex structures. The operating system kernel handling the data transfer read these structures when data began to arrive and determined where to place the incoming data. Users were allowed to create matching criteria and to specify precisely where data would eventually end up. The kernel, in turn, had the ability to DMA data directly into user space, which eliminated buffer space in kernel owned memory and slow memory-to-memory copies. We named that approach Portals Version 2.0. It was used until 2006 on the ASCI Red supercomputer, the first general-purpose machine to break the one teraflops barrier.

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

We defined the Version 3.0 API in Brightwell, Hudson, Riesen, and Maccabe (1999). Since then, Portals have gone through three revisions. The latest was Version 3.3 Riesen, Brightwell, Maccabe, Hudson, and Pedretti (2006). 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 networking 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 relative to the target message rates has risen dramatically. In addition, partitioned global address space (PGAS) models have risen in prominence and require lighter weight semantics to support them.
# Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>Acknowledgement.</td>
</tr>
<tr>
<td>FM</td>
<td>Illinois Fast Messages.</td>
</tr>
<tr>
<td>AM</td>
<td>Active Messages.</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface. A definition of the functions and semantics provided by library of functions.</td>
</tr>
<tr>
<td>ASC</td>
<td>Advanced Simulation and Computing Initiative.</td>
</tr>
<tr>
<td>ASCI</td>
<td>Advanced Simulation and Computing.</td>
</tr>
<tr>
<td>ASCI Red</td>
<td>Intel TFlops system installed at Sandia National Laboratories. First general-purpose system to break one teraflop barrier.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit.</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access.</td>
</tr>
<tr>
<td>EQ</td>
<td>Event Queue.</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out.</td>
</tr>
<tr>
<td>FLOP</td>
<td>Floating Point OPeration. (Also FLOPS or flops: Floating Point OPerations per Second.)</td>
</tr>
<tr>
<td>GM</td>
<td>Glenn’s Messages; Myricom’s Myrinet API.</td>
</tr>
<tr>
<td>ID</td>
<td>Identifier</td>
</tr>
<tr>
<td>Initiator</td>
<td>A process that initiates a message operation.</td>
</tr>
<tr>
<td>IOVEC</td>
<td>Input/Output Vector.</td>
</tr>
<tr>
<td>LE</td>
<td>List Entry.</td>
</tr>
<tr>
<td>MD</td>
<td>Memory Descriptor.</td>
</tr>
<tr>
<td>ME</td>
<td>Matching list Entry.</td>
</tr>
<tr>
<td>Message</td>
<td>An application-defined unit of data that is exchanged between processes.</td>
</tr>
<tr>
<td>Message Operation</td>
<td>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.</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface.</td>
</tr>
<tr>
<td>MPP</td>
<td>Massively Parallel Processor.</td>
</tr>
<tr>
<td>NAL</td>
<td>Network Abstraction Layer.</td>
</tr>
<tr>
<td>NAND</td>
<td>Bitwise Not AND operation.</td>
</tr>
<tr>
<td>Network</td>
<td>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.</td>
</tr>
<tr>
<td>NI</td>
<td>Abstract portals Network Interface.</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card.</td>
</tr>
<tr>
<td>Node</td>
<td>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.</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System.</td>
</tr>
<tr>
<td>PM</td>
<td>Message passing layer for SCoreD [Ishikawa et al. 1996].</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface.</td>
</tr>
<tr>
<td>Process</td>
<td>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.</td>
</tr>
<tr>
<td>RDMA</td>
<td>Remote Direct Memory Access.</td>
</tr>
<tr>
<td>RMPP</td>
<td>Reliable Message Passing Protocol.</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>SMP</td>
<td>Shared Memory Processor.</td>
</tr>
<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>Teraflop</td>
<td>$10^{12}$ flops.</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 message passing between nodes in a system area network. The goal of this interface is to improve the scalability and performance of network communication by defining the functions and semantics of message passing required for scaling a parallel computing system to 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\(^1\).

This document is divided into several sections:

Section 1 – Introduction.
This section describes the purpose and scope of the portals API\(^2\).

Section 2 – An Overview of the Portals 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 – The Semantics of Message Transmission.
This section describes the semantics of message transmission. In particular, the information transmitted in each type of message and the processing of incoming messages.

Appendix A – FAQ.
Frequently Asked Questions about Portals.

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

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

Appendix D – Implementations.
A brief description of the portals 4.0 reference implementation and the implementations that run on Cray’s XT3/XT4/XT5 Red Storm machine.

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

---

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

\(^2\)We use the lower case portals when it is used as an adjective; e.g., portals document, a (generic) portals address, or portals operations. We use the singular when we refer to a specific portal or its attributes; e.g., portal index, portal table, or a (specific) portal address.
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. 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) \cite{MPI1994} standard as well as the various partitioned global address space (PGAS) models, such as unified parallel C (UPC), Co-Array Fortran (CAF), and SHMEM \cite{Cray1994}. 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 \cite{Infiniband1999}, VIA \cite{VIA1997} 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 2.3), 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) \cite{Maccabe1994} and Puma \cite{Shuler1995} lightweight kernel development projects.
Portals went through three design phases [Riesen et al. 2005], with the most recent one being used on the 13000-node (38,400 cores) Cray Red Storm [Alverson 2003] 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 [Brightwell and Shuler 1996], 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. If the implementation or the network protocols and hardware underneath it cannot scale to one million nodes, then neither Portals nor the application can.

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.

**Implementation Note 1:**  
No wire protocol

This document does not specify a wire protocol. Portals require a reliable communication layer. Whether that is achieved through software or hardware is up to the implementation. For example, for Red Storm two reliability protocols were implemented — one by Cray and one by Sandia [Brightwell et al. 2006].
**IMPLEMENTATION NOTE 2:**

**Weak Ordering Semantics**

The default ordering semantics for Portals messages only requires that messages are *started* in order at the target. 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 implementation may, however, choose to provide stronger ordering than is required.

For example, to simplify the implementation of a `shm_fence()` function, an implementation may choose to provide strict ordering of data at the target. In addition, 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). 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 posted. Thus, triggered operations are safe, since they do not trigger until the counting event is posted.

**Discussion:** The specified ordering semantics of Portals is not sufficient to allow a `shm_fence()` operation to be treated as a no-op. Specific implementations of Portals may choose to provide more strict ordering requirements, or a SHMEM implementation may promote `shm_fence()` to `shm_quiet()`.

### 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 match list entries (potentially with 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 any specific process or can choose to ignore message operations from any specific process.

### 1.6 Zero Copy, OS Bypass, and Application Bypass

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

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

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

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 3:</th>
<th>User memory as scratch space</th>
</tr>
</thead>
<tbody>
<tr>
<td>The portals API allows for user memory where data is being received to be altered (e.g. at the target, or in a reply buffer at the initiator. That means an implementation can utilize user memory as scratch space and staging buffers. Only after an operation succeeds and the event has been posted must the user memory reflect exactly the data that has arrived. The portals API explicitly prohibits modifying the the buffer passed into a put.</td>
<td></td>
</tr>
</tbody>
</table>

1.7 Faults

Given the number of components that we are dealing with and the fact that we are interested in supporting applications that run for very long times, failures are inevitable. The portals API recognizes that the underlying transport may not be able to successfully complete an operation once it has been initiated. This is reflected in the fact that the portals API reports an event indicating the successful completion of every operation. Completion events carry a flag which indicates whether the operation completed successfully or not.

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

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 4:</th>
<th>Don’t alter put or reply buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>An implementation must not alter data in a user buffer that is used in a put or reply operation. This is independent of whether the operation succeeds or fails.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2

An Overview of the Portals API

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

2.1 Data Movement

A portal represents an opening in the address space of a process. Other processes can use a portal to read (get), write (put), or 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.

![Graphical Conventions](image)

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

Figures 2.2, 2.3, 2.4, and 2.5 present graphical interpretations of the portals data movement operations: put (send), get, and atomic (atomic operation — swap 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.

23
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 that arrives on one physical interface can nonetheless target multiple logical network interfaces. Figure 2.2 shows a matching and a non-matching network interface, but a given network interface can also use logical (rank) or physical (nid/pid) identifiers to refer to network endpoints (processes). As indicated in Figure 2.2, separate 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 5 for possible locations of the event queues.
Note 5: Location of event queues and counters
Note that data structures that can only be accessed through the API, such as counters and event queues, are intended to reside in user space. However, an implementation is free to place them anywhere it wants.

Note 6: 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 ➀ 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 ❼ that includes the requested data.

![Diagram of Portals Get from a match list entry](image)

**Figure 2.3. Portals Get from a match list entry.**

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

The third operation, atomic (atomic operation), 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 ➀, containing the put data and the operand value ➄, 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 ❼ 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 Portals Addressing

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

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

Once the target buffer has been selected, the incoming message 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 sender of the message must be allowed to access this buffer, and the operation type selected must be allowed. Each list entry and match list entry has specifiers of which types of operations are allowed — put and/or get — as well as either a user ID or a job ID that can be used to identify which initiators are allowed to access the buffer. A failure in the permissions check does not modify the Portals state in any way, except to update the status registers (see Section 3.5.4).

Figures 2.7 and 2.8 are graphical representation of the structures used by a target in the interpretation of a portals address. The node identifier is used to route the message to the appropriate node and is not reflected in this diagram. The process ID\(^1\) process identifier is used to select the correct target process and the network interfaces it has initialized. The network interface used by the initiator is used to select the correct portal table. There is one portal table for each process and each interface initialized by the process; i.e., if a process initializes an interface for a Myrinet and then initializes another interface for an Ethernet, two portal tables will be created within that process, one for each interface. Similarly, if one physical interface has been initialized as a matching interface and is later initialized as a non-matching interface, each logical interface has an independent portal table. Figure 2.7 shows the flow of addressing information in the case of an unmatched NI, while Figure 2.8 illustrates the case of a matched data

\(^1\) A logical rank can be substituted for the combination of node ID and process ID when logical end-point addressing is used.
transfer.

The portal index is used to select an entry in the portal table. Each entry of the portal table identifies two lists and, optionally, an event queue. The first list is a priority list that is posted by the application to describe remotely accessible address regions. If matching is enabled for the selected network interface, each element of the priority list specifies two bit patterns: a set of “don’t care” bits and a set of “must match” bits. Along with source node ID (NID) and source process ID (PID), these bits are used in a matching function to select the correct match list entry. If matching is not enabled, the first entry in the list is used. The second list associated with each portal table entry is an overflow list. The overflow list maintains (loosely) the same semantics as the priority list. If the network interface provides matching on the priority list, then it provides it on the overflow list. If the network interface is configured to be non-matching, then the overflow list does not provide matching. The overflow list is always traversed after the priority list. It uses locally managed offsets to provide a space for the Portals implementation to store unexpected messages, and any associated state that the implementation deems necessary. The application populates the overflow list with either list entries (non-matching network interface) or match list entries (matching network interface) that are used and then unlinked by the implementation. An overflow list entry is not required to have a buffer associated with it, since the overflow list semantics allow the application to post a list entry that drops the body of messages; however, if the portal table entry has enabled flow control, then exhaustion of the overflow list will lead to a PTL_EVENT_PT_DISABLED being posted at the target when a message arrives.

List entries identify a memory region as well as an optional counting event. Matching list entries add a set of matching criteria to this identifier. For both the list entries and match list entries, the application can specify a set of protection criteria. The protection criteria includes the type of operations allowed (put and/or get) as well as who is allowed to access the buffer (either user ID, job ID, or a wildcard). The memory region specifies the memory to be used in the operation, and the counting event is used to record the occurrence of operations. Information about the operations is (optionally) recorded in the event queue attached to the portal table entry.

Figure 2.7. Portals Non-Matching Addressing Structures: The example shows the flow of information for an unmatched 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.

Figure 2.9 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

\(^2\)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.
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 event, the operation is logged in the event queue. An event is written 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. The overflow list contains a series of buffers provided by the host for use by the implementation for messages that do not match in the priority list. The Portals implementation can capture the entire message, or any portion thereof allowed by the parameters of the match list entry. If a later match list entry is posted that matches an item in the overflow list, the implementation delivers an event (PTL_EVENT_PUT_OVERFLOW) to the application that includes a start address (which can be NULL) pointing to the location of the message. If the rlength and mlength in the event are equal, the start address must be a valid address indicating the location where the message arrived. If the mlength is less than the rlength, the message was truncated. This only occurs when the application has configured match list entries to discard message bodies; thus, the application is responsible for implementing the protocol necessary to retrieve the message body. If the overflow list does not have sufficient space for the message, the incoming request is discarded and a PTL_EVENT_DROPPED event is posted to the event queue associated with the portal table entry.

**Discussion:** While overflow list semantics are convenient for managing unexpected messages, they do provide the potential for the implementation to push data movement onto the application when unexpected messages arrive. This makes it difficult, perhaps even impossible, for the implementation to
know when the data movement associated with those messages is completed. While this does not change
the ordering semantics of Portals, it highlights a subtlety that can be easily overlooked: Portals only
guarantees that messages start in order. Portals does not guarantee that messages complete in order; thus,
a `PtlGet()` that follows a `PtlPut()` is not guaranteed to return the data delivered by the `PtlPut()` unless
other, higher level ordering semantics are enforced. Similarly, 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 posted. Thus, triggered operations are safe, since they do not trigger until the counting event is posted.

Figure 2.10. Non-Matching Portals Address Translation.

Figure 2.10 shows the comparable figure for address translation on a non-matching network interface. If matching is disabled, the portals address translation is dramatically simplified. The first list entry (LE) *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, such as locally managed offsets are not supported on the priority list when matching is not enabled. Locally managed offsets are always used in the overflow list. The overflow list is checked after the priority list, if necessary. If no list entry is present, the message is discarded and a `PTL_EVENT_DROPPED` event is posted. 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 (shown in Figure 2.10) have no loops. This allows fully pipelined operation for the non-matching address translation.

In typical scenarios, MPI uses the matching interface and requests full events in the event queue. SHMEM would use the non-matching interface and request only counting events be enabled at the initiator and no events be delivered at the target. In this mode, significantly lighter weight semantics can be delivered for PGAS style messaging, while full
offloading and independent progress can be guaranteed for MPI.

**IMPLEMENTATION NOTE 7:**

Overflow list
The overflow list can be managed in a number of ways; however, the most obvious implementation would use a locally managed offset and retain entire short messages or headers only for long messages (by posting a match list entry without a buffer and setting it to truncate). The implementation is neither required to or prohibited from using any space provided by match list entries in the overflow list to store message headers; however, the application is not required to provide such space with a match list entry. Thus, the implementation must have (or be able to acquire) state of its own. It may choose to augment that state with the space provided with the match list entries to store message headers. An implementation should never place information relating to one message into two different list entries as this will bind both entries until a matching match list entry is attached.

**IMPLEMENTATION NOTE 8:**

Non-matching address translation
A quality implementation would optimize for the common case of always using the head of the list for non-matching address translation. This could allow extremely high message rates for non-matching operations.

### 2.3 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 buffer set aside for unexpected messages. While this level of unexpected messages is an example of truly terrible programming, nonetheless it is a behavior that commercial MPI implementations encounter. In the past, this has caused the loss of an event or a message and the MPI application is lost. Users then complain. As an example of how other networks solve this issue, InfiniBand uses “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 extremely 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 an event to the application. At this point, all messages targeting that portal table entry for that process must be dropped until PtlPTEnable() is called. Note that a reply does not target a portal table entry and is not dropped. In addition, the PTL_EVENT_SEND event associated with that message (and subsequent in flight messages) fails with an appropriate indication in the ni_fail_type variable. The application (e.g. MPI library) must then use a second portal table entry to recover from the overflow. Recovery is painful — 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 events from the event queue associated with the portal table entry.
replenishing the user allocated buffers on the overflow list, and draining unexpected messages from the Portals implementation.

2.4 Multi-Threaded Applications

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

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

2.5 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 an event counter inform the user when an individual transfer has completed.

Another example is Figure 2.6 which is simpler than Figure 2.5 and probably more likely to be used. 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.
Figure 2.11. Simple Put Example: Not every option or Portals features is needed to accomplish simple tasks such as the transfer of data from an initiator region to a target region.
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>3.5</td>
</tr>
<tr>
<td>PT</td>
<td>pt</td>
<td>Portal Table Entry</td>
<td>3.6</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 implementation 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 9:

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 3.5.1) for this implementation and provide a list of status registers that are provided (Section 3.2.7). See Appendix C for a template.
3.2 Base Types

The portals API defines a variety of base types. These types represent a simple renaming of the base types provided by the C programming language. In most cases these new type names have been introduced to improve type safety and to avoid issues arising from differences in representation sizes (e.g., 16-bit or 32-bit integers). Table 3.3 lists all the types defined by Portals.

3.2.1 Sizes

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

3.2.2 Handles

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

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

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

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 10:</th>
<th>Network interface encoded in handle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Each handle must encode the network interface it is associated with.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 11:</th>
<th>Size of handle types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>It is highly recommended that a handle type should be no larger than the native machine word size.</td>
</tr>
</tbody>
</table>

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 type event. See Section 3.10.1 for uses of these values. The special constant `PTL_INVALID_HANDLE` is used to represent an invalid handle.

3.2.3 Indexes

The type `ptl_pt_index_t` is an integral type used for representing portal table indexes. See Section 3.5.1 and 3.5.2 for limits on values of this type.

3.2.4 Match Bits

The type `ptl_match_bits_t` is capable of holding unsigned 64-bit integer values.
3.2.5 Network Interfaces

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

3.2.6 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, and `ptl_jid_t` is an integral type for representing job identifiers.

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

3.2.7 Status Registers

Each network interface maintains an array of status registers that can be accessed using the `PtlNIStatus()` function (Section 3.5.4). The type `ptl_sr_index_t` defines the types of indexes that can be used to access the status registers. Only two indexes are defined for all implementations: `PTL_SR_DROP_COUNT`, which identifies the status register that counts the dropped requests for the interface, and `PTL_SR_PERMISSIONS_VIOLATIONS`, which counts the number of attempted permission violations. Other indexes (and registers) may be defined by the implementation.

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

3.3 Return Codes

The API specifies return codes that indicate success or failure of a function call. In the case where the failure is due to invalid arguments being passed into the function, the exact behavior of an implementation is undefined. The API suggests error codes that provide more detail about specific invalid parameters, but an implementation is not required to return these specific error codes. For example, an implementation is free to allow the caller to fault when given an invalid address, rather than return `PTL_SEGV`. In addition, an implementation is free to map these return codes to standard return codes where appropriate. For example, a Linux kernel-space implementation could map portals return codes to POSIX-compliant return codes. Table 3.5 lists all return codes used by Portals.

3.4 Initialization and Cleanup

The portals API includes a function, `PtlInit()`, to initialize the library and a function, `PtlFini()`, to clean up after the process is done using the library. The initialization state of Portals is reference counted so that repeated calls to `PtlInit()` and `PtlFini()` within a process (collection of threads) behave properly.

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, behavior is undefined for both the parent and the child.
3.4.1 PtlInit

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

Function Prototype for PtlInit

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

Return Codes

PTL_OK Indicates success.
PTL_FAIL Indicates an error during initialization.

IMPLEMENTATION NOTE 12: Supporting fork()
If an implementation wants to support fork(), it must detect when PtlInit() is being called from a new process context and re-initialize the state of the Portals library.

3.4.2 PtlFini

The PtlFini() function allows an application to clean up after the portals library is no longer needed by a process. Each call to PtlFini() decrements the reference count that was incremented by PtlInit(). When the reference count reaches zero, all portals resources are freed. Once the portals resources are freed, calls to any of the functions defined by the portals API or use of the structures set up by the portals API will result in undefined behavior. Each call to PtlInit() should be matched by a corresponding PtlFini().

Function Prototype for PtlFini

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

3.5 Network Interfaces

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

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 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 3.5.4 for a discussion of the PtlINIStatus() 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 PtlNIFini() function is used to shut down a logical interface and release any resources that are associated with the interface. Network interface handles are associated with processes, not threads. All threads in a process share all of the network interface handles.

Discussion: Proper initialization of a logical network interface that uses logical-end point addressing requires the user to pass in a requested mapping of logical ranks to physical node IDs and process IDs. To obtain this mapping, the process must first initialize a logical network interface that uses physical end-point addressing. The logical network interface that uses physical end-point addressing can be used to exchange a NID/PID map or the NID/PID map can be retrieved from a run-time system.

The portals API also defines the PtlINIStatus() function (Section 3.5.4) to query the status registers for a logical network interface, and the PtlNIHandle() function (Section 3.5.5) to determine the logical network interface with which an object is associated.
3.5.1 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_mes;
    int max_mds;
    int max_cts;
    int max_eqs;
    int max_pt_index;
    int max_iovecs;
    int max_me_list;
    ptl_size_t max_msg_size;
    ptl_size_t max_atomic_size;
} ptl_ni_limits_t;
```

Limits

- **max_mes**
  Maximum number of match list entries that can be allocated at any one time.

- **max_mds**
  Maximum number of memory descriptors that can be allocated at any one time.

- **max_eqs**
  Maximum number of event queues that can be allocated at any one time.

- **max_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_me_list**
  Maximum number of match list entries that can be attached to any portal table index.

- **max_msg_size**
  Maximum size (in bytes) of a message (put, get, or reply).

- **max_atomic_size**
  Maximum size (in bytes) of an atomic operation.

3.5.2 PtlNIInit

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

```c
int PtlNIInit (ptr_interface_t iface,
               unsigned int options,
               ptr_pid_t pid,
               ptr_ni_limits_t *desired,
               ptr_ni_limits_t *actual,
               ptr_size_t map_size,
               ptr_process_id_t *desired_mapping,
               ptr_process_id_t *actual_mapping,
               ptr_handle_ni_t *ni_handle);
```

Arguments

### iface
**input**
Identifies the network interface to be initialized. (See Section 3.2.5 for a discussion of values used to identify network interfaces.)

### 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 value PTL_PID_ANY may be used to let the portals library select a process identifier.

### desired
**input**
If not NULL, points to a structure that holds the desired limits.

### actual
**output**
If not NULL, on successful return, the location pointed to by actual will hold the actual limits.

### map_size
**input**
Contains the size of the map being passed in (zero for NULL). This field is ignored if the PTL_NI_LOGICAL option is not set.

### desired_mapping
**input**
If not NULL, points to an array of structures that holds the desired mapping of logical identifiers to NID/PID pairs. This field is ignored if the PTL_NI_LOGICAL option is not set.

### actual_mapping
**output**
If the PTL_NI_LOGICAL option is set, on successful return, the location pointed to by actual_mapping will hold the actual mapping of logical identifiers to NID/PID pairs.

### ni_handle
**output**
On successful return, this location will hold a the interface handle.

### options
- **PTL_NI_MATCHING**
  Request that the interface specified in iface be opened with matching enabled.
- **PTL_NI_NO_MATCHING**
  Request that the interface specified in iface be opened with matching disabled. PTL_NI_MATCHING and PTL_NI_NO_MATCHING are mutually exclusive.
- **PTL_NI_LOGICAL**
  Request that the interface specified in iface be opened with logical end-point addressing (e.g. MPI communicator and rank or SHMEM PE).
- **PTL_NI_PHYSICAL**
  Request that the interface specified in iface be opened with physical end-point addressing (e.g. NID/PID). PTL_NI_LOGICAL and PTL_NI_PHYSICAL are mutually exclusive.
Discussion: The use of desired is implementation dependent. In particular, an implementation may choose to ignore this argument.

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

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

Return Codes

PTL_OK Indicates success.
PTL_NO_INIT Indicates that the portals API has not been successfully initialized.
PTL_IFACE_INVALID Indicates that iface is not a valid network interface.
PTL_PID_INVALID Indicates that pid is not a valid process identifier.
PTL_PID_INUSE Indicates that pid is currently in use.
PTL_SEGV Indicates that actual or ni_handle is not NULL or a legal address, or that desired is not NULL and does not point to a valid address.

IMPLEMENTATION NOTE 14: Multiple calls to PtlNIInit()
If PtlNIInit() gets called more than once per logical interface, then the implementation should fill in actual, actual_mapping and ni_handle. It should ignore pid. PtlGetId() (Section 3.8) can be used to retrieve the pid.
3.5.3 PtlNIFin

The PtlNIFin() 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 PtlNIFin(). 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.

**Function Prototype for PtlNIFin**

```
int PtlNIFin ( 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_NI_INVALID** Indicates that `ni_handle` is not a valid network interface handle.

3.5.4 PtlNIStatus

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

**Function Prototype for PtlNIStatus**

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

**Arguments**

- `ni_handle` input 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 two status registers are currently required: a drop count register (PTL_SR_DROP_COUNT) and an attempted permissions violation register (PTL_SR_PERMISSIONS_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_NI_INVALID**
  Indicates that \textit{ni\_handle} is not a valid network interface handle.

- **PTL_SR_INDEX_INVALID**
  Indicates that \textit{status\_register} is not a valid status register.

- **PTL_SEGV**
  Indicates that \textit{status} is not a legal address.

### 3.5.5 PtlNIHandle

The \texttt{PtlNIHandle()} function returns the network interface handle with which the object identified by \textit{handle} is associated. If the object identified by \textit{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

- **\textit{handle}**
  \textit{input} The object handle.

- **\textit{ni\_handle}**
  \textit{output} On successful return, this location will hold the network interface handle associated with \textit{handle}.

#### Return Codes

- **PTL_OK**
  Indicates success.

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

- **PTL_HANDLE_INVALID**
  Indicates that \textit{handle} is not a valid handle.

- **PTL_SEGV**
  Indicates that \textit{ni\_handle} is not a legal address.

**IMPLEMENTATION NOTE 15:** Object encoding in handle
Every handle should encode the network interface and the object identifier relative to this handle.
3.6 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.

3.6.1 PtlPTAlloc

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

Function Prototype for PtlPTAlloc

```
int PtlPTAlloc(ptl_handle ni, unsigned int options, ptl_handle eq, ptl_pt_index_t pt_index_req, ptl_pt_index_t *pt_index);
```

Arguments

- `ni` handle: input. The interface handle to use.
- `options`: input. This field contains options that are requested for the portal index. Values for this argument can be constructed using a bitwise OR of the values defined below.
- `eq` handle: input. The event queue handle used to log the operations performed on match list entries attached to the portal table entry. The `eq` handle attached to a portal table entry must refer to an event queue containing `ptl_target_event_t` type events. 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 this portal table entry will have the PTL_ME_USE_ONCE or PTL_LE_USE_ONCE option set.
- PTL_PT_FLOW_CONTROL: Enable flow control on this portal table entry (see Section 2.3).

Return Codes

- PTL_OK: Indicates success.
- PTL_NI_INVALID: Indicates that `iface` is not a valid network interface handle.
- PTL_NO_INIT: Indicates that the portals API has not been successfully initialized.
PTL_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.

3.6.2 PtlPTFree

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

Function Prototype for PtlPTFree

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

Arguments

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

Return Codes

- PTL_OK Indicates success.
- PTL_NO_INIT Indicates that the portals API has not been successfully initialized.
- PTL_PT_INDEX_INVALID Indicates that `pt_index` is not a valid portal table index.
- PTL_PT_IN_USE Indicates that `pt_index` is currently in use (e.g. a match list entry is still attached).
- PTL_NI_INVALID Indicates that `ni_handle` is not a valid network interface handle.

3.6.3 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 posted.

Function Prototype for PtlPTDisable

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

Arguments

- `ni_handle` input The interface handle to use.
- `pt_index` input The portal index that is to be disable.
Return Codes

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

**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.

### 3.6.4 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 status has been updated.

#### Function Prototype for PtlPTEnable

```c
int PtlPTEnable(ptl_t *ni, ptl_pt_index_t index);
```

#### Arguments

- **ni** *input* The interface handle to use.
- **ptl_index** *input* The value of the portal index to enable.

#### Return Codes

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

### 3.7 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).

#### 3.7.1 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_NI_INVALID** Indicates that `ni_handle` is not a valid network interface handle.
- **PTL_NO_INIT** Indicates that the portals API has not been successfully initialized.
- **PTL_SEGV** Indicates that `uid` is not a legal address.

3.8 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.

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

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

3.8.1 The Process Identification Type

The `ptl_process_id_t` type is a union that can represent the a node as either a physical address or a logical address within the machine. The physical address uses two identifiers to represent a process identifier: a node identifier `nid` and a process identifier `pid`. In turn, a logical address uses a logical index within a translation table specified by the application (the `rank`) to identify another process.
typedef union {
    struct {
        pti_t nid;
        pti_t pid;
    } phys;
    pti_rank_t rank;
} pti_process_id_t;

3.8.2 PtlGetId

Function Prototype for PtlGetId

```c
int PtlGetId (pti_handle ni ,
              pti_process_id *id);
```

Arguments

- `ni`: A network interface handle.
- `id`: On successful return, this location will hold the identifier for the calling process.

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

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

3.9 Process Aggregation

It is useful in the context of a parallel machine to represent all of the processes in a parallel job through an aggregate identifier. The portals API provides a mechanism for supporting such job identifiers for these systems. In order to be fully supported, job identifiers must be included as a trusted part of a message header.

The job identifier is an opaque identifier shared between all of the distributed processes of an application running on a parallel machine. All application processes and job-specific support programs, such as the parallel job launcher, share the same job identifier. This identifier is assigned by the runtime system upon job launch and is guaranteed to be unique among application jobs currently running on the entire distributed system. An individual serial process may be assigned a job identifier that is not shared with any other processes in the system or can be assigned the constant `PTL_JID_NONE`. 
3.9.1 PtlGetJid

Function Prototype for PtlGetJid

```c
int PtlGetJid ( ptl_handle_ni_t ni, ptl_jid_t *jid);
```

Arguments

- `ni` (input): A network interface handle.
- `jid` (output): On successful return, this location will hold the job identifier for the calling process. `PTL_JID_NONE` may be returned for a serial job, if a job identifier is not assigned.

Return Codes

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

Discussion: The notion of a job identifier is fairly closely tied to a run-time system. It is expected that the run-time system will set this value. For implementations without a run-time system, `PTL_JID_NONE` may be assigned. It would probably be a bad idea to use job ID on those systems for access control.

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). `PtlMDBind()` is used to create a memory descriptor and `PtlMDRelease()` is used to unlink and release the resources associated with a memory descriptor.

3.10.1 The Memory Descriptor Type

The `ptl_md_t` type defines the visible parts of a memory descriptor. Values of this type are used to initialize the memory descriptors.
typedef struct {
    void *start;
    ptl_size_t length;
    unsigned int options;
    ptl_handle_eq_t eq_handle;
    ptl_handle_ct_t ct_handle;
} ptl_md_t;

Members

*start, length*  
Specify the memory region associated with the memory descriptor. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. There are no 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 disabling of end events associated with this memory descriptor. Values for this argument can be constructed using a bitwise OR of the following values:

- **PTL_MD_EVENT_DISABLE**  
  Specifies that this memory descriptor should not generate events.

- **PTL_MD_EVENT_SUCCESS_DISABLE**  
  Specifies that this memory descriptor should not generate events that indicate success. This is useful in scenarios where the application does not need normal 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_UNORDERED**  
  Indicate to the Portals implementation that messages sent from this memory descriptor do not have to arrive at the target in order.

- **PTL_MD_REMOTE_FAILURE_DISABLE**  
  Indicate to the Portals implementation that failures requiring notification from the target should not be delivered to the local application. This prevents the local events (e.g. PTL_EVENT_SEND) from having to wait for a round-trip notification before delivery.

- **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.

*eq_handle*  
The event queue handle used to log the operations performed on the memory region. If this argument is PTL_EQ_NONE, operations performed on this memory descriptor are not logged.

*ct_handle*  
A handle for counting type 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.

```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
- `iov_len`: The length (in bytes) of the vector element

**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.

**Implementation Note 16**: Support of I/O Vector Type and Offset

The implementation is required to support the mixing of the `ptl_iovec_t` type with offsets (local and remote); however, it will be difficult to make this perform well in the general case. The correct behavior in this scenario is to treat the region described by the `ptl_iovec_t` type as if it were a single contiguous region. In some cases, this may require walking the entire scatter/gather list to find the correct location for depositing the data.

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, 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.
void *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.

void *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**

<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_NI_INVALID</td>
<td>Indicates that <strong>ni_handle</strong> is not a valid network interface handle.</td>
</tr>
<tr>
<td>PTL_MD_ILLEGAL</td>
<td>Indicates that <strong>md</strong> is not a legal memory descriptor. This may happen because the memory region defined in <strong>md</strong> is invalid or because the network interface associated with the <strong>eq_handle</strong> or the <strong>ct_handle</strong> in <strong>md</strong> is not the same as the network interface, <strong>ni_handle</strong>.</td>
</tr>
<tr>
<td>PTL_EQ_INVALID</td>
<td>Indicates that the event queue associated with <strong>md</strong> is not valid.</td>
</tr>
<tr>
<td>PTL_CT_INVALID</td>
<td>Indicates that the counting event associated with <strong>md</strong> is not valid.</td>
</tr>
<tr>
<td>PTL_NO_SPACE</td>
<td>Indicates that there is insufficient memory to allocate the memory descriptor.</td>
</tr>
<tr>
<td>PTL_SEGV</td>
<td>Indicate that <strong>md_handle</strong> is not a legal address.</td>
</tr>
</tbody>
</table>

### 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.

#### IMPLEMENTATION NOTE 17:

Unique memory descriptor handles

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

### Function Prototype for PtlMDRelease

```c
int PtlMDRelease(ptl_handler_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_MD_INVALID Indicates that md_handle is not a valid memory descriptor handle.
PTL_MD_IN_USE Indicates that md_handle has pending operations and cannot be released. See Figure 3.1 for when data structures are considered to be in use.

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.

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 an event (PTL_EVENT_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 event (PTL_EVENT_FREE) will be posted. A third type of event (PTL_EVENT_DROPPED) will be posted if a message arrives, finds no entries the priority list, and the overflow list is exhausted.

**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 and an PTL_EVENT_DROPPED to be posted. 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_UNLINK and replace the buffer. In many usage scenarios, however, the application may choose to have only persistent list entries 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 unlinked from an overflow list can be reused. It must note that it is no longer holding state in the buffer and post a PTL_EVENT_FREE event.

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.

**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 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.

To facilitate access control to both list entries and match list entries, the `ptl_ac_id_t` is defined as a union of a job ID and a user ID. A `ptl_ac_id_t` is attached to each list entry or match list entry to control which user (or which job, as selected by an option) can access the entry. Either field can specify a wildcard.

```c
typedef union
{
    ptl_jid_t jid;
    ptl_uid_t uid;
} ptl_ac_id_t;
```

### Members

- **uid**
  - The user identifier of the *initiator* that may access the associated list entry or match list entry. This may be set to `PTL_UID_ANY` to allow access by any user.

- **jid**
  - The job identifier of the *initiator* that may access the associated list entry or match list entry. This may be set to `PTL_JID_ANY` to allow access by any job.

```c
typedef struct
{
    void *start;
    ptl_size_t length;
    ptl_handle_ct_t ct_handle;
    ptl_ac_id_t ac_id;
    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. 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 type events associated with the memory region. If this argument is `PTL_CT_NONE`, operations performed on this list entry are not counted.
ac_id

Specifies either the user ID or job ID (as selected by the options) that may access this list entry. Either the user ID or job ID may be set to a wildcard (PTL_UID_ANY or PTL_JID_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 through the ni_fail_type in the PTL_EVENT_SEND event, unless the PTL_MD_REMOTE_FAILURE_DISABLE option is set.

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 disable events. 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 a permissions failure.

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 a permissions failure.

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 is triggered.

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 standard and counting type events. Acknowledgments are never sent for get operations. The data sent in the reply serves as an implicit acknowledgment.

PTL_IOVEC Specifies that the start argument is a pointer to an array of type ptl_iovec_t (Section 3.10.2) and the length argument is the length of the array. This allows for a scatter/gather capability for list entries. A scatter/gather list entry behaves exactly as a list entry that describes a single virtually contiguous region of memory. All other semantics are identical.

PTL_LE_EVENT_DISABLE Specifies that this list entry should not generate events.

PTL_LE_EVENT_SUCCESS_DISABLE Specifies that this list entry should not generate events that indicate success. This is useful in scenarios where the application does not need normal events, but does require failure information to enhance reliability.

PTL_LE_EVENT_OVER_DISABLE Specifies that this list entry should not generate overflow list events.

PTL_LE_EVENT_UNLINK_DISABLE Specifies that this list entry should not generate unlink (PTL_EVENT_UNLINK) or free (PTL_EVENT_FREE) events.

PTL_LE_EVENT_CT_GET Enable the counting of PTL_EVENT_GET events.
PTL_LE_EVENT_CT_PUT Enable the counting of PTL_EVENT_PUT events.
PTL_LE_EVENT_CT_PUT_OVERFLOW Enable the counting of PTL_EVENT_PUT_OVERFLOW events.
PTL_LE_EVENT_CT_ATOMIC Enable the counting of PTL_EVENT_ATOMIC events.
PTL_LE_EVENT_CT_ATOMIC_OVERFLOW Enable the counting of PTL_EVENT_ATOMIC_OVERFLOW events.

PTL_LE_AUTH_USE_JID Use job ID for authentication instead of user ID. By default, the user ID must match to allow a message to access a list entry.

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 list, the overflow list is checked to see if a message has arrived prior to posting the list entry. If so, a PTL_EVENT_PUT_OVERFLOW event is generated. No searching is performed when a list entry is posted to the overflow list.

typedef enum {
    PTL_PRIORITY_LIST, PTL_OVERFLOW, PTL_PROBE_ONLY
} ptl_list_t ;

LE List Types

PTL_PRIORITY_LIST The priority list associated with a portal table entry
PTL_OVERFLOW The overflow list associated with a portal table entry
PTL_PROBE_ONLY Do not attach to a list. Use the LE to probe the overflow list, without consuming an item in the list and without being attached anywhere.

Function Prototype for PtlLEAppend

int PtlLEAppend(ni_handle_t ni_handle, ptl_pt index_t pt_index, ptl_le_t le, ptl_list_t ptl_list, void *user_ptr, ptl_handle_le_t *le_handle);

Arguments

ni_handle input The interface handle to use.
pt_index input The portal table index where the list entry should be appended.
le input Provides initial values for the user-visible parts of a list entry. Other than its use for initialization, there is no linkage between this structure and the list entry maintained by the API.

ptl_list input Determines whether the list entry is appended to the priority list, appended to the overflow list, or simply queries 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 events associated with operations on this list entry.

le_handle output On successful return, this location will hold the newly created list entry handle.

Return Codes

PTL_OK Indicates success.
PTL_NI_INVALID Indicates that ni_handle is not a valid network interface handle.
PTL_NO_INIT Indicates that the portals API has not been successfully initialized.
PTL_PT_INDEX_INVALID Indicates that pt_index is not a valid portal table index.
PTL_NO_SPACE Indicates that there is insufficient memory to allocate the match list entry.
PTL_LE_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. This operation also releases any resources associated with the list entry. It is an error to use the list entry handle after calling PtlLEUnlink().

Function Prototype for PtlLEUnlink

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

Arguments

le_handle input The list entry handle to be unlinked.

Discussion: If this list entry has pending operations; e.g., an unfinished reply operation, then PtlLEUnlink() will return PTL_LE_IN_USE, and the list entry will not be unlinked. This essentially creates a race between the application retrying the unlink operation and a new operation arriving. This is believed to be reasonable as the application rarely wants to unlink an LE while new operations are arriving to it.

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 lookup or a search for the appropriate MPI Request.
3.12 Match List Entries and Matching Lists

Matching list entries add matching semantics to the basic list constructs. Each match list entry (ME) adds a set of match criteria to the basic memory region description in the list entry. The match criteria added can be used to reject incoming requests based on process identifier or the match bits provided in the request. 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.

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 an event (PTL_EVENT_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 event (PTL_EVENT_FREE) will be posted. A third type of event (PTL_EVENT_DROPPED) will be posted if a message arrives, does not match in the priority list, and the overflow list is exhausted.

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 and an PTL_EVENT_DROPPED to be posted. 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_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 in the buffer and post a PTL_EVENT_FREE event.

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.

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_size_t min_free;
  ptl_ac_id_t ac_id;
  unsigned int options;
  ptl_process_id_t match_id;
  ptl_match_bits_t match_bits;
  ptl_match_bits_t ignore_bits;
} 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. 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 type 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. This value is only used if the `PTL_ME_MIN_FREE` option is specified and `PTL_ME_MANAGE_LOCAL` is set.

`ac_id` Specifies either the user ID or job ID (as selected by the options) that may access this match list entry. Either the user ID or job ID may be set to a wildcard (`PTL_UID_ANY` or `PTL_JID_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 through the `ni_fail_type` in the `PTL_EVENT_SEND` event, unless the `PTL_MD_REMOTE_FAILURE_DISABLE` option is set.

`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 disable events. 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 a permissions failure.

**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 a permissions failure.

**Note:** It is not considered an error to have a match list entry that does not respond to either *put* or *get* operations: Every match list entry responds to *reply* operations. Nor is it 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_MANAGELOCAL**

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 can cause the match to fail. (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 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 type events. Acknowledgments are never sent for *get* operations. The data sent in the *reply* serves as an implicit acknowledgment.

**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_MIN_FREE**

Specifies that the *min_free* field in the match list entry is to be used. This option is only used if **PTL_ME_MANAGE_LOCAL** is set.
PTL_ME_EVENT_DISABLE Specifies that this match list entry should not generate events.
PTL_MEEvento SUCCESS_DISABLE Specifies that this match list entry should not generate events that indicate success. This is useful in scenarios where the application does not need normal 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 events (PTL_EVENT_PUT_OVERFLOW events).
PTL_ME_EVENT_UNLINK_DISABLE Specifies that this match list entry should not generate unlink (PTL_EVENT_UNLINK) or free (PTL_EVENT_FREE) events.
PTL_ME_EVENT_CT_GET Enable the counting of PTL_EVENT_GET events.
PTL_ME_EVENT_CT_PUT Enable the counting of PTL_EVENT_PUT events.
PTL_ME_EVENT_CT_PUT_OVERFLOW Enable the counting of PTL_EVENT_PUT_OVERFLOW events.
PTL_ME_EVENT_CT_ATOMIC Enable the counting of PTL_EVENT_ATOMIC events.
PTL_ME_EVENT_CT_ATOMIC_OVERFLOW Enable the counting of PTL_EVENT_ATOMIC_OVERFLOW events.
PTL_ME_AUTH_USE_JID Use job ID for authentication instead of user ID. By default, the user ID must match to allow a message to access a match list entry.

**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_id_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**: 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
```

### 3.12.2 PtlMEAppend

The PtlMEAppend() function creates a single match list entry. If PTL_PRIORITY_LIST or PTL_OVERFLOW 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 overflow list is searched to see if a matching message has arrived prior to posting the match list entry. If so, a PTL_EVENT_PUT_OVERFLOW event is generated. No searching is performed when a match list entry is posted to the overflow list.

If ptl_list is set to PTL_PROBE_ONLY, the overflow list is probed to support the MPI_Probe functionality. A probe of the overflow list will always generate a PTL_EVENT_PROBE event. If a matching message was found in the overflow list, PTL_NI_OK is returned in the event. Otherwise, the event indicates that the probe operation failed.
typedef enum {
    PTL_PRIORITY_LIST, PTL_OVERFLOW, PTL_PROBE_ONLY
} ptl_list_t;

ME List Types

PTL_PRIORITY_LIST The priority list associated with a portal table entry
PTL_OVERFLOW The overflow list associated with a portal table entry
PTL_PROBE_ONLY Do not attach to a list. Use the ME to probe the overflow list, without consuming an item in the list and without being attached anywhere.

Function Prototype for PtlMEAppend

```c
int PtlMEAppend(ptl_handle ni, ptl_pt_index_t pt_index, 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, appended to the overflow list, or simply queries 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 events associated with operations on this match list entry^2.
- **me_handle** output On successful return, this location will hold the newly created match list entry handle.

Return Codes

- **PTL_OK** Indicates success.
- **PTL NI INVALID** Indicates that ni_handle is not a valid network interface handle.

---

^2Tying 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 lookup or a search for the appropriate MPI Request.
PTL_NO_INIT Indicates that the portals API has not been successfully initialized.
PTL_PT_INDEX_INVALID Indicates that pt_index is not a valid portal table index.
PTL_PROCESS_INVALID Indicates that match_id in the match list entry is not a valid process identifier.
PTL_NO_SPACE Indicates that there is insufficient memory to allocate the match list entry.
PTL_ME_LIST_TOO_LONG Indicates that the resulting list is too long. The maximum length for a list is defined by the interface.

### IMPLEMENTATION NOTE 18:

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_PROCESS_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. This operation also releases any resources associated with the match list entry. It is an error to use the match list entry handle after calling PtlMEUnlink().

#### Function Prototype for PtlMEUnlink

```c
int PtlMEUnlink(ptl_handle_t me_handle);
```

#### Arguments

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

#### Discussion:
If this match list entry has pending operations; e.g., an unfinished reply operation, then PtlMEUnlink() will return PTL_ME_IN_USE, and the match list entry will not be unlinked. This essentially creates a race between the application retrying the unlink operation and a new operation arriving. This is believed to be reasonable as the application rarely wants to unlink an ME while new operations are arriving to it.

#### Return Codes

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

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

In addition to the `ptl_handle_eq_t` type, the portals API defines four types associated with events: The `ptl_event_kind_t` type defines the kinds of events that can be stored in an event queue. The `ptl_event_t` type defines the structure that is placed into event queues, while `ptl_initiator_event_t` and `ptl_target_event_t` types define sub-fields that hold the information associated with an event.

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

#### 3.13.1 Kinds of Events

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

```c
typedef enum {
    PTL_EVENT_GET,
    PTL_EVENT_PUT,
    PTL_EVENT_PUT_OVERFLOW,
    PTL_EVENT_ATOMIC,
    PTL_EVENT_REPLY,
    PTL_EVENT_SEND,
    PTL_EVENT_ACK,
    PTL_EVENT_UNLINK,
    PTL_EVENT_FREE,
    PTL_EVENT_DROPPED,
    PTL_EVENT_PROBE
} ptl_event_kind_t;
```

**Event types**

- **PTL_EVENT_GET**: A previously initiated get operation completed successfully.
- **PTL_EVENT_PUT**: A previously initiated put operation completed successfully. The underlying layers will not alter the memory (on behalf of this operation) once this event has been logged.
PTL_EVENT_PUT_OVERFLOW

A match list entry posted by PtlMEAppend() matched a message that has already arrived and is managed within the overflow list. All, some, or none of the message may have been captured in local memory as requested by the match list entry and described by the rlen and mlen fields in the event. The event will point to the start of the message in the memory region described by the match list entry from the overflow list, if any of the message was captured. When the rlen and mlen fields do not match (i.e. the message was truncated), the application is responsible for performing the remaining transfer. This typically occurs when the application has provided an overflow list entry designed to accept headers but not message bodies. The transfer is typically done by the initiator creating a match list entry using a unique set of bits and then placing the match bits in the hdr.data field. The target can then use the hdr.data field (along with other information in the event) to retrieve the message.

PTL_EVENT_ATOMIC

A previously initiated atomic operation completed successfully.

PTL_EVENT_REPLY

A previously initiated reply operation has completed successfully. This event is logged after the data (if any) from the reply has been written into the memory descriptor.

PTL_EVENT_SEND

A previously initiated send operation has completed. This event is logged after the entire buffer has been sent and it is safe to reuse the buffer.

PTL_EVENT_ACK

An acknowledgment was received. This event is logged when the acknowledgment is received.

PTL_EVENT_DROPPED

A message arrived, but did not match in the priority list and the overflow list was out of space. Thus, the message had to be dropped.

PTL_EVENT_UNLINK

A match list entry was unlinked (Section 3.12.2).

PTL_EVENT_FREE

A match list entry in the overflow list that was previously unlinked is now free to be reused by the application (Section 3.12.2).

PTL_EVENT_PROBE

A previously initiated PtlMEAppend() call that was set to “probe only” completed. If a match 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 a PTL_EVENT_PUT_OVERFLOW event. Otherwise, a failure is recorded in the ni.fail_type field, the user.ptr is filled in correctly, and the other fields are undefined.

**IMPLEMENTATION NOTE 19:** Overflow Events

An implementation is not required to deliver overflow events, if it can prevent an overflow from happening. For example, if an implementation used rendezvous at the lowest level, it could always choose to deliver the message into the memory of the ME that would eventually be posted.

### 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. 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 notes 20 and 21.)

**IMPLEMENTATION NOTE 20:**
Pending operations and buffer modifications

Figure 3.1(a) indicates that the memory descriptor is in use from the operation initiation until PTL\_EVENT\_ACK. However, the initiator is free to modify the buffer the memory descriptor describes after the PTL\_EVENT\_SEND event. Also see implementation note 21.

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.

If during any of the operations shown in the diagrams of Figure 3.1, a match list entry is unlinked, then a PTL\_EVENT\_UNLINK event is generated on the target where it was unlinked. This is not shown in the diagrams. None of these events are generated if the memory descriptor or match list entry has no event queue attached to it (see the description of PTL\_EQ\_NONE on page 45 of Section 3.10.1). The various types of events can be disabled individually. (See the description of PTL\_ME\_EVENT\_DISABLE and PTL\_ME\_EVENT\_UNLINK\_DISABLE on page 62, also in Section 3.12.1.)
**Note 21:** Pending operations and acknowledgment

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

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

---

Table 3.2 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, where *(initiator or target)* they can occur and the meaning of those events.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>initiator</th>
<th>target</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_EVENT_GET</td>
<td>*</td>
<td>•</td>
<td>Data was &quot;pulled&quot; from a local match list entry.</td>
</tr>
<tr>
<td>PTL_EVENT_PUT</td>
<td>*</td>
<td>•</td>
<td>A put matched a previously posed match list entry.</td>
</tr>
<tr>
<td>PTL_EVENT_PUT_OVERFLOW</td>
<td>*</td>
<td>•</td>
<td>A previous put arrived and matched a new match list entry.</td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC</td>
<td>*</td>
<td>•</td>
<td>Data was manipulated atomically in a local match list entry.</td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC_OVERFLOW</td>
<td>*</td>
<td>•</td>
<td>A previous atomic operation arrived and matched a new match list entry.</td>
</tr>
<tr>
<td>PTL_EVENT_REPLY</td>
<td>•</td>
<td>•</td>
<td>Data arrived at a local memory descriptor because of a local get or atomic operation.</td>
</tr>
<tr>
<td>PTL_EVENT_SEND</td>
<td>•</td>
<td>•</td>
<td>Data left a local memory descriptor because of a local put or atomic operation.</td>
</tr>
<tr>
<td>PTL_EVENT_ACK</td>
<td>•</td>
<td>•</td>
<td>An acknowledgment has arrived.</td>
</tr>
<tr>
<td>PTL_EVENT_DROPPED</td>
<td>•</td>
<td>•</td>
<td>A message was dropped because the overflow list was out of space.</td>
</tr>
<tr>
<td>PTL_EVENT_PT_DISABLED</td>
<td>•</td>
<td>•</td>
<td>A portal table entry has been disabled due to resource exhaustion.</td>
</tr>
<tr>
<td>PTL_EVENT_UNLINK</td>
<td>•</td>
<td>•</td>
<td>A local match list entry has been unlinked.</td>
</tr>
<tr>
<td>PTL_EVENT_FREE</td>
<td>•</td>
<td>•</td>
<td>A local match list entry that was posted to the overflow list and was previously is now free for reuse by the application (applies to overflow lists).</td>
</tr>
<tr>
<td>PTL_EVENT_PROBE</td>
<td>•</td>
<td>•</td>
<td>A PtrMEAppend() that was set to probe only completed</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 \texttt{ptl\_ni\_fail\_t} to define specific information regarding the failure of the operation and record this information in the \texttt{ni\_fail\_type} field of an event. The constant \texttt{PTL\_NI\_OK} should be used in successful end events to indicate that there has been no failure. In turn, the constant \texttt{PTL\_NI\_UNDELIVERABLE} should indicate a system failure that prevents message delivery. The constant \texttt{PTL\_NI\_FLOW\_CTRL} should indicate that the remote node has exhausted its resources and has enabled flow control and dropped this message. The constant \texttt{PTL\_NI\_PERM\_VIOLATION} should indicate that the remote Portals addressing has indicated a permissions violation for this message. The latter two error types require the stateful delivery of information from the target, and can be disabled by using \texttt{PTL\_MD\_REMOTE\_FAILURE\_DISABLE} in the MD options (see Section 3.10).

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 22:</th>
<th>Completion of portals operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Portals guarantees that every operation started will finish with an event if events are not disabled. While this document cannot enforce or recommend a suitable time, a quality implementation will keep the amount of time between an operation initiation and a corresponding event as short as possible. That includes operations that do not complete successfully. Timeouts of underlying protocols should be chosen accordingly.</td>
</tr>
</tbody>
</table>

3.13.4 The Event Queue Types

An event queue contains \texttt{ptl\_event\_t} structures, which contain a \texttt{type} and a union of the target specific event structure and the initiator specific event structure.

```c
typedef struct {
    ptl_event_kind_t type;
    union {
        ptl_target_event_t tevent;
        ptl_initiator_event_t ievent;
    } event;
} ptl_event_t;
```

Members

- **type**: Indicates the type of the event.
- **event**: Contains the event information.
An operation on the target needs information about the local match list entry modified, the initiator of the operation and the operation itself. These fields are included in a structure:

```
typedef struct {
  ptl_process_id_t initiator; /* nid, pid or rank */
  ptl_pt_index_t pt_index;
  ptl_uid_t uid;
  ptl_jid_t jid;
  ptl_match_bits_t match_bits;
  ptl_size_t rlength;
  ptl_size_t mlength;
  ptl_size_t remote_offset;
  void *start;
  void *user_ptr;
  ptl_hdr_data_t hdr_data;
  ptl_ni_fail_t ni_fail_type;
  ptl_op_t atomic_operation;
  ptl_datatype_t atomic_type;
  volatile ptl_seq_t sequence;
} ptl_target_event_t;
```

### Members

**initiator**
- The identifier of the initiator (`ptl_process_id_t`).

**pt_index**
- The portal table index where the message arrived.

**uid**
- The user identifier of the initiator.

**jid**
- The job identifier of the initiator. May be `PTL_JID_NONE` in implementations that do not support job identifiers.

**match_bits**
- The match bits specified by the initiator.

**rlength**
- The length (in bytes) specified in the request.

**mlength**
- The length (in bytes) of the data that was manipulated by the operation.
  - For truncated operations, the manipulated length will be the number of bytes specified by the memory descriptor operation (possibly with an offset). For all other operations, the manipulated length will be the length of the requested operation.

**remote_offset**
- The offset requested by the initiator.

**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 match 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).
  - When the `PtlMEAppend()` 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**
- A user-specified value that is associated with each command that can generate an event. The `user_ptr` is placed in the event. 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).

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 event corresponds to an atomic operation, this indicates the atomic operation that was performed

atomic_type If this event corresponds to an atomic operation, this indicates the data type of the atomic operation that was performed

sequence The sequence number for this event. Sequence numbers are unique to each event.

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. These fields are provided by a much smaller event structure:

```c
typedef struct {
    intptr_t mlength;
    intptr_t offset;
    intptr_t* user_ptr;
    intptr_t ni_fail_type;
    volatile intptr_t* sequence;
} pti_initiator_event_t;
```

Members

mlength, ni_fail_type, sequence, user_ptr See the discussion of pti_target_event_t.

offset The displacement (in bytes) into the memory region that the operation used. The offset can be determined by the operation (Section 3.15) for a remote managed memory descriptor or by the local memory descriptor (Section 3.10). The offset and the length of the memory descriptor can be used to determine if min_free has been exceeded.

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

3.13.5 PtlEQAlloc

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

**Function Prototype for PtlEQAlloc**

```c
int PtlEQAlloc(ptl_handle ni_t *ni_handle,
                 intptr_t count,
                 ptr_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 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 events. The event queue is circular. If flow control is not enabled on the portal table entry (Sections 3.6.1 and 2.3), 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_NI_INVALID** | Indicates that ni.handle is not a valid network interface handle.
- **PTL_NO_SPACE** | Indicates that there is insufficient memory to allocate the event queue.
- **PTL_SEGV** | Indicates that eq.handle is not a legal address.

**IMPLEMENTATION NOTE 23:** Location of event queue
The event queue is designed to reside in user space. High-performance implementations can be designed so they only need to write to the event queue but never have to read from it. This limits the number of protection boundary crossings to update the event queue. However, implementors are free to place the event queue anywhere they like; inside the kernel or the NIC for example.

**IMPLEMENTATION NOTE 24:** Size of event queue and reserved space
Because flow control may be enabled on the portal table entries that this EQ is attached to, the implementation should insure that the space allocated for the EQ is large enough to hold the requested number of 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 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 match list 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_EQ_INVALID` Indicates that `eq_handle` is not a valid event queue handle.

3.13.7 PtlEQGet

The `PtlEQGet()` function is a nonblocking 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.

Return Codes

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

The PtlEQWait() function can be used to block the calling process or thread until there is an 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,
              ptt_event_t *event);
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq_handle</td>
<td>input</td>
<td>The event queue handle to wait on. The calling process (thread) will be blocked until the event queue is not empty.</td>
</tr>
<tr>
<td>event</td>
<td>output</td>
<td>On successful return, this location will hold the values associated with the next event in the event queue.</td>
</tr>
</tbody>
</table>

Return Codes

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

3.13.9 PtlEQPoll

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

If PtlEQPoll() returns success, the corresponding event is consumed. PtlEQPoll() provides a timeout to allow applications to poll, block for a fixed period, or block indefinitely. PtlEQPoll() is sufficiently general to implement both PtlEQGet() and PtlEQWait(), but these functions have been retained in the API for backward compatibility.

**IMPLEMENTATION NOTE 25:**

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

Arguments

- `eq_handles`: input, An array of event queue handles. All the handles must refer to the same interface.
- `size`: input, Length of the array.
- `timeout`: input, Time in milliseconds to wait for an event to occur on one of the event queue handles. The constant `PTL_TIME_FOREVER` can be used to indicate an infinite timeout.
- `event`: output, On successful return (PTL_OK or PTL_EQ_DROPPED), this location will hold the values associated with the next event in the event queue.
- `which`: output, On successful return, this location will contain the index 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 event between this event and the last event obtained from the event queue indicated by `which` has been dropped due to limited space in the event queue.
- **PTL_NO_INIT**: Indicates that the portals API has not been successfully initialized.
- **PTL_EQ_INVALID**: Indicates that one or more of the event queue handles is not valid; e.g., not all handles in `eq_handles` are on the same network interface.
- **PTL_SEGV**: Indicates that `event` or `which` is not a legal address.
- **PTL_EQ_EMPTY**: Indicates that the timeout has been reached and all of the event queues are empty.

**Implementation Note 26:** Macros using `PtlEQPoll()`

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

**Implementation Note 27:** Filling in the `ptl_event_t` and `ptl_target_event_t` structures

All of the members of the `ptl_event_t` structure (and corresponding `ptl_initiator_event_t` or `ptl_target_event_t` sub-field) returned from `PtlEQGet()`, `PtlEQWait()`, and `PtlEQPoll()` must be filled in with valid information. An implementation may not leave any field in an event unset.
3.14 Lightweight “Counting” Events

Standard 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. Counting events can be set to count either the total number of operations or the number of bytes transferred for the associated operations.

Counting events mirror standard 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 `PtlICTAlloc()`, queried through a `PtlICTGet()` or `PtlICTWait()`, set through a `PtlICTSet()`, incremented through a `PtlICTInc()`, and freed through a `PtlICTFree()`. To mirror the failure semantics of the standard events, one variable counts the successful events and the second variable counts the events that failed.

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 28: Counting Event Handles</th>
</tr>
</thead>
<tbody>
<tr>
<td>A high performance implementation could choose to make a <code>ptl.handle_ct_t</code> 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).</td>
</tr>
</tbody>
</table>

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.

```c
typedef struct {
    ptl.size_t success;
    ptl.size_t failure;
} ptt.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; thus, an allocation routine is provided. A newly allocated count is initialized to zero.

```c
typedef enum {
    PTL_CT_OPERATION, PTL_CT_BYTE
} ptl_ct_type_t;
```

Function Prototype for PtlCTAlloc

```
int PtlCTAlloc(ptl_handle ni, ptl_ct_type_t ct_type, ptl_handle ct, *ct_handle);
```

Arguments

- **ni**.handle: input The interface handle with which the counting event will be associated.
- **ct**.type: input A selection between counting operations and counting bytes.
- **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_NI_INVALID**: Indicates that ni.handle is not a valid network interface handle.
- **PTL_NO_SPACE**: Indicates that there is insufficient memory to allocate the counting event.
- **PTL_SEGV**: Indicates that ct.handle is not a legal address.

**IMPLEMENTATION NOTE 29:** 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.

**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_CT_INVALID**  
  Indicates that `ct_handle` is not a valid counting event handle.

3.14.4  PtlCTGet

The **PtlCTGet()** function is used to obtain the current value of a counting event.

**Function Prototype for PtlCTGet**

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

**Arguments**

- `ct_handle`  
  **input**  
  The counting event handle.
- `event`  
  **output**  
  On successful return, this location will hold the current value associated with the counting event.

**Return Codes**

- **PTL_OK**  
  Indicates success.
- **PTL_NO_INIT**  
  Indicates that the portals API has not been successfully initialized.
- **PTL_CT_INVALID**  
  Indicates that `ct_handle` is not a valid counting event handle.
- **PTL_SEGV**  
  Indicates that `event` is not a legal address.
3.14.5 PtlCTWait

The **PtlCTWait()** function is used to wait until the value of a counting event is equal to a test value.

**Function Prototype for PtlCTWait**

```
int PtlCTWait(ptl_handle_ct_t ct_handle ,
ptl_size_t test );
```

**Arguments**

- **ct_handle input** The counting event handle.
- **test input** On successful return, the sum of the success and failure fields of the counting event will be greater than or equal to this value.

**Return Codes**

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

3.14.6 PtlCTSet

Periodically, it is desirable to reinitialize or adjust the value of a counting event. The **PtlCTSet()** function is used to set the value of a counting event.

**Function Prototype for PtlCTSet**

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

**Arguments**

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

**Return Codes**

- **PTL_OK** Indicates success.
- **PTL_NO_INIT** Indicates that the portals API has not been successfully initialized.
- **PTL_CT_INVALID** Indicates that **ct_handle** is not a valid counting event handle.
3.14.7 PtlCTInc

In some scenarios, the counting event will need to be incremented by the application. This must be done atomically, so a functional interface is provided. The PtlCTInc() function is used to increment the value of a counting event.

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

```
int PtlCTInc(ptl_handle_t ct_handle, ptl_ct_event_t tcteventt, 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_CT_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 30:** 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

Values of the type **ptl_ack_req_t** are used to control whether an acknowledgment should be sent when the operation completes (i.e., when the data has been written to a match list entry of the target process). The value PTL_ACK_REQ requests an acknowledgment, the value PTL_NO_ACK_REQ requests that no acknowledgment should be generated, the value PTL_CT_ACK_REQ requests a simple counting acknowledgment, and the value PTL_OC_ACK_REQ requests an
operation completed acknowledgement. When a counting acknowledgment is requested, either PTL_CT_OPERATION or PTL_CT_BYTE can be set in the ct handle. If PTL_CT_OPERATION is set, the number of acknowledgments is counted. If PTL_CT_BYTE is set, the modified length (mlength) from the target is counted at the initiator. The operation completed acknowledgement is an acknowledgement that simply indicated that the operation has completed at the target. It does not indicate what was done with the message. The message may have been dropped due to a permission violation or may not have matched in the priority list or overflow list; however, the operation completed acknowledgement would still be sent. The operation completed acknowledgement is a subset of the counting acknowledgement with weaker semantics. That is, it is a counting type of acknowledgement, but it can only count operations.

```
typedef enum {
    PTL_ACK_REQ,
    PTL_NO_ACK_REQ,
    PTL_CT_ACK_REQ,
    PTL_OC_ACK_REQ
} ptl_ack_req_t;
```

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, when the send completes successfully, 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, while PTL_EVENT_PUT_OVERFLOW can be used on the target node when a message arrives before the corresponding match list entry (Figure 3.1).

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

The local (initiator) offset is used to determine the starting address of the memory region within the region specified by the memory descriptor and the length specifies the length of the region in bytes. It is an error for the local offset and length parameters to specify memory outside the memory described by the memory descriptor.

**Function Prototype for PtlPut**

```
int PtlPut (ptl_handle_md_t md_handle,
           ptl_size_t local_offset,
           ptl_size_t length,
           ptl_ack_req_t ack_req,
           ptl_process_id_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**
<table>
<thead>
<tr>
<th>Field</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 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).</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 to be sent.</td>
</tr>
<tr>
<td>ack_req</td>
<td>input</td>
<td>Controls whether an acknowledgment event is requested. Acknowledgments are only sent when they are requested by the initiating process and the memory descriptor has an event queue and the target memory descriptor enables them. Allowed constants: PTL_ACK_REQ, PTL_NO_ACK_REQ, PTL_CT_ACK_REQ, PTL_OC_ACK_REQ.</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 (only used when matching is enabled on the network interface).</td>
</tr>
<tr>
<td>remote_offset</td>
<td>input</td>
<td>The offset into the target memory descriptor (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>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 events associated with this put operation.</td>
</tr>
<tr>
<td>hdr_data</td>
<td>input</td>
<td>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.</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_MD_INVALID</td>
<td>Indicates that md_handle is not a valid memory descriptor.</td>
</tr>
<tr>
<td>PTL_PROCESS_INVALID</td>
<td>Indicates that target_id is not a valid process identifier.</td>
</tr>
</tbody>
</table>

### 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. 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.

---

3Tying 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 event by the MPI implementation without a table lookup or a search for the appropriate MPI Request.
Function Prototype for PtlGet

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

Arguments

- **md_handle**
  - input
  - The memory descriptor handle that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.

- **local_offset**
  - input
  - Offset from the start of the memory descriptor.

- **length**
  - input
  - Length of the memory region for the reply.

- **target_id**
  - 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.

- **user_ptr**
  - input
  - See the discussion for PtlPut().

- **remote_offset**
  - input
  - The offset into the target match list entry (used unless the target match list entry has the PTL_ME_MANAGE_LOCAL option set).

Return Codes

- **PTL_OK**
  - Indicates success.

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

- **PTL_MD_INVALID**
  - Indicates that md_handle is not a valid memory descriptor.

- **PTL_PROCESS_INVALID**
  - Indicates that target_id is not a valid process identifier.

### 3.15.4 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() or PtlFetchAtomic() is restricted to no more than max_atomic_size bytes. PtlSwap() operations can also be up to max_atomic_size bytes, except for PTL_CLASS and PTL_MSWAP operations, which are further restricted to 8 bytes (the length of the longest native data type) in all
implementations. 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 three 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. If data is sent from the target node, a PTL_EVENT_ATOMIC event is registered on the target node; and if data is returned from the target node, a PTL_EVENT_REPLY event is registered on the initiator node. 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.

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.

```c
typedef enum {
    PTL_MIN, PTL_MAX,
    PTL_SUM, PTL_PROD,
    PTL_LOR, PTL_LAND,
    PTL_BOR, PTL_BAND,
    PTL_LXOR, PTL_BXOR,
    PTL_SWAP, PTL_CSWAP, PTL_MSWAP
} ptl_op_t;
```

**Atomic Operations**

- **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_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.
typedef enum {
    PTL_CHAR, PTL_UCHAR,
    PTL_SHORT, PTL_U SHORT,
    PTL_INT, PTL_UINT,
    PTL_LONG, PTL_ULONG,
    PTL_FLOAT, PTL_DOUBLE
} ptl datatype_t;

Atomic Datatypes

PTL_CHAR 8-bit signed integer
PTL_UCHAR 8-bit unsigned integer
PTL_SHORT 16-bit signed integer
PTL_USHORT 16-bit unsigned integer
PTL_INT 32-bit signed integer
PTL_UINT 32-bit unsigned integer
PTL_LONG 64-bit signed integer
PTL_ULONG 64-bit unsigned integer
PTL_FLOAT 32-bit floating-point number
PTL_DOUBLE 64-bit floating-point number

3.15.5 PtlAtomic

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 id_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.

`local_offset` input Offset from the start of the memory descriptor referenced by the `md_handle` to use for transmitted data.
Length of the memory region to be sent and/or received.

Controls whether an acknowledgment event is requested. Acknowledgments are only sent when they are requested by the initiating process and the memory descriptor has an event queue and the target memory descriptor enables them. Allowed constants: PTL_ACK_REQ, PTL_NO_ACK_REQ, PTL_CT_ACK_REQ, PTL_OC_ACK_REQ.

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 descriptor (used unless the target memory descriptor 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.

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 md_handle is not a valid memory descriptor.

Indicates that target_id is not a valid process identifier.

Function Prototype for PtlFetchAtomic

```c
int PtlFetchAtomic(ptrl_handle_md_t get_md_handle,
        ptrl_size_t local_get_offset ,
        ptrl_handle_md_t put_md_handle,
        ptrl_size_t local_put_offset ,
        ptrl_size_t length ,
        ptrl_process_id_t target_id ,
        ptrl_pt_index_t pt_index ,
        ptrl_match_bits_t match_bits ,
        ptrl_size_t remote_offset ,
        void *user_ptr ,
        ptrl_hdr_data_t hdr_data,
        ptrl_op_t operation ,
        ptrl_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.

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.

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.

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 descriptor (used unless the target memory descriptor 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_MD_INVALID  Indicates that put_md_handle or get_md_handle is not a valid memory descriptor.

PTL_PROCESS_INVALID  Indicates that target_id is not a valid process identifier.
3.15.7 PtlSwap

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_process_id_t target_id ,
            ptl_pt_index_t plt_index ,
            ptl_match_bits_t match_bits ,
            ptl_size_t remote_offset ,
            void *user_ptr ,
            ptl_hdr_data_t hdr_data ,
            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.

- **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.

- **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.

- **target_id**  
  *input*  
  A process identifier for the `target` process.

- **plt_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 descriptor (used unless the target memory descriptor 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.

- **operand**  
  *input*  
  A pointer to the data to be used for the `PTL_CSWAP` and `PTL_MSWAP` operations (ignored for other operations). The data pointed to is of the type specified by the `datatype` argument and must be included in the message.

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

- **datatype**  
  *input*  
  The type of data being operated on at the initiator and target.
Return Codes

<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_MD_INVALID</td>
<td>Indicates that <code>put_md_handle</code> or <code>get_md_handle</code> is not a valid memory descriptor.</td>
</tr>
<tr>
<td>PTL_PROCESS_INVALID</td>
<td>Indicates that <code>target_id</code> is not a valid process identifier.</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 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.

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

### IMPLEMENTATION NOTE 31: 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 32: 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 33: 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_id_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` (input) See description in Section 3.15.2.
- `trig_ct_handle` (input) Handle used for triggering the operation.
- `threshold` (input) Threshold at which the operation triggers.

Return Codes

- PTL_OK Indicates success.
- PTL_NO_INIT Indicates that the portals API has not been successfully initialized.
- PTL_MD_INVALID Indicates that `md_handle` is not a valid memory descriptor.
- PTL_PROCESS_INVALID Indicates that `target_id` is not a valid process identifier.
- PTL_CT_INVALID Indicates that `ct_handle` is not a valid counting event handle.

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_id_t target_id ,
                      ptl_pt_index_t pt_index ,
                      ptl_match_bits_t match_bits ,
                      void *user_ptr ,
                      ptl_size_t remote_offset ,
                      ptl_handle ct_t ct_handle ,
                      ptl_size_t threshold );
```

Arguments

- `md_handle`: Input. See the discussion for `PtlGet()`.
- `target_id`: Input. Handle used for triggering the operation.
- `pt_index`: Input. Threshold at which the operation triggers.
- `user_ptr`: Input. Indicates that the portals API has not been successfully initialized.
- `remote_offset`: Input. Indicates that `md_handle` is not a valid memory descriptor.
- `local_offset`: Input. Indicates that `target_id` is not a valid process identifier.
- `length`: Input. Indicates that `ct_handle` is not a valid counting event handle.

Return Codes

- `PTL_OK`
- `PTL_NO_INIT`
- `PTL_MD_INVALID`
- `PTL_PROCESS_INVALID`
- `PTL_CT_INVALID`

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()`), 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_id_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_MD_INVALID**
  - Indicates that `put_md_handle` or `get_md_handle` is not a valid memory descriptor.
- **PTL_PROCESS_INVALID**
  - Indicates that `target_id` is not a valid process identifier.
- **PTL_CT_INVALID**
  - Indicates that `ct_handle` is not a valid counting event handle.
### 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_id_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` - Get memory descriptor handle.
- `local_get_offset` - Local offset for get operation.
- `put_md_handle` - Put memory descriptor handle.
- `local_put_offset` - Local offset for put operation.
- `length` - Data length.
- `target_id` - Target process identifier.
- `pt_index` - Portal index.
- `match_bits` - Match bits.
- `remote_offset` - Remote offset.
- `user_ptr` - User pointer.
- `hdr_data` - Header data.
- `operation` - Operation type.
- `datatype` - Data type.
- `trig_ct_handle` - Triggering counting event handle.
- `threshold` - Threshold.

**Input**

- See the discussion of `PtlFetchAtomic()`.

**Return Codes**

- **PTL_OK** - Indicates success.
- **PTL_NO_INIT** - Indicates that the portals API has not been successfully initialized.
- **PTL_MD_INVALID** - Indicates that `put_md_handle` or `get_md_handle` is not a valid memory descriptor.
- **PTL_PROCESS_INVALID** - Indicates that `target_id` is not a valid process identifier.
- **PTL_CT_INVALID** - Indicates that `ct_handle` is not a valid counting event handle.
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_id_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,
void *operand,
ptl_op_t operation,
ptl_datatype_t datatype,
ptl_handle_ct_t trig_ct_handle,
ptl_size_t threshold );
```

Arguments

- `get_md_handle`, `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`

  input  See the discussion of PtlSwap().

- `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_MD_INVALID**  Indicates that `put_md_handle` or `get_md_handle` is not a valid memory descriptor.
- **PTL_PROCESS_INVALID**  Indicates that `target_id` is not a valid process identifier.
- **PTL_CT_INVALID**  Indicates that `ct_handle` is not a valid counting event handle.
3.16.6 PtlTriggeredCTInc

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

Function Prototype for PtlTriggeredCTInc

| int PtlTriggeredCTInc (ptr_handle_ct_t ct.handle, ptr_size_t increment, ptr_handle_ct_t trig_ct.handle, ptr_size_t threshold); |

Arguments

- **ct.handle**, **increment** input See the discussion of PtlCTInc().
- **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_CT_INVALID** Indicates that ct.handle is not a valid counting event handle.

3.17 Operations on Handles

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

3.17.1 PtlHandleIsEqual

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

Function Prototype for PtlHandleIsEqual

| PtlHandleIsEqual (ptr_handle_any_t handle1, ptr_handle_any_t handle2); |

Arguments
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.

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

Return Codes

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

3.18 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 3.3 presents a summary of the types defined by the portals API. The first column in this table gives the type name, the second column gives a brief description of the type, the third column identifies the section where the type is defined, and the fourth column lists the functions that have arguments of this type.
Table 3.3. Portals Data Types: Data Types Defined by the Portals API.

<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 types</td>
<td>3.15.2</td>
<td>PtlPut(), PtlAtomic(), PtlTriggeredPut(), PtlTriggeredAtomic()</td>
</tr>
<tr>
<td>ptl_ct_type_t</td>
<td>counting event type</td>
<td>3.14.2</td>
<td>PtlCTAlloc()</td>
</tr>
<tr>
<td>ptl_event_t</td>
<td>counting event structure</td>
<td>3.14.2</td>
<td>PtlCTAlloc()</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_initiator_event_t</td>
<td>event queue entry</td>
<td>3.13.4</td>
<td>PtlEQGet(), PtlEQWait(), PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_target_event_t</td>
<td>target event information</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>3.2.2</td>
<td>PtlNIHandle(), PtlHandleIsEqual()</td>
</tr>
<tr>
<td>ptl_handle_eq_t</td>
<td>event queue handles</td>
<td>3.2.2</td>
<td>PtlEQAlloc(), PtlEQFree(), PtlEQGet(), PtlEQWait(), PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_handle_md_t</td>
<td>memory descriptor handles</td>
<td>3.2.2</td>
<td>PtlMDRelease(), PtlMEAppend(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()</td>
</tr>
<tr>
<td>ptl_handle_me_t</td>
<td>match list entry handles</td>
<td>3.2.2</td>
<td>PtlMEAppend(), PtlMEUnlink()</td>
</tr>
<tr>
<td>ptl_handle_nc_t</td>
<td>network interface handles</td>
<td>3.2.2</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>3.2.5</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>ptl_jid_t</td>
<td>job identifier</td>
<td>3.2.6</td>
<td>PtlGetJid()</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>3.2.4</td>
<td>PtlMEAppend(), 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()</td>
</tr>
<tr>
<td>ptl nid_t</td>
<td>node identifiers</td>
<td>3.2.6</td>
<td>PtlGetId()</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>3.5.1</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>ptl pid_t</td>
<td>process identifier</td>
<td>3.2.6</td>
<td>PtlGetId()</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_pt_index_t</td>
<td>portal table indexes</td>
<td>3.2.3</td>
<td>PtlMEAppend(), PtlPTAlloc(), PtlPTFree(), PtlPTEnable(), PtlPTDisable(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetchAtomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()</td>
</tr>
<tr>
<td>ptl_rank_t</td>
<td>rank within job</td>
<td>3.2.6</td>
<td>PtlGetId()</td>
</tr>
<tr>
<td>ptl_seq_t</td>
<td>event sequence number</td>
<td>3.13.4</td>
<td>PtlEQGet(), PtlEQWait(), PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_size_t</td>
<td>sizes</td>
<td>3.2.1</td>
<td>PtlEQAlloc(), PtlPut(), PtlGet(), PtlAtomic(), PtlFetch Atomic(), PtlSwap(), PtlTriggeredPut(), PtlTriggeredGet(), PtlTriggeredAtomic(), PtlTriggeredFetchAtomic(), PtlTriggeredSwap()</td>
</tr>
<tr>
<td>ptl_sr_index_t</td>
<td>status register indexes</td>
<td>3.2.7</td>
<td>PtlNIStatus()</td>
</tr>
<tr>
<td>ptl_sr_value_t</td>
<td>status register values</td>
<td>3.2.7</td>
<td>PtlNIStatus()</td>
</tr>
<tr>
<td>ptl_time_t</td>
<td>time in milliseconds</td>
<td>3.13.9</td>
<td>PtlEQPoll()</td>
</tr>
<tr>
<td>ptl_uid_t</td>
<td>user identifier</td>
<td>3.2.6</td>
<td>PtlGetUid()</td>
</tr>
</tbody>
</table>

Table 3.4 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 3.4. 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>PtlCTFree()</td>
<td>free a counting event</td>
<td>3.14.3</td>
</tr>
<tr>
<td>PtlCTInc()</td>
<td>increment a counting event by a certain value</td>
<td>3.14.7</td>
</tr>
<tr>
<td>PtlCTGet()</td>
<td>get the current value of a counting event</td>
<td>3.14.4</td>
</tr>
<tr>
<td>PtlCTWait()</td>
<td>wait for a counting event to reach a certain value</td>
<td>3.14.5</td>
</tr>
<tr>
<td>PtlCTSet()</td>
<td>set a counting event to a certain value</td>
<td>3.14.6</td>
</tr>
<tr>
<td>PtlEQAlloc()</td>
<td>create an event queue</td>
<td>3.13.5</td>
</tr>
<tr>
<td>PtlEQFree()</td>
<td>release the resources for an event queue</td>
<td>3.13.6</td>
</tr>
<tr>
<td>PtlEQGet()</td>
<td>get the next event from an event queue</td>
<td>3.13.7</td>
</tr>
<tr>
<td>PtlEQPoll()</td>
<td>poll for a new event on multiple event queues</td>
<td>3.13.9</td>
</tr>
<tr>
<td>PtlEQWait()</td>
<td>wait for a new event in an event queue</td>
<td>3.13.8</td>
</tr>
<tr>
<td>PtlFini()</td>
<td>shut down the portals API</td>
<td>3.4.2</td>
</tr>
<tr>
<td>PtlGet()</td>
<td>perform a get operation</td>
<td>3.15.3</td>
</tr>
<tr>
<td>PtlGetId()</td>
<td>get the identifier for the current process</td>
<td>3.8.2</td>
</tr>
</tbody>
</table>

continued on next page
Table 3.5 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>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_CT_INVALID</td>
<td>invalid counting event handle</td>
<td>PtlCTFree(), PtlCTGet(), PtlCTWait()</td>
</tr>
<tr>
<td>PTL_EQ_DROPPED</td>
<td>at least one event has been dropped</td>
<td>PtlEQGet(), PtlEQWait()</td>
</tr>
<tr>
<td>PTL_EQ_EMPTY</td>
<td>no events available in an event queue</td>
<td>PtlEQGet()</td>
</tr>
<tr>
<td>PTL_EQ_INVALID</td>
<td>invalid event queue handle</td>
<td>PtlEQFree(), PtlEQGet()</td>
</tr>
<tr>
<td>PTL_FAIL</td>
<td>error during initialization or cleanup</td>
<td>PtlInit(), PtlFini()</td>
</tr>
<tr>
<td>PTL_HANDLE_INVALID</td>
<td>invalid handle</td>
<td>PtlNIHandle()</td>
</tr>
<tr>
<td>PTL_IFACE_INVALID</td>
<td>initialization of an invalid interface</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>PTL_MD_ILLEGAL</td>
<td>illegal memory descriptor values</td>
<td>PtlMDRelease(), PtlMDBind()</td>
</tr>
<tr>
<td>PTL_MD_IN_USE</td>
<td>memory descriptor has pending operations</td>
<td>PtlMDRelease()</td>
</tr>
</tbody>
</table>

Table 3.5. Portals Return Codes: Function Return Codes for the Portals API.
Table 3.6 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>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTL_MD_INVALID</td>
<td>invalid memory descriptor handle</td>
<td>PtlMDRelease()</td>
</tr>
<tr>
<td>PTL_ME_IN_USE</td>
<td>ME has pending operations</td>
<td>PtlMEUnlink()</td>
</tr>
<tr>
<td>PTL_ME_INVALID</td>
<td>invalid match list entry handle</td>
<td>PtlMEAppend()</td>
</tr>
<tr>
<td>PTL_ME_LIST_TOO_LONG</td>
<td>match list entry too long</td>
<td>PtlMEAppend()</td>
</tr>
<tr>
<td>PTL_NI_INVALID</td>
<td>invalid network interface handle</td>
<td>PtlNIFini(), PtlMDBind(), PtlEQAlloc()</td>
</tr>
<tr>
<td>PTL_NI_NOT_LOGICAL</td>
<td>not a logically addressed network interface handle</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>PTL_NO_INIT</td>
<td>uninitialized API</td>
<td>all, except PtlInit()</td>
</tr>
<tr>
<td>PTL_NO_SPACE</td>
<td>insufficient memory</td>
<td>PtlNIFini(), PtlMDBind(), PtlEQAlloc(), PtlMEAppend()</td>
</tr>
<tr>
<td>PTL_OK</td>
<td>success</td>
<td>all</td>
</tr>
<tr>
<td>PTL_PID_INVALID</td>
<td>invalid pid</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>PTL_PID_INUSE</td>
<td>pid is in use</td>
<td>PtlNIInit()</td>
</tr>
<tr>
<td>PTL_PROCESS_INVALID</td>
<td>invalid process identifier</td>
<td>PtlNIInit(), PtlMEAppend(), PtlPut(), PtlGet()</td>
</tr>
<tr>
<td>PTL_PT_FULL</td>
<td>portal table is full</td>
<td>PtlPTAlloc()</td>
</tr>
<tr>
<td>PTL_PT_EQ_NEEDED</td>
<td>EQ must be attached when flow control is enabled</td>
<td>PtlPTAlloc()</td>
</tr>
<tr>
<td>PTL_PT_INDEX_INVALID</td>
<td>invalid portal table index</td>
<td>PtlMEAppend(), PtlPTFree()</td>
</tr>
<tr>
<td>PTL_PT_INDEX_INVALID</td>
<td>invalid portal table index</td>
<td>PtlPTFree()</td>
</tr>
<tr>
<td>PTL_SR_INDEX_INVALID</td>
<td>invalid status register index</td>
<td>PtlNIInit(), PtlNISstatus(), PtlNHandle(), PtlMDBind(), PtlEQAlloc(), PtlEQGet(), PtlEQWait()</td>
</tr>
</tbody>
</table>

Table 3.6. 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_EVENT_GET</td>
<td>get event</td>
<td>ptr_event_kind_t</td>
<td>3.13.1, 3.15.3</td>
</tr>
<tr>
<td>PTL_EVENT_ATOMIC</td>
<td>atomic event</td>
<td>ptr_event_kind_t</td>
<td>3.13.1, 3.15.5</td>
</tr>
<tr>
<td>PTL_EVENT_DROPPED</td>
<td>overflow list exhaustion</td>
<td>ptr_event_kind_t</td>
<td>3.13.1</td>
</tr>
<tr>
<td>PTL_EVENT_PUT</td>
<td>put event</td>
<td>ptr_event_kind_t</td>
<td>3.13.1, 3.15.2</td>
</tr>
<tr>
<td>PTL_EVENT_PUT_OVERFLOW</td>
<td>put event overflow</td>
<td>ptr_event_kind_t</td>
<td>3.13.1, 3.15.2</td>
</tr>
<tr>
<td>PTL_EVENT_REPLY</td>
<td>reply event</td>
<td>ptr_event_kind_t</td>
<td>3.13.1, 3.15.3, 3.15.5</td>
</tr>
<tr>
<td>PTL_EVENT_SEND</td>
<td>send event</td>
<td>ptr_event_kind_t</td>
<td>3.13.1, 3.15.2, 3.15.5</td>
</tr>
<tr>
<td>PTL_EVENT_UNLINK</td>
<td>unlink event</td>
<td>ptr_event_kind_t</td>
<td>3.12.1, 3.12.3, 3.13.1</td>
</tr>
<tr>
<td>PTL_EVENT_FREE</td>
<td>free event</td>
<td>ptr_event_kind_t</td>
<td>3.12.1, 3.12.3, 3.13.1</td>
</tr>
<tr>
<td>PTL_EVENT_PT_DISABLED</td>
<td>portal table entry disabled event</td>
<td>ptr_event_kind_t</td>
<td>3.13.1, 3.12.1, 2.3</td>
</tr>
<tr>
<td>PTL_EVENT_PROBE</td>
<td>probe event</td>
<td>ptr_event_kind_t</td>
<td>3.12.1, 3.12.3, 3.13.1</td>
</tr>
<tr>
<td>PTL_IFACE_DEFAULT</td>
<td>default interface</td>
<td>ptr_interface_t</td>
<td>3.2.5</td>
</tr>
<tr>
<td>PTL_INVALID_HANDLE</td>
<td>invalid handle</td>
<td>ptr_handle_any_t</td>
<td>3.2.2, 3.17.1</td>
</tr>
<tr>
<td>PTL_JID_ANY</td>
<td>wildcard for job identifier</td>
<td>ptr_jid_t</td>
<td>3.9, 3.2.6, 3.11, 3.12</td>
</tr>
<tr>
<td>PTL_JID_NONE</td>
<td>job identifiers not supported for process</td>
<td></td>
<td></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_DISABLE</td>
<td>a flag to disable events</td>
<td>int</td>
<td>3.10.1</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_LE_AUTH_USE_JID</td>
<td>a flag to indicate that the job ID should be used for access control</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_DISABLE</td>
<td>a flag to disable events</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_GET</td>
<td>a flag to count get events</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_CT_PUT</td>
<td>a flag to count put events</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_CT_PUT_OVERFLOW</td>
<td>a flag to count “overflow” put events</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_CT_ATOMIC</td>
<td>a flag to count atomic events</td>
<td>int</td>
<td>3.11.1</td>
</tr>
<tr>
<td>PTL_LE_EVENT_CT_ATOMIC_OVERFLOW</td>
<td>a flag to count “overflow” atomic events</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_OP_GET</td>
<td>a flag to enable get operations</td>
<td>int</td>
<td>3.11.1, 4.2</td>
</tr>
<tr>
<td>PTL_LE_OP_PUT</td>
<td>a flag to enable put operations</td>
<td>int</td>
<td>3.11.1, 4.2</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_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_LE_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>3.11.1</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_AUTH_USE_JID</td>
<td>a flag to indicate that the job ID should be used for access control</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_DISABLE</td>
<td>a flag to disable events</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_GET</td>
<td>a flag to count get events</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_CT_PUT</td>
<td>a flag to count put events</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_CT_PUT_OVERFLOW</td>
<td>a flag to count “overflow” put events</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_CT_ATOMIC</td>
<td>a flag to count atomic events</td>
<td>int</td>
<td>3.12.1</td>
</tr>
<tr>
<td>PTL_ME_EVENT_CT_ATOMIC_OVERFLOW</td>
<td>a flag to count “overflow” atomic events</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_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_REMOTE_FAILURE_DISABLE</td>
<td>a flag to indicate that remote failures should not be delivered to the local EQ</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_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_MIN_FREE</td>
<td>use the min_free field in a match list entry</td>
<td>unsigned int</td>
<td>3.12.1</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</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</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</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>3.2.6, 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_FLOW_CTRL</td>
<td>message encountered a flow control condition</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>3.5.2</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>3.5.2</td>
</tr>
<tr>
<td>PTL_NI_LOGICAL</td>
<td>a flag to indicate that the network interface must provide logical addresses for network end-points</td>
<td>int</td>
<td>3.5.2</td>
</tr>
<tr>
<td>PTL_NI_PHYSICAL</td>
<td>a flag to indicate that the network interface must provide physical addresses for network end-points</td>
<td>int</td>
<td>3.5.2</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, 4.1</td>
</tr>
<tr>
<td>PTL_CT_OPERATION</td>
<td>a flag to indicate a counting event that counts operations specifies the overflow list attached to a portal table entry</td>
<td>ptl_ct_type_t</td>
<td>3.14.2</td>
</tr>
<tr>
<td>PTL_OVERFLOW</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>3.2.6, 3.5.2, 3.12.2, 3.12</td>
</tr>
<tr>
<td>PTL_PT_ANY</td>
<td>wildcard for portal table entry identifier fields</td>
<td>ptl_pt_index_t</td>
<td>3.6.1</td>
</tr>
<tr>
<td>PTL_PT_ONLY_USE_ONCE</td>
<td>a flag to indicate that the 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>3.6.1</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_PROBE_ONLY</td>
<td>specifies that the match list entry should not be attached, but should probe only</td>
<td>int</td>
<td>3.12.2</td>
</tr>
<tr>
<td>PTL_RANK_ANY</td>
<td>wildcard for rank fields</td>
<td>ptl_rank_t</td>
<td>3.2.6, 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>3.2.7, 3.5.4</td>
</tr>
<tr>
<td>PTL_SR_PERMISSIONS_VIOLATIONS</td>
<td>index for the permission violations register</td>
<td>ptl_sr_index_t</td>
<td>3.2.7, 3.5.4</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>3.2.6, 3.12.2, 3.11, 3.12</td>
</tr>
</tbody>
</table>
Chapter 4

The Semantics of Message Transmission

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.

4.1 Sending Messages

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

<table>
<thead>
<tr>
<th>IMPLEMENTATION NOTE 34:</th>
<th>Information on the wire</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This section describes the information that portals semantics require to be passed between an initiator and its target. The portals specification does not enforce a given wire protocol or in what order and what manner information is passed along the communication path. For example, portals semantics require that an acknowledgment event contains the 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 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.</td>
</tr>
</tbody>
</table>

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 acknowledgement was requested, md_handle and user_ptr do not need to be sent. If an acknowledgement is requested (either PTL_CT_ACK_REQ, PTL_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 acknowledgement event. The user_ptr is only required in the case of a full acknowledgement (PTL_ACK_REQ), PTL_CT_ACK_REQ and PTL_OC_ACK_REQ requests do not require the user_ptr field to generate the acknowledgement 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 82.)

**IMPLEMENTATION NOTE 35:**

Table 4.1 specifies sizes for each data item that are conformant to the Portals 4.0 specification; however, a given implementation can impose additional constraints to reduce the size of some of these fields. For example, the `remote_offset` could each be reduced to 5 bytes on a platform that supported less than 1 TB of memory. Further reductions for the special case of the non-matching operation with only a `PTL_CT_ACK_REQ` or `PTL_OC_ACK_REQ` would reduce the Portals Send Request significantly. Similar optimizations are available in other pieces of wire information.

<table>
<thead>
<tr>
<th>Information</th>
<th>Type</th>
<th><code>PtlPut()</code> Argument</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td>operation</td>
<td>4b</td>
<td>indicates a put request</td>
</tr>
<tr>
<td>ack type</td>
<td><code>ptl_ack_req_t</code></td>
<td><code>ack_req</code></td>
<td>2b</td>
<td></td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td><code>md_handle</code></td>
<td>2b</td>
<td><code>options</code> field from NI associated with MD</td>
</tr>
<tr>
<td>job identifier</td>
<td><code>ptl_jid_t</code></td>
<td></td>
<td>4B</td>
<td>local information (if supported)</td>
</tr>
<tr>
<td>initiator</td>
<td><code>ptl_process_id_t</code></td>
<td></td>
<td>4B</td>
<td>local information</td>
</tr>
<tr>
<td>user</td>
<td><code>ptl_uid_t</code></td>
<td></td>
<td>4B</td>
<td>local information</td>
</tr>
<tr>
<td>target</td>
<td><code>ptl_process_id_t</code></td>
<td><code>target_id</code></td>
<td>4B</td>
<td></td>
</tr>
<tr>
<td>portal index</td>
<td><code>ptl_pt_index_t</code></td>
<td><code>pt_index</code></td>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>match bits</td>
<td><code>ptl_match_bits_t</code></td>
<td><code>match_bits</code></td>
<td>8B</td>
<td>opt. if <code>options.PTL_NI_NO_MATCHING</code></td>
</tr>
<tr>
<td>offset</td>
<td><code>ptl_size_t</code></td>
<td><code>remote_offset</code></td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>memory desc</td>
<td><code>ptl_memory_md_t</code></td>
<td><code>md_handle</code></td>
<td>2B</td>
<td>opt. if <code>ack_req = PTL_NO_ACK_REQ</code></td>
</tr>
<tr>
<td>header data</td>
<td><code>ptl_hdr_data_t</code></td>
<td><code>hdr.data</code></td>
<td>8B</td>
<td>user data in header</td>
</tr>
<tr>
<td>put user pointer</td>
<td>void *</td>
<td><code>user_ptr</code></td>
<td>8B</td>
<td>opt. if <code>ack_req = PTL_NO_ACK_REQ</code> or <code>ack_req = PTL_CT_ACK_REQ</code> or <code>ack_req = PTL_OC_ACK_REQ</code></td>
</tr>
<tr>
<td>length</td>
<td><code>ptl_size_t</code></td>
<td><code>length</code></td>
<td>8B</td>
<td><code>length</code> argument</td>
</tr>
<tr>
<td>data</td>
<td>bytes</td>
<td><code>md_handle</code></td>
<td></td>
<td>user data</td>
</tr>
<tr>
<td>total</td>
<td>unsigned int</td>
<td></td>
<td>61B</td>
<td></td>
</tr>
</tbody>
</table>

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 the target memory descriptor has the PTL_ME_MANAGE_LOCAL flag set, the offset local to the target memory descriptor is used. If the flag is set, the offset requested by the initiator is used. An acknowledgment message returns the actual value used.

Lightweight “counting” acknowledgments do not require the actual offset used or user pointer since they do not generate a pti_process_id_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>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td></td>
<td>4B</td>
<td>indicates an acknowledgment</td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td>put_md_handle</td>
<td>2B</td>
<td>options field from NI associated with MD</td>
</tr>
<tr>
<td>initiator</td>
<td>pti_process_id_t</td>
<td>target_id</td>
<td>4B</td>
<td>echo target of put</td>
</tr>
<tr>
<td>target</td>
<td>pti_process_id_t</td>
<td>initiator</td>
<td>4B</td>
<td>echo initiator of put</td>
</tr>
<tr>
<td>memory descriptor</td>
<td>pti_md_handle</td>
<td>md_handle</td>
<td>2B</td>
<td>echo md_handle of put</td>
</tr>
<tr>
<td>put user pointer</td>
<td>void *</td>
<td>user_ptr</td>
<td>8B</td>
<td>echo user_ptr of put</td>
</tr>
<tr>
<td>offset</td>
<td>pti_size_t</td>
<td>remote_offset</td>
<td>8B</td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>manipulated length</td>
<td>pti_size_t</td>
<td></td>
<td>8B</td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>Total</td>
<td>unsigned int</td>
<td></td>
<td>35B</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>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td></td>
<td>4B</td>
<td>indicates an acknowledgment</td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td>put_md_handle</td>
<td>2B</td>
<td>options field from NI associated with MD</td>
</tr>
<tr>
<td>initiator</td>
<td>pti_process_id_t</td>
<td>target_id</td>
<td>4B</td>
<td>local information on put target</td>
</tr>
<tr>
<td>target</td>
<td>pti_process_id_t</td>
<td>initiator</td>
<td>4B</td>
<td>echo initiator of put</td>
</tr>
<tr>
<td>memory descriptor</td>
<td>pti_md_handle</td>
<td>md_handle</td>
<td>2B</td>
<td>echo md_handle of put</td>
</tr>
<tr>
<td>manipulated length</td>
<td>pti_size_t</td>
<td></td>
<td>8B</td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>Total</td>
<td>unsigned int</td>
<td></td>
<td>19B</td>
<td></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 4.5 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 4.4. 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>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td></td>
<td>4b</td>
<td>indicates a get operation</td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td>md_handle</td>
<td>2b</td>
<td>options field from NI associated with MD</td>
</tr>
<tr>
<td>job identifier</td>
<td>ptl_jid_t</td>
<td></td>
<td>4B</td>
<td>local information (if supported)</td>
</tr>
<tr>
<td>initiator</td>
<td>ptl_process_id_t</td>
<td></td>
<td>4B</td>
<td>local information</td>
</tr>
<tr>
<td>user</td>
<td>ptl_uid_t</td>
<td></td>
<td>4B</td>
<td>local information</td>
</tr>
<tr>
<td>target</td>
<td>ptl_process_id_t</td>
<td>target_id</td>
<td>4B</td>
<td></td>
</tr>
<tr>
<td>portal index</td>
<td>ptl_pt_index_t</td>
<td>pt_index</td>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>match bits</td>
<td>ptl_match_bits_t</td>
<td>match_bits</td>
<td>8B</td>
<td>optional if the PTL_NI_NO_MATCHING option is set.</td>
</tr>
<tr>
<td>offset</td>
<td>ptl_size_t</td>
<td>remote_offset</td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>memory descriptor</td>
<td>ptl_handle_md_t</td>
<td>md_handle</td>
<td>2B</td>
<td>destination of reply</td>
</tr>
<tr>
<td>length</td>
<td>ptl_size_t</td>
<td>length</td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>initiator offset</td>
<td>ptl_size_t</td>
<td>local_offset</td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>get user pointer</td>
<td>void *</td>
<td>user_ptr</td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>unsigned int</td>
<td></td>
<td>61B</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5. Reply: Information Passed in a Reply.

<table>
<thead>
<tr>
<th>Information</th>
<th>Type</th>
<th>PtlGet() Argument</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td></td>
<td>4b</td>
<td>indicates a reply</td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td>get_md_handle</td>
<td>2b</td>
<td>options field from NI associated with MD</td>
</tr>
<tr>
<td>initiator</td>
<td>ptl_process_id_t</td>
<td>target_id</td>
<td>4B</td>
<td>local information on get target</td>
</tr>
<tr>
<td>target</td>
<td>ptl_process_id_t</td>
<td>initiator</td>
<td>4B</td>
<td>echo initiator of get</td>
</tr>
<tr>
<td>memory descriptor</td>
<td>ptl_handle_md_t</td>
<td>md_handle</td>
<td>2B</td>
<td>echo md_handle of get</td>
</tr>
<tr>
<td>initiator offset</td>
<td>ptl_size_t</td>
<td>local_offset</td>
<td>8B</td>
<td>echo local_offset of get</td>
</tr>
<tr>
<td>get user pointer</td>
<td>void *</td>
<td>user_ptr</td>
<td>8B</td>
<td>echo user_ptr of get</td>
</tr>
<tr>
<td>manipulated length</td>
<td>ptl_size_t</td>
<td></td>
<td>8B</td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>offset</td>
<td>ptl_size_t</td>
<td>remote_offset</td>
<td>8B</td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>data</td>
<td>bytes</td>
<td></td>
<td>8B</td>
<td>obtained from the operation</td>
</tr>
<tr>
<td>Total</td>
<td>unsigned int</td>
<td></td>
<td>43B</td>
<td></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.

4.2 Receiving Messages

When an incoming message arrives on a network interface, the communication system first checks that the target process identified in the request is a valid process that has initialized the network interface (i.e., that the target process has a valid portal table). If this test fails, the communication system discards the message and increments the dropped message count for the interface. The remainder of the processing depends on the type of the incoming message. put, get, and 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.
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>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>int</td>
<td></td>
<td>2B</td>
<td></td>
</tr>
<tr>
<td>options</td>
<td>unsigned int</td>
<td>put_md_handle</td>
<td>2b</td>
<td>options field from NI associated with MD</td>
</tr>
<tr>
<td>ack type</td>
<td>put_ack_req_t</td>
<td>ack_dest</td>
<td>2b</td>
<td></td>
</tr>
<tr>
<td>job identifier</td>
<td>put_jid_t</td>
<td></td>
<td>4B</td>
<td>local information (if supported)</td>
</tr>
<tr>
<td>initiator</td>
<td>put_process_id_t</td>
<td>target_id</td>
<td>4B</td>
<td>local information</td>
</tr>
<tr>
<td>user</td>
<td>put_uid_t</td>
<td></td>
<td>4B</td>
<td>local information</td>
</tr>
<tr>
<td>target</td>
<td>put_process_id_t</td>
<td>target_id</td>
<td>4B</td>
<td></td>
</tr>
<tr>
<td>portal index</td>
<td>put_pt_index_t</td>
<td>pt_index</td>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>memory descriptor</td>
<td>put_md_handle</td>
<td>put_md_handle</td>
<td>2B</td>
<td>opt. if ack_dest = PTL_NO_ACK_REQ</td>
</tr>
<tr>
<td>user pointer</td>
<td>void *</td>
<td>user_ptr</td>
<td>8B</td>
<td>opt. if ack_dest = PTL_NO_ACK_REQ or ack_dest = PTL_REF_ACK_REQ or ack_dest = PTL_DC_ACK_REQ</td>
</tr>
<tr>
<td>match bits</td>
<td>pctl_match_bits_t</td>
<td>match_bits</td>
<td>8B</td>
<td>optional if the PTL_NI_NO_MATCHING option is set.</td>
</tr>
<tr>
<td>offset</td>
<td>pctl_size_t</td>
<td>remote_offset</td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>memory descriptor</td>
<td>put_md_handle</td>
<td>get_md_handle</td>
<td>2B</td>
<td>destination of reply</td>
</tr>
<tr>
<td>length</td>
<td>pctl_size_t</td>
<td>put_md_handle</td>
<td>8B</td>
<td>length member</td>
</tr>
<tr>
<td>operand</td>
<td>bytes</td>
<td>operand</td>
<td>8B</td>
<td>Used in CSWAP and MSWAP operations</td>
</tr>
<tr>
<td>data</td>
<td>bytes</td>
<td>put_md_handle</td>
<td></td>
<td>user data</td>
</tr>
</tbody>
</table>

| Total       | unsigned int | 65B           |      |       |

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 2.2 and Figure 2.9.

<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


**NOTE:** This is the final version of the document for Portals version 3.3. It supersedes SAND99-2959


Appendix A

Frequently Asked Questions

This document is a specification for the portals 4.0 API. People using and implementing Portals sometimes have questions that the specification does not address. In this appendix we answer some of the more common questions.

Q Are Portals a wire protocol?

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

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

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

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

MPI is a higher level API than Portals. In many ways, it provides simpler semantics and APIs. It also provides a variety of higher level APIs (derived data types, collective operations) that Portals does not.

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

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

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

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

Most of the research message passing layers do not provide reliability because they were designed for networks that are, for all practical purposes, reliable. While Portals themselves do not provide a wire protocol, Portals demand that the transport layer underneath is reliable. This places Portals a level above the other APIs in the networking stack. On
reliable networks, such as ASCI Red, Portals can be implemented without a wire protocol. On unreliable networks, such as Myrinet, Portals can run over GM or some other protocol that implements reliability.

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

**Q** What is a NAL?

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

**Q** Must Portals be implemented using a NAL?

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

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

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

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

Portals Design Guidelines

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

B.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.
B.2 The Will Requirements

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

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

**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.

B.3 The Should Requirements

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

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

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

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

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

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

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

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

A README Template

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

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

**Status Registers.** Portals define a set of status registers (Section 3.2.7). The type `pt1_sr_index_t` defines the mandatory `PTL_SR_DROP_COUNT` and `PTL_SR_PERMISSIONS_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 3.2.5 and 3.5.2). An implementation that supports multiple interfaces must specify the constants used to access the various interfaces through `PtlNIInit()`.

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

**Job identifiers.** The README file should indicate whether the implementation supports job identifiers (Section 3.9).

**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).
In this appendix we briefly mention two portals 3.3 implementations: A reference implementation and one that runs on Cray’s XT3/XT4/XT5 Red Storm.

D.1 Reference Implementation

A portals 3.3 reference implementation has been written and is maintained by Jim Schutt. The main goal of the reference implementation is to provide a working example that implements the syntax, semantics, and spirit of Portals as described in the version 3.3 document. While many of the semantics remain the same or similar, many semantics have been added or revised.

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

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

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


In addition to comments in the code, it contains several README files that describe the implementation. Feedback is highly encouraged to the code author, jaschut@sandia.gov, and the Portals 4.0 team at Sandia National Laboratories, p3@sandia.gov.

A NAL that runs in Linux kernel space is currently under development.
We maintain a portals web site at http://www.cs.sandia.gov/Portals with links to the latest reference implementation and other information.

## D.2 Portals 3.3 on the Cray XT3/XT4/XT5 Red Storm

There are two implementations of Portals available on Cray’s XT3/XT4/XT5 Red Storm system. One, generic, is provided by Cray with the machine. The second, accelerated, is under active development at Sandia National Laboratories. There are plans to merge the two versions in the future.

### D.2.1 Generic

This is the version provided by Cray with its XT3/XT4/XT5 Red Storm systems. A large portion of the portals code is implemented inside the kernel. When messages arrive at the Seastar NIC, it causes an interrupt and lets the kernel process the portals header; i.e., resolve portal table addressing and match list traversal. The accelerated version under development places more of the portals code inside the Seastar NIC and avoids the interrupt processing on each message arrival.

The generic implementation does not completely match the definitions in the version 3.3 document. The main differences are listed here:

- **PtlHandleIsEqual()** is not implemented.

- **Limitations on IOVECs**: Only the first and last entry can be unaligned (at the head of the buffer and at the tail of the buffer, everything else must be quad-byte aligned).

- There are three new functions that are not part of this document: `PtlIsValidHandle()`, `PtlSetInvalidHandle()`, and `PtlTestAtomic()`.

- The following return codes are not implemented: `PTL_MD_ILLEGAL`, and `PTL_IFACE_INVALID`.

- The type `ptl_size_t` is 32 bits wide, not 64 bits.

- `PtlEQGet()` and `PtlEQWait()` may return a `ptl_event_t` structure that is not fully filled in.

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

### D.2.2 Accelerated

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

- Adds a `PtlPost()` call which combines a and `PtlMDUpdate()` call. This eliminates a protection domain boundary crossing in many of the common usage cases.

- The `PtlGet()` operation generates a `PTL_EVENT_SEND` event.

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

Summary of Changes

The most recent version of this document described Portals version 3.3 [Riesen et al. 2006]. 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 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 second fundamental 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.

In support of the lightweight communication semantics required by PGAS models, lightweight “counting” events and acknowledgements were added. In addition, a non-matching interface was created to decrease the processing required for PGAS messages. Finally, a PtlAtomic() function was added to support functionalities commonly provided in PGAS models.

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 or job ID based authentication were moved to permissions fields on the respective list entry or match list entry.
### Index

<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_id (field)</td>
<td></td>
<td>56, 60</td>
</tr>
<tr>
<td>accelerated</td>
<td></td>
<td>122</td>
</tr>
<tr>
<td>ack_req (field)</td>
<td></td>
<td>82, 86, 90, 92, 106, 109</td>
</tr>
<tr>
<td>acknowledgment</td>
<td></td>
<td>see operations</td>
</tr>
<tr>
<td>acknowledgment type</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>actual (field)</td>
<td></td>
<td>41, 42</td>
</tr>
<tr>
<td>actual_mapping (field)</td>
<td></td>
<td>41, 42</td>
</tr>
<tr>
<td>address space opening</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>address translation</td>
<td></td>
<td>23, 25, 27, 28, 29, 31, 108</td>
</tr>
<tr>
<td>addressing, portals</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>alignment</td>
<td></td>
<td>51, 55, 60, 119</td>
</tr>
<tr>
<td>API</td>
<td></td>
<td>13, [14]</td>
</tr>
<tr>
<td>API summary</td>
<td></td>
<td>96</td>
</tr>
<tr>
<td>application bypass</td>
<td></td>
<td>18, 20, 21, 23</td>
</tr>
<tr>
<td>application space</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>argument names</td>
<td></td>
<td>see structure fields</td>
</tr>
<tr>
<td>ASC</td>
<td></td>
<td>[14]</td>
</tr>
<tr>
<td>ASCI</td>
<td></td>
<td>[14]</td>
</tr>
<tr>
<td>atomic</td>
<td></td>
<td>see operations</td>
</tr>
<tr>
<td>datatypes</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>operations</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>atomic operation</td>
<td></td>
<td>23, 25, 83, 99</td>
</tr>
<tr>
<td>atomic swap</td>
<td></td>
<td>see swap</td>
</tr>
<tr>
<td>atomic_operation (field)</td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>atomic_type (field)</td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>authors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compaq, Microsoft, and Intel</td>
<td></td>
<td>18, 21, (111)</td>
</tr>
<tr>
<td>Infiniband Trade Association</td>
<td></td>
<td>18, (111)</td>
</tr>
<tr>
<td>Message Passing Interface Forum</td>
<td></td>
<td>18, (111)</td>
</tr>
<tr>
<td>Myricom, Inc.</td>
<td></td>
<td>21, (111)</td>
</tr>
<tr>
<td>Task Group of Technical Committee T11</td>
<td></td>
<td>21, 27, (111)</td>
</tr>
<tr>
<td>Alverson</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Alverson, Robert</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Brightwell and Shuler</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>Brightwell et al.</td>
<td></td>
<td>13, 19</td>
</tr>
<tr>
<td>Brightwell, Ron</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Chien, Andrew</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Cray Research, Inc.</td>
<td></td>
<td>18, 27</td>
</tr>
<tr>
<td>Fisk, Lee Ann</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Greenberg, David S.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Hori, A.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Hudson, Tramm</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Hudson, Trammell</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Ishikawa et al.</td>
<td></td>
<td>14, 21</td>
</tr>
<tr>
<td>Ishikawa, Y.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Jong, Chu</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Lauria et al.</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Lauria, Mario</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Maccabe et al.</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Maccabe, Arthur B.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>McCurley, Kevin S.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Message Passing Interface Forum</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>Pakin, Scott</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Pedretti, Kevin</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Pedretti, Kevin T.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Riesen et al.</td>
<td></td>
<td>19, 123</td>
</tr>
<tr>
<td>Riesen, Rolf</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Shuler et al.</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Shuler, Lance</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Stallcup, T. Mack</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Tezuka, H.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>van Dresser, David</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>Wheat, Stephen R.</td>
<td></td>
<td>(111)</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>background</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Barrett, Brian</td>
<td></td>
<td>1, 3</td>
</tr>
<tr>
<td>barrier operation</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>Barsis, Ed</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Barton, Eric</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Braam, Peter</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Brightwell, Ron</td>
<td></td>
<td>1, 3</td>
</tr>
<tr>
<td>buffer alignment</td>
<td></td>
<td>51, 55, 60, 119</td>
</tr>
<tr>
<td>bypass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>application</td>
<td></td>
<td>18, 20, 21, 23</td>
</tr>
<tr>
<td>OS</td>
<td></td>
<td>18, 20, 21, 116</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAF</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Camp, Bill</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>changes, API and document</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>collective operations</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>communication model</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>connection-oriented</td>
<td></td>
<td>18, 113</td>
</tr>
<tr>
<td>connectionless</td>
<td></td>
<td>18, 19</td>
</tr>
<tr>
<td>constants</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTL_ACK_REQ</td>
<td></td>
<td>35, 80, 82, 86, 100, 105</td>
</tr>
<tr>
<td>PTL_BAND</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>PTL_BOR</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>PTL_BXOR</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>PTL_CHAR</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>PTL_CSWAP</td>
<td></td>
<td>83, 84, 88</td>
</tr>
<tr>
<td>PTL_CT_ACK_REQ</td>
<td></td>
<td>80, 82, 86, 100, 105, 106, 109</td>
</tr>
<tr>
<td>PTL_CT_BYTE</td>
<td></td>
<td>81, 100</td>
</tr>
<tr>
<td>PTL_CT_NONE</td>
<td></td>
<td>36, 51, 55, 60, 100</td>
</tr>
<tr>
<td>PTL_CT_OPERATION</td>
<td></td>
<td>81, 103</td>
</tr>
</tbody>
</table>
PTL_DOUBLE ............................. 85
PTL_EQ_NONE .......................... 36, 45, 51, 67, 100
PTL_EVENT_ACK ....................... 51, 66–68, 81, 82, 100
PTL_EVENT_ATOMIC .................... 57, 62, 66, 68, 84, 101
PTL_EVENT_ATOMIC_OVERFLOW ........... 57, 62, 68
PTL_EVENT_DROPPED ................... 29, 31, 54, 59, 66, 68, 101
PTL_EVENT_FREE ....................... 54, 56, 59, 62, 66, 68, 101
PTL_EVENT_GET ....................... 56, 62, 65, 68, 82, 101
PTL_EVENT_PROBE ..................... 62, 66, 68, 101
PTL_EVENT_PT_DISABLED ................. 28, 68, 72, 101
PTL_EVENT_PUT ....................... 57, 62, 65, 68, 81, 101
PTL_EVENT_PUT_OVERFLOW ............... 29, 57, 62, 66, 81, 101
PTL_EVENT_REPLY ..................... 51, 66, 68, 82, 101
PTL_EVENT_SEND ........................ 32, 51, 56, 60, 66–68, 81, 82, 101, 102
PTL_EVENT_UNLINK ..................... 54, 56, 59, 62, 66–68, 101
PTL_FLOAT .............................. 85
PTL_IFACE_DEFAULT .................... 37, 101, 119
PTL_INT ................................ 85
PTL_INVALID_HANDLE ................... 36, 96, 101
PTL_IOVEC ............................. 51, 52, 56, 61, 102
PTL_JID_ANY .................... 37, 55, 56, 60, 101
PTL_JID_NONE ......................... 49, 50, 70, 101
PTL_RANK .............................. 85
PTL_LAND .............................. 84
PTL_LE_ACK_DISABLE ................... 56, 101
PTL_LE_AUTH_USE_JID .................. 57, 101
PTL_LE_EVENT_CT_ATOMIC ............. 57, 101
PTL_LE_EVENT_CT_ATOMIC_OVERFLOW ...... 57, 101
PTL_LE_EVENT_CT_GET .................. 56, 101
PTL_LE_EVENT_CT_PUT .................. 57, 101
PTL_LE_EVENT_CT_PUT_OVERFLOW ........ 57, 101
PTL_LE_EVENT_DISABLE ................ 56, 101
PTL_LE_EVENT_DISABLE ................. 56, 101
PTL_LE_EVENT_OVER_DISABLE .......... 56, 101
PTL_LE_EVENT_SUCCESS_DISABLE ....... 56, 101
PTL_LE_EVENT_UNLINK_DISABLE ......... 56, 101
PTL_LE_MAY_ALIGN .................... 102
PTL_LE_OP_GET ........................ 56, 101, 110, 123
PTL_LE_OP_PUT ........................ 56, 101, 110, 123
PTL_LE_USE_ONCE ..................... 45, 56, 102, 103
PTL_LONG ................................ 85
PTL_LOR ................................ 84
PTL_LXOR ................................ 84
PTL_MAX ................................ 84
PTL_MD_EVENT_CT_ACK .................. 51, 102
PTL_MD_EVENT_CT_REPLY ................ 51, 102
PTL_MD_EVENT_CT_SEND ................ 51, 102
PTL_MD_EVENT_DISABLE ................ 51, 101
PTL_MD_EVENT_SUCCESS_DISABLE ...... 51, 101
PTL_MD_REMOTE_FAILURE_DISABLE ..... 51, 56, 60, 69, 102
PTL_MD_UNORDERED .................... 51, 102
PTL_ME_ACK_DISABLE .................. 61, 102
PTL_ME_AUTH_USE_JID .................. 62, 102
PTL_ME_EVENT_CT_ATOMIC ............ 62, 102
PTL_ME_EVENT_CT_ATOMIC_OVERFLOW ...... 62, 102
PTL_ME_EVENT_CT_GET .................. 62, 102
PTL_ME_EVENT_CT_PUT .................. 62, 102
PTL_ME_EVENT_CT_PUT_OVERFLOW ...... 62, 102
PTL_ME_EVENT_DISABLE ................. 62, 67, 102
PTL_ME_EVENT_OVER_DISABLE .......... 62
PTL_ME_EVENT_SUCCESS_DISABLE ...... 62, 102
PTL_ME_EVENT_UNLINK_DISABLE ....... 62, 67, 102
PTL_ME_MANAGE_LOCAL 60, 61, 82, 83, 86–88, 102, 107
PTL_ME_MAY_ALIGN .................... 61, 103
PTL_ME_MIN_FREE ..................... 60, 61, 102
PTL_ME_NO_TRUNCATE .................. 61, 102, 110
PTL_ME_OP_GET ........................ 61, 84, 102, 110
PTL_ME_OP_PUT ........................ 61, 84, 102, 110
PTL_ME_USE_ONCE ..................... 45, 61, 103
PTL_MIN ................................ 84
PTL_MSWAP ............................ 83, 84, 88
PTL_NI_FLOW_CTRL ..................... 69, 103
PTL_NI_LOGICAL ....................... 37, 41, 59, 103
PTL_NI_MATCHING ...................... 41, 103
PTL_NI_NO_MATCHING .................. 41, 54, 59, 103, 106, 108, 109
PTL_NI_OK ............................. 62, 66, 69, 71, 103
PTL_NI_PERM_VIOLATION ............... 69, 103
PTL_NI_PHYSICAL ...................... 37, 41, 103
PTL_NI_UNDELIVERABLE ................ 69, 103
PTL_NID_ANY ......................... 37, 62, 103
PTL_NO_ACK_REQ ...................... 80, 82, 86, 103, 106, 109
PTL_OC_ACK_REQ 80, 82, 86, 100, 105, 106, 109
PTL_OVERFLOW ......................... 57, 62, 63, 103
PTL_PID_ANY ......................... 37, 41, 62, 103
PTL_PRIORITY_LIST .................... 57, 62, 63, 101
PTL_PROBEONLY ....................... 57, 62, 63, 104
PTL_PROD ................................ 84
PTL_PT_ANY ............................ 45, 103
PTL_PT_FLOW_CTRL ..................... 45
PTL_PT_ONLY_USE_ONCE ................. 45, 103
PTL_RANK_ANY ......................... 37, 62, 104
PTL_SHORT ............................ 85
PTL_SR_DROP_COUNT ................... 37, 43, 104, 119
PTL_SR_PERMISSIONS_VIOLATIONS ..... 37, 43, 56, 60, 104, 119
PTL_SUM ................................ 84
PTL_SWAP ................................ 84
PTL_TIME_FOREVER ..................... 75, 104
PTL_UCHAR ............................ 85
PTL_UID_ANY ......................... 37, 55, 56, 60, 104
PTL_UINT .............................. 85
PTL_ULONG ............................ 85
PTL_USHORT ........................... 85
<table>
<thead>
<tr>
<th>Field</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_ID_checking</td>
<td>64</td>
</tr>
<tr>
<td>match_list</td>
<td>59</td>
</tr>
<tr>
<td>match_list_entries</td>
<td>20</td>
</tr>
<tr>
<td>match_list_entry</td>
<td>see ME, 54, 59, 62</td>
</tr>
<tr>
<td>match_id(field)</td>
<td>59, 62, 64</td>
</tr>
<tr>
<td>matching_address_translation</td>
<td>30</td>
</tr>
<tr>
<td>max_atomic_size(field)</td>
<td>40, 83</td>
</tr>
<tr>
<td>max_csts(field)</td>
<td>40</td>
</tr>
<tr>
<td>max_eqs(field)</td>
<td>40</td>
</tr>
<tr>
<td>max_idvec(field)</td>
<td>40</td>
</tr>
<tr>
<td>max_mdvec(field)</td>
<td>40</td>
</tr>
<tr>
<td>max_mes(field)</td>
<td>40</td>
</tr>
<tr>
<td>max_msg_size(field)</td>
<td>40</td>
</tr>
<tr>
<td>max_pt_index(field)</td>
<td>40</td>
</tr>
<tr>
<td>McCurley, Kevin</td>
<td>4</td>
</tr>
<tr>
<td>MD</td>
<td>50</td>
</tr>
<tr>
<td>alignment</td>
<td>51, 119</td>
</tr>
<tr>
<td>bind</td>
<td>51</td>
</tr>
<tr>
<td>options</td>
<td>51</td>
</tr>
<tr>
<td>pending_operation</td>
<td>53, 107</td>
</tr>
<tr>
<td>release</td>
<td>50, 53, 97, 99, 100</td>
</tr>
<tr>
<td>unlink</td>
<td>107</td>
</tr>
<tr>
<td>md(field)</td>
<td>53</td>
</tr>
<tr>
<td>md_handle(field)</td>
<td>53, 54, 81–83, 85, 86, 90–92, 105–108</td>
</tr>
<tr>
<td>ME</td>
<td>59</td>
</tr>
<tr>
<td>access_control</td>
<td>27–29, 47, 55, 62</td>
</tr>
<tr>
<td>alignment</td>
<td>60, 119</td>
</tr>
<tr>
<td>append</td>
<td>62</td>
</tr>
<tr>
<td>free</td>
<td>66, 68, 101</td>
</tr>
<tr>
<td>ignore_bits</td>
<td>see ignore bits list types</td>
</tr>
<tr>
<td>list_types</td>
<td>63</td>
</tr>
<tr>
<td>match_bits</td>
<td>see match bits</td>
</tr>
<tr>
<td>message_reject</td>
<td>110</td>
</tr>
<tr>
<td>options</td>
<td>60</td>
</tr>
<tr>
<td>pending_operation</td>
<td>64</td>
</tr>
<tr>
<td>permissions</td>
<td>27–29, 62</td>
</tr>
<tr>
<td>probe</td>
<td>62, 63, 66, 68, 101</td>
</tr>
<tr>
<td>protection</td>
<td>27–29</td>
</tr>
<tr>
<td>truncate</td>
<td>61, 70, 102, 109, 110</td>
</tr>
<tr>
<td>unlink</td>
<td>29, 59, 60, 64–68, 97, 99–101</td>
</tr>
<tr>
<td>me(field)</td>
<td>63</td>
</tr>
<tr>
<td>me_handle(field)</td>
<td>63, 64</td>
</tr>
<tr>
<td>memory_descriptor</td>
<td>see also MD, [14], 20, 31, 50</td>
</tr>
<tr>
<td>message</td>
<td>[14]</td>
</tr>
<tr>
<td>message_operation</td>
<td>[14]</td>
</tr>
<tr>
<td>message_rejection</td>
<td>109</td>
</tr>
<tr>
<td>messages, receiving</td>
<td>108</td>
</tr>
<tr>
<td>messages, sending</td>
<td>105</td>
</tr>
<tr>
<td>min_free(field)</td>
<td>29, 60, 61, 71, 102</td>
</tr>
<tr>
<td>mlength(field)</td>
<td>29, 66, 70, 71, 81</td>
</tr>
<tr>
<td>MPI</td>
<td>[14], 18, 19, 27, 58, 63, 82, 113, 116</td>
</tr>
<tr>
<td>progress_rule</td>
<td>18, 21</td>
</tr>
<tr>
<td>MPI_scalability</td>
<td>18</td>
</tr>
<tr>
<td>MPP</td>
<td>[14]</td>
</tr>
<tr>
<td>Myrinet</td>
<td>114</td>
</tr>
<tr>
<td>N</td>
<td>14, 114, 121</td>
</tr>
<tr>
<td>NAL</td>
<td>35</td>
</tr>
<tr>
<td>network</td>
<td>[14]</td>
</tr>
<tr>
<td>network_independence</td>
<td>18</td>
</tr>
<tr>
<td>network_interface</td>
<td>see also NI, 20, 35–37, 38, 40, 54, 59, 108</td>
</tr>
<tr>
<td>network_interface_initialization</td>
<td>40</td>
</tr>
<tr>
<td>network_interfaces</td>
<td>multiple</td>
</tr>
<tr>
<td>network_scalability</td>
<td>18</td>
</tr>
<tr>
<td>new_ct(field)</td>
<td>79</td>
</tr>
<tr>
<td>NI</td>
<td>41</td>
</tr>
<tr>
<td>options</td>
<td>43</td>
</tr>
<tr>
<td>NI_fini</td>
<td>44</td>
</tr>
<tr>
<td>NI_handle</td>
<td>40</td>
</tr>
<tr>
<td>NI_init</td>
<td>43</td>
</tr>
<tr>
<td>NI_status</td>
<td>43</td>
</tr>
<tr>
<td>ni_fails_type(field)</td>
<td>32, 56, 60, 66, 69, 71</td>
</tr>
<tr>
<td>ni_handle(field)</td>
<td>41–50, 52, 53, 57, 58, 62, 63, 72, 77</td>
</tr>
<tr>
<td>nid(field)</td>
<td>48</td>
</tr>
<tr>
<td>node</td>
<td>[14]</td>
</tr>
<tr>
<td>node_ID</td>
<td>27, 29, 37, 48</td>
</tr>
<tr>
<td>non-matching_address_translation</td>
<td>31</td>
</tr>
<tr>
<td>NULL_LE</td>
<td>55</td>
</tr>
<tr>
<td>NULL_ME</td>
<td>60</td>
</tr>
<tr>
<td>O</td>
<td>14, 20, 27, 70, 106–109</td>
</tr>
<tr>
<td>offset</td>
<td>60, 61, 70, 71, 81–83, 102</td>
</tr>
<tr>
<td>remote</td>
<td>56, 60, 70, 71, 82, 83, 86–88</td>
</tr>
<tr>
<td>offset(field)</td>
<td>71</td>
</tr>
<tr>
<td>one-sided_operation</td>
<td>20, 27</td>
</tr>
<tr>
<td>opening_into_address_space</td>
<td>23</td>
</tr>
<tr>
<td>operand</td>
<td>109</td>
</tr>
<tr>
<td>operand(field)</td>
<td>83, 88, 94</td>
</tr>
<tr>
<td>operation(field)</td>
<td>86–88, 92–94</td>
</tr>
<tr>
<td>operation_completed</td>
<td>81</td>
</tr>
<tr>
<td>operations</td>
<td>26, 43, 56, 61, 65–67, 105–109, 121</td>
</tr>
<tr>
<td>acknowledgment</td>
<td>14, 23, 25, 28, 43, 56, 61, 66–68, 85, 105, 108–110</td>
</tr>
<tr>
<td>atomic</td>
<td>83</td>
</tr>
<tr>
<td>fetch_and_atomic</td>
<td>86</td>
</tr>
</tbody>
</table>
one-sided ........................................ 20, 27
reply ..21, 25, 32, 40, 43, 56, 58, 61, 64, 66, 68, 69, 83, 105, 107–109
swap .................................................. 88
two-sided ............................................ 20, 27
options (field) .................. 41, 45, 51, 56, 60, 106–109
ordering semantics ............... 19, 20, 30, 51
OS bypass ......................... 18, 20, 21, 116
Otto, Jim ........................................ 4
overflow list 24, 28, 29, 31, 33, 34, 59, 66, 70, 80, 81, 123

P
parallel job ........................................ 19, 49
Pedretti, Kevin ................. 1, 3
pending operation .......... see MD
people
Barrett, Brian ................. 1, 3
Barsis, Ed .............................. 4
Barton, Eric ......................... 4
Braam, Peter ...................... 4
Brightwell, Ron ............... 1, 3
Camp, Bill ......................... 4
Fisk, Lee Ann .................... 4
Greenberg, David ............ 4
Hale, Art ......................... 4
Hoffman, Eric .................... 4
Hudson, Trammell ............ 1, 3
Istrail, Gabi ..................... 4
Johnston, Jeanette ............. 4
Jong, Chu ......................... 4
Kaul, Clint ..................... 4
Levenhagen, Mike ............ 4
Maccabe, Arthur B. ............ 1, 3
McCurley, Kevin ............ 4
Otto, Jim ......................... 4
Pedretti, Kevin .............. 1, 3
Pundit, Neil .................... 4
Riesen, Rolf .................... 1, 3
Robboy, David .................. 4
Schutt, Jim .................... 4, 121
Sears, Mark .................... 4
Shuler, Lance .................. 4
Stallcup, Mack .................. 4
Underwood, Keith ............ 1, 3
Underwood, Todd ........... 4
Vigil, Dena .................... 4
Ward, Lee .................... 4
Wheat, Stephen .............. 4
van Dresser, David ............ 4

performance ........................................ 115
permission violations count ........... 43, 104
PGAS ........................................ 18, 116
pid (field) ............................. 41, 42, 48
portability .................................. 39, 113
portal
indexes ........................................ 36
table ........................................ 27, 39, 119
table index .............. 45–47, 54, 59, 106, 108–110
portal table entry ........ 35, 45
allocation ......................... 45
disable ......................... 46
enable .......................... 47
freeing ......................... 46
portal table entry disabled event .......... 101
Portals
early versions .................... 13
Version 2.0 ............... 13
Version 3.0 ............... 13
portals
addressing ................ see address translation
constants ...................... see constants
constants summary ....... 100
data types ...................... see 36, 97
functions ......................... see functions
functions summary ........ 98
handle ....................... 36
multi-threading ............. 33
naming conventions ........ 35
operations ................ see operations
return codes ................ see return codes
return codes summary ........ 99
scalability .................. 19
semantics ................ 105
sizes ..................... 36
portals.h .................. [14], 28, 31, 54, 66, 81
priority list ................. 14, 33
process ID ............ 27, 29, 37, 41, 48, 49, 59, 62, 64, 82, 86–88, 97
process ....................... 14, 33
process aggregation ...... 49
well known .................... 41
progress ....................... 21
progress rule ............... 18, 21, 121
protected space ............ 24, 25
PT
options ..................... 45
pt_index_req (field) ........ 45
PTL_ACK_REQ (const) ......... 35, 80, 82, 86, 100, 105
PTL_BAND (const) .......... 84
PTL_BOR (const) .......... 84
P

PtlHandleIsEqual (func) ............... 95, 96, 97, 99, 122
PtlInit (func) ............... 35, 37, 38, 39, 99, 100
PtlIsValidHandle (func) ............... 122
PtlLEAppend (func) ............... 54, 57
PtlLEUnlink (func) ............... 54, 58
PtlMDBind (func) ............... 50, 52, 53, 97, 99, 100
PtlMDRelease (func) ............... 50, 53, 97, 99, 100
PtlMDUpdate (func) ............... 122
PtlMEAppend (func) ............... 39, 59, 62, 63, 64, 66, 68, 70, 97–100
PtlMEUnlink (func) ............... 59, 64, 97, 99, 100
PtlNIInit (func) ............... 39, 40, 43, 97, 99, 100
PtlNIHandle (func) ............... 39, 44, 97, 99, 100
PtlNIStatus (func) ............... 39, 40, 41, 42, 43, 97, 99, 100, 119
PtlPut (func) ............... 30, 80, 81, 81, 83, 86–88, 90, 97–100, 106, 107, 109
PtlSetInvalidHandle (func) ............... 122
PtlSwap (func) ............... 80, 83, 88, 91, 94, 97–99
PtlTestAtomic (func) ............... 122
PtlTriggeredAtomic (func) ............... 89, 92, 97–99
PtlTriggeredCTln (func) ............... 91, 95, 99
PtlTriggeredFetchAtomic (func) ............... 93, 97–99
PtlTriggeredGet (func) ............... 89, 90, 91, 97–99
PtlTriggeredPut (func) ............... 90, 95, 97–99
PtlTriggeredSwap (func) ............... 94, 97–99
Puma ............... 18
Pundit, Neil ............... 4
purpose ............... 18
put ............... see operations
put_md_handle (field) ............... 87–89, 92–94, 107, 109

Q

quality implementation ............... 42
quality of implementation ............... 19

R

rank ............... 37, 48, 49
rank (field) ............... 37, 48
README ............... 35, 119
receiver-managed ............... 18
Red Storm ............... 121, 122
reliable communication ............... 19, 113
remote offset ............... see offset
remote_offset (field) ............... 70, 82, 83, 86–88, 90–94, 106–109
reply ............... see operations
return codes ............... 37, 99, 122
PTL_CT_INVALID ............... 53, 78–80, 90–95, 99
PTL_EQ_DROPPED ............... 73–75, 99
PTL_EQ_EMPTY ............... 73, 75, 99
PTL_EQ_INVALID ............... 53, 73–75, 99
PTL_FAIL ............... 38, 96, 99
PTL_HANDLE_INVALID ............... 44, 99
PTL_IFACE_INVALID ............... 42, 99, 122
PTL_LE_IN_USE ............... 58, 59
PTL_LE_INVALID ............... 59
PTL_LE_LIST_TOO_LONG ............... 58
PTL_MD_ILLEGAL ............... 53, 99, 122
PTL_MD_IN_USE ............... 54, 68, 99
PTL_MD_INVALID ............... 54, 82, 83, 86, 87, 89–94, 100
PTL_ME_IN_USE ............... 64, 68, 100
PTL_ME_INVALID ............... 64, 100
PTL_ME_LIST_TOO_LONG ............... 64, 100
PTL_NI_INVALID ............... 43–50, 53, 58, 63, 72, 77, 100
PTL_NI_NOT_LOGICAL ............... 100
PTL_NO_INIT ............... 42–50, 53, 54, 58, 59, 64, 72–75, 77–80, 82, 83, 86, 87, 89–95, 100
PTL_NO_SPACE ............... 42, 53, 58, 64, 72, 77, 100
PTL_OK ............... 35, 38, 42–50, 53, 54, 58, 59, 63, 64, 72–75, 77–80, 82, 83, 86, 87, 89–96, 100
PTL_PID_INUSE ............... 42, 100
PTL_PID_INVALID ............... 42, 100
PTL_PROCESS_INVALID ............... 64, 82, 83, 86, 87, 89–94, 100
PTL_PT_EQ_NEEDED ............... 46, 100
PTL_PT_FULL ............... 46, 100
PTL_PT_IN_USE ............... 46, 100
PTL_PT_INDEX_INVALID ............... 46, 58, 64, 100
PTL_SEGV ............... 37, 42, 44, 48–50, 53, 72–75, 77, 78, 100
PTL_SR_INDEX_INVALID ............... 44, 100

S

scalability ............... 19, 113, 115
guarantee ............... 19
MPI ............... 18
network ............... 18
scatter/gather ............... 51, 52, 56, 60, 61, 97, 102
Schutt, Jim ............... 4, 121
Sears, Mark ............... 4
semantics ............... 105
send ............... 23
send event ............... 81, 84, 101
sequence (field) ............... 71
sequence number ............... 71, 98
SHMEM ............... 18, 20
shmem_fence() ............... 20

132
structure fields and argument names

- ac_id
- actual
- actual_mapping
- atomic_operation
- atomic_type
- count
- ct_handle
- ct_type
- datatype
- desired
- desired_mapping
- eq_handle
- eq_handles
- event
- failure
- get_md_handle
- handle
- handle1
- handle2
- hdr_data
- id
- iface
- ignore_bits
- increment
- initiator
- iov_base
- iov_len
- jid
- le
- le_handle
- length
- local_get_offset
- local_offset
- local_put_offset
- map_size
- match_bits
- match_id
- max_atomic_size
- max_search
- max_csz
- max_eqs
- max_iovecs
- max_mds
- max_me_list
- max_mes
- max_msg_size
- max_pt_index
- md
- md_handle
- me
- min_free
- mlength
- new_ct
- ni_fail_type
- ni_handle
- nid
- offset
- operand
- options
- pid
- pt_index
- pt_index Req
- put_md_handle
- rank
- remote_offset
- rlength
- sequence
- size
- start
- status
- status_register
- success
- target_id
- test
- threshold
- timeout
- trig_ct_handle
- type
- uid
- user_ptr
- which
- success (field)
- summary
- SUNMOS
DISTRIBUTION:

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

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

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

1 MS 0806 Jim Schutt, 4336
1 MS 0817 Doug Doerfler, 1422
1 MS 0817 Sue Kelly, 1422
1 MS 1110 Ron Brightwell, 1423
1 MS 1110 Neil Pundit, 1423
4 MS 1110 Rolf Riesen, 1423
1 MS 1110 Lee Ward, 1423
1 MS 1110 Ron Oldfield, 1423
1 MS 1110 Kevin Pedretti, 1423
1 MS 1110 Keith Underwood, 1422
1 MS 0899 Technical Library, 9536 (electronic)