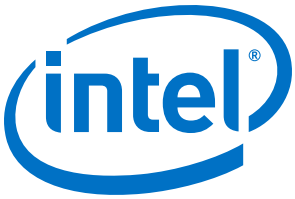


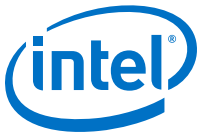
Intel® Performance Scaled Messaging 2 (PSM2)

Programmer's Guide

Rev. 8.0

October 2017

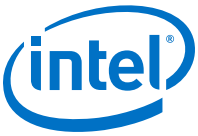
Order No.: H76473-8.0



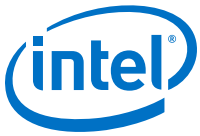
Revision History

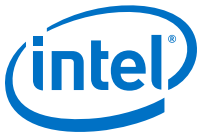
For the latest documentation, go to http://www.intel.com/omnipath/FabricSoftwarePublications.

Date	Revision	Description
October 2017	8.0	No technical changes to document; clerical change only. The *Intel® Omni-Path Fabric Suite FastFabric Command Line Interface Reference Guide* has been merged into the *Intel® Omni-Path Fabric Suite FastFabric User Guide*. See the Intel® Omni-Path Documentation Library for details.
August 2017	7.0	Updates to this document include: • Added Differences between PSM2 and PSM. • Added PSM2 Multi-Endpoint Functionality. • Additions to support multi-endpoint functionality: PSM2_MULTI_EP, psm2_ep_query, psm2_ep_epid_lookup, psm2_ep_epid_lookup2, and psm2_epaddr_to_epid.
April 2017	6.0	Updates to this document include: • Added: PSM2_CCA_PRESCAN, PSM2_CUDA, PSM2_DISABLE_CCA, PSM2_GPUDIRECT, PSM2_GPUDIRECT_RECV_THRESH, PSM2_GPUDIRECT_SEND_THRESH, and PSM2_MAX_PENDING_SDMA_REQS. • Updated: PSM2_MAX_CONTEXTS_PER_JOB, PSM2_MULTIRAIL, and PSM2_MULTIRAIL_MAP. • Added Intel® Omni-Path Documentation Library.
December 2016	5.0	Updates to this document include: • Updated psm2_ep_open_opts_get_defaults to add Return Value PSM2_PARAM_ERR. • Updated psm2_ep_open as follows: added Return Value PSM2_PARAM_ERR, changed default timeout value to 30, and added bullets to Options section. • Updated psm2_ep_open_opts to add fields in rows 7-12. • Added Cluster Configurator for Intel® Omni-Path Fabric.
August 2016	4.0	Updates to this document include: • Added: PSM2_MULTIRAIL, PSM2_MULTIRAIL_MAP, PSM2_PATH_SELECTION. • Updated: PSM2_IB_SERVICE_ID, PSM2_MAX_CONTEXTS_PER_JOB, PSM2_MAX_PENDING_SDMA_REQS, PSM2_MQ_RECVREQS_MAX, PSM2_MTU.
May 2016	3.0	Updates to this document include: • Added Environment Variable: PSM2_MAX_CONTEXTS_PER_JOB. • Deprecated Environment Variable: PSM2_SHAREDCONTEXTS_MAX. • Updated Environment Variable: HFI_NO_CPUAFFINITY.
February 2016	2.0	Updates to this document include: • Added Environment Variables: PSM2_MTU, PSM2_PATH_REC, and PSM2_IB_SERVICE_ID.
November 2015	1.0	Starting with this release, the Intel® PSM2 API library is a stand-alone package with its own documentation.

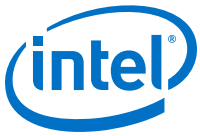


Contents





Tables



Preface

This manual is part of the documentation set for the Intel® Omni-Path Fabric (Intel® OP Fabric), which is an end-to-end solution consisting of Intel® Omni-Path Host Fabric Interfaces (HFIs), Intel® Omni-Path switches, and fabric management and development tools.

The Intel® OP Fabric delivers a platform for the next generation of High-Performance Computing (HPC) systems that is designed to cost-effectively meet the scale, density, and reliability requirements of large-scale HPC clusters.

Both the Intel® OP Fabric and standard InfiniBand* are able to send Internet Protocol (IP) traffic over the fabric, or *IPoFabric*. In this document, however, it is referred to as *IP over IB* or *IPoIB*. From a software point of view, IPoFabric and IPoIB behave the same way and, in fact, use the same `ib_ipoib` driver to send IP traffic over the ib0 and/or ib1 ports.

Intended Audience

The intended audience for the Intel® Omni-Path (Intel® OP) document set is network administrators and other qualified personnel.

Intel® Omni-Path Documentation Library

Intel® Omni-Path publications are available at the following URLs:

- Intel® Omni-Path Switches Installation, User, and Reference Guides

  http://www.intel.com/omnipath/SwitchPublications
- Intel® Omni-Path Software Installation, User, and Reference Guides (includes HFI documents)

  http://www.intel.com/omnipath/FabricSoftwarePublications
- Drivers and Software (including Release Notes)

  http://www.intel.com/omnipath/Downloads

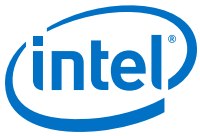
Use the tasks listed in this table to find the corresponding Intel® Omni-Path document.

Task	Document Title	Description
Key: Shading indicates the URL to use for accessing the particular document.		
• Intel® Omni-Path Switches Installation, User, and Reference Guides: http://www.intel.com/omnipath/SwitchPublications		
• Intel® Omni-Path Software Installation, User, and Reference Guides (includes HFI documents): http://www.intel.com/omnipath/FabricSoftwarePublications (no shading)		
• Drivers and Software (including Release Notes): http://www.intel.com/omnipath/Downloads		

continued...

Task	Document Title	Description
Using the Intel® OPA documentation set	*Intel® Omni-Path Fabric Quick Start Guide*	A roadmap to Intel's comprehensive library of publications describing all aspects of the product family. It outlines the most basic steps for getting your Intel® Omni-Path Architecture (Intel® OPA) cluster installed and operational.
Setting up an Intel® OPA cluster	*Intel® Omni-Path Fabric Setup Guide* (Old title: *Intel® Omni-Path Fabric Staging Guide*)	Provides a high level overview of the steps required to stage a customer-based installation of the Intel® Omni-Path Fabric. Procedures and key reference documents, such as Intel® Omni-Path user guides and installation guides are provided to clarify the process. Additional commands and BKMs are defined to facilitate the installation process and troubleshooting.
Installing hardware	*Intel® Omni-Path Fabric Switches Hardware Installation Guide*	Describes the hardware installation and initial configuration tasks for the Intel® Omni-Path Switches 100 Series. This includes: Intel® Omni-Path Edge Switches 100 Series, 24 and 48-port configurable Edge switches, and Intel® Omni-Path Director Class Switches 100 Series.
	Intel® Omni-Path Host Fabric Interface Installation Guide	Contains instructions for installing the HFI in an Intel® OPA cluster. A cluster is defined as a collection of nodes, each attached to a fabric through the Intel interconnect. The Intel® HFI utilizes Intel® Omni-Path switches and cabling.
Installing host software Installing HFI firmware Installing switch firmware (externally-managed switches)	*Intel® Omni-Path Fabric Software Installation Guide*	Describes using a Text-based User Interface (TUI) to guide you through the installation process. You have the option of using command line interface (CLI) commands to perform the installation or install using the Linux* distribution software.
Managing a switch using Chassis Viewer GUI Installing switch firmware (managed switches)	*Intel® Omni-Path Fabric Switches GUI User Guide*	Describes the Intel® Omni-Path Fabric Chassis Viewer graphical user interface (GUI). It provides task-oriented procedures for configuring and managing the Intel® Omni-Path Switch family. Help: GUI online help.
Managing a switch using the CLI Installing switch firmware (managed switches)	*Intel® Omni-Path Fabric Switches Command Line Interface Reference Guide*	Describes the command line interface (CLI) task information for the Intel® Omni-Path Switch family. Help: -help for each CLI.
Managing a fabric using FastFabric	*Intel® Omni-Path Fabric Suite FastFabric User Guide* (Merged with: *Intel® Omni-Path Fabric Suite FastFabric Command Line Interface Reference Guide*)	Provides instructions for using the set of fabric management tools designed to simplify and optimize common fabric management tasks. The management tools consist of TUI menus and command line interface (CLI) commands. Help: -help and man pages for each CLI. Also, all host CLI commands can be accessed as console help in the Fabric Manager GUI.
Managing a fabric using Fabric Manager	*Intel® Omni-Path Fabric Suite Fabric Manager User Guide*	The Fabric Manager uses a well defined management protocol to communicate with management agents in every Intel® Omni-Path Host Fabric Interface (HFI) and switch. Through these interfaces the Fabric Manager is able to discover, configure, and monitor the fabric.
	Intel® Omni-Path Fabric Suite Fabric Manager GUI User Guide	Provides an intuitive, scalable dashboard and set of analysis tools for graphically monitoring fabric status and configuration. It is a user-friendly alternative to traditional command-line tools for day-to-day monitoring of fabric health. Help: Fabric Manager GUI Online Help.

continued...



Task	Document Title	Description
Configuring and administering Intel® HFI and IPoIB driver Running MPI applications on Intel® OPA	*Intel® Omni-Path Fabric Host Software User Guide*	Describes how to set up and administer the Host Fabric Interface (HFI) after the software has been installed. The audience for this document includes both cluster administrators and Message-Passing Interface (MPI) application programmers, who have different but overlapping interests in the details of the technology.
Writing and running middleware that uses Intel® OPA	*Intel® Performance Scaled Messaging 2 (PSM2) Programmer's Guide*	Provides a reference for programmers working with the Intel® PSM2 Application Programming Interface (API). The Performance Scaled Messaging 2 API (PSM2 API) is a low-level user-level communications interface.
Optimizing system performance	*Intel® Omni-Path Fabric Performance Tuning User Guide*	Describes BIOS settings and parameters that have been shown to ensure best performance, or make performance more consistent, on Intel® Omni-Path Architecture. If you are interested in benchmarking the performance of your system, these tips may help you obtain better performance.
Designing an IP or storage router on Intel® OPA	*Intel® Omni-Path IP and Storage Router Design Guide*	Describes how to install, configure, and administer an IPoIB router solution (Linux* IP or LNet) for inter-operating between Intel® Omni-Path and a legacy InfiniBand* fabric.
Building a Lustre* Server using Intel® OPA	*Building Lustre* Servers with Intel® Omni-Path Architecture Application Note*	Describes the steps to build and test a Lustre* system (MGS, MDT, MDS, OSS, OST, client) from the HPDD master branch on a x86_64, RHEL*/CentOS* 7.1 machine.
Building Containers for Intel® OPA fabrics	*Building Containers for Intel® Omni-Path Fabrics using Docker* and Singularity* Application Note*	Provides basic information for building and running Docker* and Singularity* containers on Linux*-based computer platforms that incorporate Intel® Omni-Path networking technology.
Writing management applications that interface with Intel® OPA	*Intel® Omni-Path Management API Programmer's Guide*	Contains a reference for programmers working with the Intel® Omni-Path Architecture Management (Intel OPAMGT) Application Programming Interface (API). The Intel OPAMGT API is a C-API permitting in-band and out-of-band queries of the FM's Subnet Administrator and Performance Administrator.
Learning about new release features, open issues, and resolved issues for a particular release	*Intel® Omni-Path Fabric Software Release Notes*	
	Intel® Omni-Path Fabric Manager GUI Release Notes	
	Intel® Omni-Path Fabric Switches Release Notes (includes managed and externally-managed switches)	

Cluster Configurator for Intel® Omni-Path Fabric

The Cluster Configurator for Intel® Omni-Path Fabric is available at: http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-configurator.html.

This tool generates sample cluster configurations based on key cluster attributes, including a side-by-side comparison of up to four cluster configurations. The tool also generates parts lists and cluster diagrams.

Documentation Conventions

The following conventions are standard for Intel® Omni-Path documentation:

- *Note:* provides additional information.
- ***Caution:*** indicates the presence of a hazard that has the potential of causing damage to data or equipment.

- ***Warning:*** indicates the presence of a hazard that has the potential of causing personal injury.
- Text in blue font indicates a hyperlink (jump) to a figure, table, or section in this guide. Links to websites are also shown in blue. For example:

  See License Agreements on page 10 for more information.

  For more information, visit www.intel.com.
- Text in **bold** font indicates user interface elements such as menu items, buttons, check boxes, key names, key strokes, or column headings. For example:

  Click the **Start** button, point to **Programs**, point to **Accessories**, and then click **Command Prompt**.

  Press **CTRL+P** and then press the **UP ARROW** key.
- Text in `Courier` font indicates a file name, directory path, or command line text. For example:

  Enter the following command: `sh ./install.bin`
- Text in *italics* indicates terms, emphasis, variables, or document titles. For example:

  Refer to *Intel® Omni-Path Fabric Software Installation Guide* for details.

  In this document, the term *chassis* refers to a managed switch.

Procedures and information may be marked with one of the following qualifications:

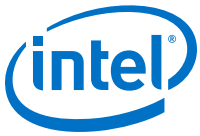
- **(Linux)** – Tasks are only applicable when Linux* is being used.
- **(Host)** – Tasks are only applicable when Intel® Omni-Path Fabric Host Software or Intel® Omni-Path Fabric Suite is being used on the hosts.
- **(Switch)** – Tasks are applicable only when Intel® Omni-Path Switches or Chassis are being used.
- Tasks that are generally applicable to all environments are not marked.

License Agreements

This software is provided under one or more license agreements. Please refer to the license agreement(s) provided with the software for specific detail. Do not install or use the software until you have carefully read and agree to the terms and conditions of the license agreement(s). By loading or using the software, you agree to the terms of the license agreement(s). If you do not wish to so agree, do not install or use the software.

Technical Support

Technical support for Intel® Omni-Path products is available 24 hours a day, 365 days a year. Please contact Intel Customer Support or visit http://www.intel.com/omnipath/support for additional detail.



1.0 Intel® PSM2 API

This manual is a reference for programmers working with the Intel® PSM2 Application Programming Interface (API). The Performance Scaled Messaging 2 API (PSM2 API) is a low-level user-level communications interface.

For details about the other documents for the Intel® Omni-Path product line, refer to Intel® Omni-Path Documentation Library on page 7.

1.1 Introduction

The Intel® Performance Scaled Messaging 2 (Intel® PSM2) API is a high-performance, vendor-specific protocol that provides a low-level communications interface for the Intel® Omni-Path family of products. PSM2 enables mechanisms necessary to implement higher level communications interfaces in parallel environments.

PSM2 targets clusters of multicore processors and transparently implements two levels of communication: inter-node communication and intra-node shared memory communication.

Differences between PSM2 and PSM

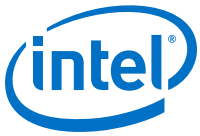
The Intel® PSM2 interface differs from the Intel® True Scale PSM interface in the following ways:

- PSM2 includes new features and optimizations for Intel® Omni-Path hardware and processors.
- The PSM2 API was ported to directly use Intel® Omni-Path hardware, because PSM2 uses kernel bypass mode to achieve higher performance.
- PSM2 supports a larger 96-bit tag format, while Intel® True Scale PSM only supports 64-bit tags.
- PSM2 includes performance improvements specific to Intel® OPA and larger workloads.
- PSM2 adjusted the field width for job rank numbers to accommodate jobs larger than 64K ranks.
- PSM2 is actively under development and will continue to improve on Intel® OPA platforms, while Intel® True Scale PSM is a legacy product which is maintained for bug fixes only.

1.2 Compatibility

PSM2 can coexist with other Intel software distributions, such as OpenFabrics, which allows applications to simultaneously target PSM2-based and non-PSM2-based applications on a single node without changing any system-level configuration.

However, unless otherwise noted, PSM2 does not support running PSM2-based and non-PSM2-based communication within the same user process.



PSM2 is currently a single-threaded library. This means that you cannot make any concurrent PSM2 library calls. While threads may be a valid execution model for the wider set of potential PSM2 clients, applications should currently expect better effective use of Intel® Omni-Path resources (and hence better performance) by dedicating a single PSM2 communication endpoint to every CPU core.

Except where noted, PSM2 does not assume a single program, multiple data (SPMD) parallel model, and extends to multiple program, multiple data (MPMD) environments in specific areas. However, PSM2 assumes the runtime environment to be homogeneous on all nodes in bit width (64-bit only) and endianness (little or big), and fails at startup if any of these assumptions do not hold.

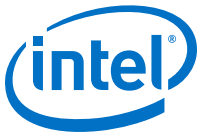
1.3 Endpoint Communication Model

PSM2 follows an endpoint communication model where an endpoint is defined as an object (or handle) instantiated to support sending and receiving messages to other endpoints. In order to prevent PSM2 from being tied to a particular parallel model (such as SPMD), you retain control over the parallel layout of endpoints. Opening endpoints (`psm2_ep_open`) and connecting endpoints to enable communication (`psm2_ep_connect`) are two decoupled mechanisms. If you do not dynamically change the number of endpoints beyond parallel startup, you can combine both mechanisms at startup. If you wish to manipulate the location and amount of endpoints at runtime, you can do so by explicitly connecting sets or subsets of endpoints.

As a side effect, this greater flexibility allows you to manage a two-stage initialization process. In the first stage of opening an endpoint (`psm2_ep_open`), you obtain an opaque handle to the endpoint and a globally distributable endpoint identifier (`psm2_epid_t`). Prior to the second stage of connecting endpoints (`psm2_ep_connect`), you must distribute all relevant endpoint identifiers through an out-of-band mechanism. Once the endpoint identifiers are successfully distributed to all processes that wish to communicate, you connect all endpoint identifiers to the locally opened endpoint (`psm2_ep_connect`). In connecting the endpoints, you obtain an opaque endpoint address (`psm2_epaddr_t`), which is required for all PSM2 communication primitives.

1.4 PSM2 Components

PSM2 exposes a single endpoint initialization model, but enables various levels of communication functionality and semantics through components. The first major component available in PSM2 is PSM2 Matched Queues (Intel® PSM2 Component Documentation on page 23). Matched Queues (MQ) present a queue-based communication model with the distinction that queue consumers use a 3-tuple of metadata to match incoming messages against a list of preposted receive buffers. The MQ semantics are sufficiently akin to MPI to cover the entire MPI-1.2 standard. With future releases of the PSM2 interface, more components may be exposed to accommodate users who implement parallel communication models that deviate from the Matched Queue semantics.



1.5 PSM2 Multi-Endpoint Functionality

PSM2 Multi-Endpoint (Multi-EP) functionality is part of the PSM2 API library, however, it is not default behavior and must be enabled using the PSM2_MULTI_EP environment variable.

By default, only one PSM2 endpoint may be opened in a process or MPI rank. Enabling PSM2_MULTI_EP allows more than one PSM2 endpoint to be opened in a single process and expands the behavior of several APIs, including `psm2_init`, `psm2_ep_open`, and the `psm2_mq_*` APIs listed below.

PSM2 has added minimal thread safety for using with Multi-EP in a performant manner. Along with each EP (endpoint) created, an associated MQ (matched queue) is created, which tracks message completion and ordering. The following APIs have been made thread-safe to allow for multiple threaded access, assuming each is called with a different MQ.

Table 1. Intel® PSM2 Thread-Safe APIs

psm2_mq_cancel	psm2_mq_iprobe	psm2_mq_send
psm2_mq_improbe	psm2_mq_iprobe2	psm2_mq_send2
psm2_mq_improbe2	psm2_mq_irecv	psm2_mq_test
psm2_mq_imrecv	psm2_mq_irecv2	psm2_mq_test2
psm2_mq_ipeek	psm2_mq_isend	psm2_mq_wait
psm2_mq_ipeek2	psm2_mq_isend2	psm2_mq_wait2

Limitation

By default, PSM2 allows hardware context sharing to increase the number of local ranks. This feature requires that the total number of connections is specified at job startup. Since the Multi-EP feature allows the middleware or end user to dynamically create and teardown endpoints, context sharing is disabled while Multi-EP is enabled. This limits the number of local MPI ranks to the number of real hardware resources exposed by the Intel® Omni-Path hfi1 driver. More information can be obtained on this topic in the *Intel® Omni-Path Fabric Performance Tuning User Guide* and *Intel® Omni-Path Fabric Host Software User Guide*. However, by default, the number of endpoints that can be opened is limited to the number of real CPU cores present on the machine.

Related Information

- **Intel® MPI Library Multi-Thread (MT)**

  Intel® MPI MT design is motivated by the need to improve communication throughput and concurrency in hybrid MPI applications on Intel hardware, particularly when using Intel® Omni-Path Architecture (Intel® OPA). However, the design is universal, so it can be used on any other hardware that is supported with specific abstractions (Scalable Endpoints). The design is entirely based on the Open Fabric Interface (OFI) libfabric concept of Scalable Endpoints (SEP).

  For details, go to: https://software.intel.com/en-us/intel-mpi-library/documentation

- **OpenFabrics Alliance* (OFA) Open Fabric Interfaces libfabric**

  Starting with libfabric 1.5.0 release, the psm2 provider supports scalable endpoints when running over newer PSM2 libraries that have the multi-EP feature enabled. When the psm2 provider is initialized, it checks the feature set of the

underlying PSM2 library and turns on/off the scalable endpoint support automatically. This is an unconditional dependency and the scalable endpoint support does not work with older PSM2 libraries.

For details, go to: https://ofiwg.github.io/libfabric/

1.6 PSM2 Communication Progress Guarantees

PSM2 internally ensures progress of both intra-node and inter-node messages, but not autonomously. This means that while performance does not depend greatly on how you decide to schedule communication progress, explicit progress calls are required for correctness. The `psm2_poll` function is available to make progress over all PSM2 components in a generic manner. For more information on making progress over many communication operations in the MQ component, see MQ Progress Requirements on page 26.

1.7 PSM2 Completion Semantics

PSM2 currently only implements the MQ component, which documents its own message completion semantics (see MQ Completion Semantics on page 25).

1.8 PSM2 Error Handling

PSM2 exposes a list of user and runtime errors enumerated in `psm2_error`. While most errors are fatal in that you are not expected to be able to recover from them, PSM2 still allows some level of control. By default, PSM2 returns all errors, but as a convenience, allows you to either defer errors internally to PSM2 or to have PSM2 call a user-provided error callback function.

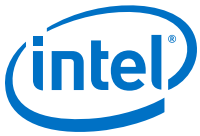
PSM2 attempts to deallocate its resources as a best effort, but exits are always non-collective with respect to endpoints opened in other processes. You are expected to be able to handle non-collective exits from any endpoint and cleanly and independently terminate the parallel environment.

Local error handling can be handled in three modes, two of which are predefined PSM2 mechanisms:

- PSM2-internal error handler (`PSM2_ERRHANDLER_PSM_HANDLER`)
- No-op PSM2 error handler where errors are returned (`PSM2_ERRHANDLER_NO_HANDLER`)
- User-registered error handlers

The default PSM2-internal error handler effectively frees you from explicitly handling the return values of every PSM2 function, but may not return in a function determined to have caused a fatal error.

The No-op PSM2 error handler bypasses all error handling functionality and always returns the error. You can then use `psm2_error_get_string` to obtain a generic string from an error code (compared to a more detailed error message available through registering of error handlers).



For even more control, you can register your own error handlers to have access to more precise error strings and selectively control when and when not to return to callers of PSM2 functions. All error handlers shown defer error handling to PSM2 for errors that are not recognized using `psm2_error_defer`. Deferring an error from a custom error handler is equivalent to relying on the default error handler.

Errors and error handling can be individually registered either globally or per-endpoint:

- **Per-endpoint** error handling captures errors for functions where the error scoping is determined to be over an endpoint. This includes all communication functions that include an EP or MQ handle as the first parameter.
- **Global** error handling captures errors for functions where a particular endpoint cannot be identified or for `psm2_ep_open`, where errors (if any) occur before the endpoint is opened.

Error handling is controlled by registering error handlers (`psm2_error_register_handler`). The global error handler can be set at any time (even before `psm2_init`), whereas a per-endpoint error handler can be set as soon as a new endpoint is successfully created. If a per-endpoint handle is not registered, the per-endpoint handler inherits from the global error handler at time of open.

1.9 Environment Variables

This section describes how to control PSM2 behavior using environment variables.

1.9.1 PSM2_CCA_PRESCAN

Enables Congestion Control Prescanning when set. Can improve the response time of the PSM2 software stack by prescanning packet headers for network notification for congestion. This will slightly increase CPU usage of the local rank, but may provide faster response to notification and thus less congestion and more fair play within the network.

Options:

- `1` enabled
- `0` disabled (default)

Default: `PSM2_CCA_PRESCAN=0` (disabled)

1.9.2 PSM2_CUDA

Enables CUDA* support in PSM2 when set. Requires libpsm2 to be compiled with CUDA* support.

For additional details, see the *Intel® Omni-Path Fabric Performance Tuning User Guide*.

Note: If GPU buffers are used in the workloads and `PSM2_CUDA` is not set to 1, undefined behavior will result.

Default: `PSM2_CUDA=0`

1.9.3 PSM2_DEVICES

PSM2 implements the following devices for communication: `self`, `shm`, and `hfi`. For PSM2 jobs that do not require shared-memory communications, `PSM2_DEVICES` can be specified as `self`, `hfi`. Similarly, for shared-memory only jobs, the `hfi` device can be disabled. You must ensure that the endpoint IDs passed in `psm2_ep_connect` do not require a device that has been explicitly disabled. In some instances, enabling only the devices that are required may improve performance.

Default: `PSM2_DEVICES="self,shm,hfi"`

For shared-memory only jobs: `PSM2_DEVICES="shm,self"`

1.9.4 PSM2_DISABLE_CCA

Disables use of Congestion Control Architecture (CCA).

Options:

- `1` disabled
- `0` enabled (default)

Default: `PSM2_DISABLE_CCA=0` (enabled)

1.9.5 PSM2_GPUDIRECT

GPUDirect* RDMA is a technology that enables a direct path for data exchange between a graphics processing unit (GPU) and a third-party peer device using standard features of PCI Express. For more information, see the NVIDIA* CUDA* toolkit documentation: http://docs.nvidia.com/cuda/gpudirect-rdma/index.html

Enables GPUDirect* RDMA support when set and allows direct data exchange between GPU and HFI. For complete operation, you also need the appropriate hfi1 driver support. For details, see the *Intel® Omni-Path Fabric Software Installation Guide*.

Default: `PSM2_GPUDIRECT=0`

1.9.6 PSM2_GPUDIRECT_RECV_THRESH

Allows you to specify a threshold value (in bytes). If the threshold is exceeded, the GPUDirect* RDMA feature will not be used on the receive side of a connection.

Range: 0 to (2^{32}-1)

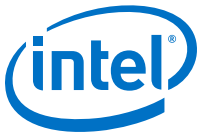
Default: `PSM2_GPUDIRECT_RECV_THRESH=0`

1.9.7 PSM2_GPUDIRECT_SEND_THRESH

Allows you to specify a threshold value (in bytes). If the threshold is exceeded, the GPUDirect* RDMA feature will not be used on the send side of a connection.

Range: 1 to (2^{32}-1) Note that 0 is invalid.

Default: `PSM2_GPUDIRECT_SEND_THRESH=30000`



1.9.8 PSM2_IB_SERVICE_ID

Sets IB Service ID for path resolution. Using this overrides value set by the options used by applications or upper layer transports.

If you pass in a value with `psm2_ep_open` in the `psm2_ep_open_opts` structure, then the default of `HFI_DEFAULT_SERVICE_ID` or `0x1000117500000000ULL` is replaced. If the environment variable here is listed, it replaces the default or any value passed in.

Default: `PSM2_IB_SERVICE_ID=0x1000117500000000ULL`

1.9.9 PSM2_MAX_CONTEXTS_PER_JOB

Maximum number of contexts that a job opens.

If required for resource sharing in batch systems, users can restrict the number of Intel® Omni-Path contexts that are made available on each node of an MPI job by setting that number in the `PSM2_MAX_CONTEXTS_PER_JOB` environment variable. The default is to use all possible contexts.

Default: `PSM2_MAX_CONTEXTS_PER_JOB=all available`

1.9.10 PSM2_MAX_PENDING_SDMA_REQS

Sets maximum pending SDMA requests.

Range = 8 to `sdma_comp_size - 1`, where `sdma_comp_size` is the number of entries in the SDMA request ring. Any other value is replaced with the default value.

Default: `PSM2_MAX_PENDING_SDMA_REQS=sdma_comp_size - 1`

1.9.11 PSM2_MEMORY

Memory usage mode. Controls the amount of memory used for MQ entries by setting the number of entries. Setting this value also sets PSM2_MQ_RECVREQS_MAX and PSM2_MQ_RNDV_HFI_THRESH to preset internal values, see Options for details.

Options:

Note: You must enter the desired option as text, not a numerical value.

- `min` = reserves memory to hold 65536 pending requests
- `normal` = reserves memory to hold 1048576 pending requests
- `large` = reserves memory to hold 16777216 pending requests

Default: `PSM2_MEMORY=normal`

1.9.12 PSM2_MQ_RECVREQS_MAX

Sets the maximum number of `irecv` requests pending completion. Setting this value overrides the PSM2_MAX_PENDING_SDMA_REQS default for any mode.

Default: `PSM2_MQ_RECVREQS_MAX=1048576`

1.9.13 PSM2_MQ_RNDV_HFI_THRESH

Sets the threshold (in bytes) for the `hfi` eager-to-rendezvous switchover.

Default: `PSM2_MQ_RNDV_HFI_THRESH=64000`

1.9.14 PSM2_MQ_RNDV_SHM_THRESH

Sets the threshold (in bytes) for shared memory eager-to-rendezvous switchover.

Default: `PSM2_MQ_RNDV_SHM_THRESH=16000`

1.9.15 PSM2_MQ_SENDREQS_MAX

Sets the maximum number of `isend` requests pending completion. Setting this value overrides the PSM2_MAX_PENDING_SDMA_REQS default for any mode.

Default: `PSM2_MQ_SENDREQS_MAX=1048576`

1.9.16 PSM2_MTU

Sets PSM2 MTU to user-specified size, if defined. The default behavior is controlled by driver or switch. PSM2 does not query the path record unless `PSM2_PATH_REC` is enabled. This environment variable, when defined, overrides the path record value only allowing selections of MTU values equal to or less than that maximum indicated by the path records.

Valid values are 1-7, 256-8192, 10240. Using bad values will silently use the smaller of the internal default of 8192 or the network configured value. Values 1-7 are indexes into this table:

- `1` = 256
- `2` = 512
- `3` = 1024
- `4` = 2048
- `5` = 4096
- `6` = 8192
- `7` = 10240

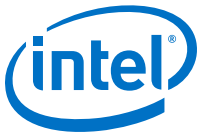
Default: `PSM2_MTU=Automatic based on network configs, typically 8192.`

1.9.17 PSM2_MULTI_EP

Enables more than one PSM2 endpoint to be opened in a process.

Options:

- `0` Disabled (default).
- `1` Enabled.



1.9.18 PSM2_MULTIRAIL

Enables multi-rail capability so a process can use multiple network interface cards to transfer messages. The PSM2 multi-rail feature can be applied to a single fabric with multiple ports (multiple HFIs), or multiple fabrics.

Options:

- `0` Multi-rail capability disabled (default for single rank jobs).
- `1` Enable multi-rail capability and use all available HFI(s) in the system.
- `2` Enable multi-rail within a single NUMA socket capability.

  PSM2 looks for at least one available HFI(s) in the same NUMA socket on which you pin the task. If no such HFIs are found, PSM2 falls back to `PSM2_MULTIRAIL=1` behavior and uses any other available HFI(s). You are responsible for physical placement of HFI(s). Job launchers, middleware, and end users are responsible for correctly affinitizing MPI ranks and processes for best performance. For details, see the *Intel® Omni-Path Fabric Performance Tuning User Guide*.

Default: `PSM2_MULTIRAIL=0x0=Disabled` (multi-rail is not supported)

1.9.19 PSM2_MULTIRAIL_MAP

Tells PSM2 which unit/port pair is used to set up a rail.

If only one rail is specified, it is equivalent to a single-rail case. The Unit/Port is specified instead of using Unit/Port assigned by the `hfi1` driver. PSM2 scans the above pattern until a violation or error is encountered, and uses the information it has gathered.

Note: `PSM2_MULTIRAIL_MAP` overrides any auto-selection and affinity logic in PSM2, regardless of whether PSM2_MULTIRAIL on page 19 is set to 1 or 2. For details, see the *Intel® Omni-Path Fabric Performance Tuning User Guide*.

Options: `unit:port,unit:port,unit:port,...`

- `unit` starts from 0.
- `port` is always 1.
- Multiple specifications are separated by a comma.

1.9.20 PSM2_PATH_REC

Sets mechanism to query HFI path record.

Options:

- `NONE` Default same as previous instances. Utilizes static data.
- `OPP` Use OFED Plus Plus library to do path record queries.
- `UMAD` Use raw libibumad interface to form and process path records.

Default: `PSM2_PATH_REC=NONE`

1.9.21 PSM2_PATH_SELECTION

Policy to use if multiple paths are available between endpoints. For details, see the *Intel® Omni-Path Fabric Host Software User Guide*, Routing section.

Options:

- `adaptive`
- `static_src`
- `static_dest`
- `static_base`

Default: `PSM2_PATH_SELECTION=adaptive`

1.9.22 PSM2_RANKS_PER_CONTEXT

Provides an alternate way of specifying how PSM should use contexts. The variable is the number of ranks that share each hardware context. The supported values include:

- `1` no context sharing
- `2` 2-way context sharing
- `3` 3-way context sharing
- `4` 4-way context sharing
- `8` 8-way context sharing (maximum)

The same value of `PSM2_RANKS_PER_CONTEXT` must be used for all ranks on a node, and typically, you use the same value for all nodes in that job.

Default:

If this value is not set, then by default PSM2 assigns one context per rank when possible. However, if too many MPI ranks are present, then context sharing is enabled to be able to give each rank a portion of a context. The value is determined by the number of ranks present at job launch. Since context sharing impacts performance by way of limiting queue sizes, PSM2 only enables the minimum required level of context sharing to evenly spread the ranks among the contexts and retain what performance is possible.

1.9.23 PSM2_RCVTHREAD

PSM2 uses an extra background thread per rank to make MPI communication progress more efficiently. This thread does not aggressively compete with resources against the main computation thread, but can be disabled by setting `PSM2_RCVTHREAD=0`.

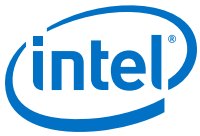
Default: `PSM2_RCVTHREAD=0x1`

1.9.24 PSM2_SHAREDCONTEXTS

Enable shared contexts. Context sharing is on by default.

Default (either option works):

- `PSM2_SHAREDCONTEXTS=1`



- `PSM2_SHAREDCONTEXTS=YES`

To explicitly disable context sharing, set this environment variable in one of the two following ways:

- `PSM2_SHAREDCONTEXTS=0`
- `PSM2_SHAREDCONTEXTS=NO`

1.9.25 PSM2_SHAREDCONTEXTS_MAX

Deprecated.

See PSM2_MAX_CONTEXTS_PER_JOB for details.

1.9.26 PSM2_TID

TID (Token ID) protocol flags. A value of 0 disables the protocol.

Default: `PSM2_TID=0x1`

1.9.27 PSM2_TRACEMASK

Depending on the value of the tracemask, various parts of PSM2 output debugging information. With a default value of 0x1, informative messages are printed; this value should be considered a minimum. At 0x101, startup and finalization messages are added to the output. At 0x1c3, every communication event is logged and should hence be used for extreme debugging only.

Default: `PSM2_TRACEMASK=0x1`

1.10 HFI Environment Variables

The following HFI environment variables are also related to PSM2 functionality.

1.10.1 HFI_DISABLE_MMAP_MALLOC

Disable `mmap` for `malloc()`.

Uses `glibc mallopt()` to disable all uses of `mmap` by setting `M_MMAP_MAX` to 0 and `M_TRIM_THRESHOLD` to -1. Refer to the Linux* man page for `mallopt()` for details.

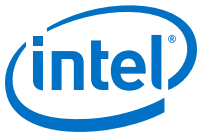
Default: `HFI_DISABLE_MMAP_MALLOC=NO`

Note: Choosing `YES` may reduce the memory footprint required by your program, at the potential expense of increasing CPU overhead associated with memory allocation and memory freeing. The default `NO` option is better for performance.

1.10.2 HFI_NO_CPUAFFINITY

Prevents PSM2 from setting affinity.

During initialization with `HFI_NO_CPUAFFINITY` unset, if the "affinity" option is passed to the `psm2_ep_open()` call, PSM2 may set affinity based on the affinity hints from the driver.



With `HFI_NO_CPUAFFINITY` set, PSM2 does not set affinity regardless of the aforementioned "affinity" option. This allows either user applications to control affinity or the OS to automatically choose affinity.

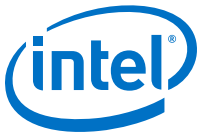
Default: `HFI_NO_CPUAFFINITY` is unset.

1.10.3 HFI_UNIT

Device Unit number. Used to restrict the number of contexts used on an Intel® Omni-Path unit. When context sharing is enabled on a system with multiple Intel® Omni-Path boards (units) and the `HFI_UNIT` environment variable is set, the number of Intel® Omni-Path contexts made available to MPI jobs are restricted to the number of contexts available on that unit.

Note: The Intel® PSM2 implementation has a limit of four (4) HFIs.

Default: `HFI_UNIT` is unset. All available contexts from all units are autodetected and used, and are made available to MPI jobs.



2.0 Intel® PSM2 Component Documentation

The Intel® PSM2 Matched Queues (MQ) interface implements a queue-based communication model with the distinction that queue message consumers use a 3-tuple of metadata to match incoming messages against a list of preposted receive buffers. These semantics are consistent with those presented by MPI-1.2, and all the features and side-effects of message passing find their way into matched queues.

There is currently a single MQ context. If need be, MQs may expose a function to allocate more than one MQ context in the future. Since an MQ is implicitly bound to a locally opened endpoint handle, all MQ functions use an MQ handle instead of an EP handle as a communication context.

2.1 MQ Tag Matching

Note: Tag matching is different in PSM2 compared to the original version. PSM2 tags are 96-bit values of type `psm2_mq_tag_t`. The behavior of send and receive tags and tag selectors is the same, and any 64-bit tags used in existing code are automatically padded to 96 bits within PSM2. The functions designed for 64-bit tags remain in PSM2 and can exist within the same program. Since these two types of functions can operate on the same MQ, care should be taken to avoid unintentional tag matches. Intel recommends that you use a single tag size within a single program.

Users of PSM2 can interpret the 96-bit tag type as a sequence of three 32-bit integers, or any other convenient interpretation scheme. The extended tags can be helpful in high node-count environments.

A successful MQ tag match requires a 3-tuple of unsigned 96-bit ints, two of which are provided by the receiver when posting a receive buffer (`psm2_mq_irecv` and `psm2_mq_irecv2`) and the last is provided by the sender as part of every message sent (`psm2_mq_send` and `psm2_mq_isend`). Since MQ is a receiver-directed communication model, the tag matching done at the receiver involves matching a sent message send tag (`stag`) with the tag (`rtag`) and tag selector (`rtagsel`) attached to every preposted receive buffer. The incoming stag is compared to the posted `rtag` but only for significant bits set in the `rtagsel`. The `rtagsel` can be used to mask off parts (or even all) of the bitwise comparison between sender and receiver tags. A successful match causes the message to be received into the buffer with which the tag is matched. If the incoming message is too large, it is truncated to the size of the posted receive buffer. The bitwise operation corresponding to a successful match and receipt of an expected message amounts to the following expression evaluating as true:

```
((stag ^ rtag) & rtagsel) == 0
```

You must encode (pack) into the 96-bit unsigned integers, including employing the `rtagsel` tag selector as a method to wildcard part or all of the bits significant in the tag matching operation. For example, MPI could use a triple based on context (MPI communicator), source rank, and send tag.

Note: The following code example will be updated in a future release of this document.

The following code example shows how the triple can be packed into 64 bits:

```
// 64-bit send tag formed by packing the triple:
// ( context_id_16bits | source_rank_16bits | send_tag_32bits )

stag = ( (((context_id)&0xffffULL)<<48)|    \
(((source_rank)&0xffffULL)<<32)|    \
(((send_tag)&0xffffffffULL)) );
```

Similarly, the receiver applies the `rtag` matching bits and `rtagsel` masking bits against a list of send tags and returns the first successful match. Zero bits in the `tagsel` can be used to indicate wildcarded bits in the 64-bit tag, which can be useful for implementing MPI's `MPI_ANY_SOURCE` and `MPI_ANY_TAG`. Following the example bit splicing in the previous `stag` example:

```
// Example MPI implementation
// where MPI_COMM_WORLD implemented as 0x3333
// MPI_Irecv source_rank=MPI_ANY_SOURCE,
// tag=7, comm=MPI_COMM_WORLD

rtag = 0x3333000000000007;
rtagsel = 0xffff0000ffffffff;

// MPI_Irecv source_rank=3, tag=MPI_ANY_TAG,
// comm=MPI_COMM_WORLD

rtag = 0x3333000300000000;
rtagsel = 0xffffffff80000000; // can’t ignore sign bit in tag

// MPI_Irecv source_rank=MPI_ANY_SOURCE,
// tag=MPI_ANY_TAG, comm=MPI_COMM_WORLD

rtag = 0x3333000000000000;
rtagsel = 0xffff000080000000; // can’t ignore sign bit in tag
```

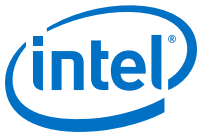
Applications that do not follow tag matching semantics can simply always pass a value of 0 for `rtagsel`, which always yields a successful match to the first preposted buffer. If a message cannot be matched to any of the preposted buffers, the message is delivered as an unexpected message.

2.2 MQ Message Reception

MQ messages are either received as expected or unexpected:

- The received message is expected if the incoming message tag matches the combination of tag and tag selector of at least one of the user-provided receive buffers preposted with `psm2_mq_irecv` or `psm2_mq_irecv2`.
- The received message is unexpected if the incoming message tag doesn't match any combination of tag and tag selector from all the user-provided receive buffers preposted with `psm2_mq_irecv or psm2_mq_irecv2`.

The difference between `psm2_mq_irecv()` and `psm2_mq_irecv2()` is that `psm2_mq_irecv()` does not specify where the message should come from; it purely relies on the tag matching mechanism and the message could come from any other source process. However, `psm2_mq_irecv2()` has an additional argument to specify the source process, where only messages from this specified process can match the



receiving operation. One special case for `psm2_mq_irecv2()` is to specify `PSM2_MQ_TAG_ANY` for the source process argument, which is equivalent to `psm2_mq_irecv()`. Therefore, `psm2_mq_irecv()` is equivalent to a call to `psm2_mq_irecv2()` with `PSM2_MQ_TAG_ANY` as the source value.

Unexpected messages are messages buffered by the MQ library until a receive buffer that can match the unexpected message is provided. With Matched Queues and MPI alike, unexpected messages can occur as a side-effect of the programming model, whereby the arrival of messages can be slightly out of step with receive buffer ordering. Unexpected messages can also be triggered by the difference between the rate at which a sender produces messages and the rate at which a paired receiver can post buffers and hence consume the messages.

In all cases, too many unexpected messages can negatively affect performance. Use some of the following mechanisms to reduce the effect of added memory allocations and copies that result from unexpected messages:

- If and when possible, receive buffers should be posted as early as possible and ideally before calling into the progress engine.
- Use rendezvous messaging that can be controlled with `PSM2_MQ_RNDV_HFI_SZ` and `PSM2_MQ_RNDV_SHM_SZ` options. These options default to values determined to make effective use of bandwidth, and hence not advisable for all communication message sizes. However, rendezvous messaging inherently prevents unexpected messages by synchronizing the sender with the receiver.
- The amount of memory that is allocated to handle unexpected messages can be bounded by adjusting the Global `PSM2_MQ_MAX_SYSBUF_MBYTES` option.
- MQ statistics, such as the amount of received unexpected messages and the aggregate amount of unexpected bytes are available in the `psm2_mq_stats` structure.

Whenever a match occurs, whether the message is expected or unexpected, you must ensure that the message is not truncated. Message truncation occurs when the size of the preposted buffer is less than the size of the incoming matched message. MQ correctly handles message truncation by always copying the appropriate amount of bytes as to not overwrite any data. While it is valid to send less data than the amount of data that has been preposted, messages that are truncated are marked `PSM2_MQ_TRUNCATION` as part of the error code in the message status structure (`psm2_mq_status_t`).

The `psm2_mq_status_t` structure also returns the source ID of the message. During PSM2 initialization time, each process registers an application interpreted ID. When a message from that process is received by any other process, the application interpreted ID is returned in the status structure so that application can interpret where the message comes from. The source ID is returned in the status structure, regardless of which receiving function is used to receive the message. If a process did not register such ID, the default ID is -1.

2.3 MQ Completion Semantics

Message completion in Matched Queues follows local completion semantics. When sending an MQ message, it is deemed complete when MQ guarantees that the source data has been sent and that the entire input source data memory location can be safely overwritten. As with standard Message Passing, MQ does not make any remote completion guarantees for sends. MQ does however, allow a sender to synchronize

with a receiver to send a synchronous message which sends a message only after a matching receive buffer has been posted by the receiver (`PSM2_MQ_FLAG_SENDSYNC`).

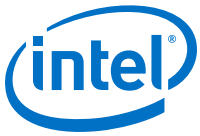
A receive is deemed complete after it has matched its associated receive buffer with an incoming send and that the data from the send has been completely delivered to the receive buffer.

2.4 MQ Progress Requirements

You must explicitly ensure progress on MQs for correctness. The progress requirement holds even if certain areas of the MQ implementation require less network attention than others, or if progress may internally be guaranteed through interrupts. The main polling function, `psm2_poll`, is the most general form of ensuring progress on a given endpoint. Calling `psm2_poll` ensures that progress is made over all the MQs and other components instantiated over the endpoint passed to `psm2_poll`.

While `psm2_poll` is the only way to directly ensure progress, other MQ functions conditionally ensure progress depending on how they are used:

- `psm2_mq_wait` and `psm2_mq_wait2` employ polling and wait until the request is completed. For blocking communication operations where the caller is waiting on a single send or receive to complete, `psm2_mq_wait` or `psm2_mq_wait2` usually provides the best responsiveness in terms of latency.
- `psm2_mq_test` and `psm2_mq_test2` test a particular request for completion, but never directly or indirectly ensure progress because they only test the completion status of a request, nothing more. See functional documentation for `psm2_mq_test` and `psm2_mq_test2` for details.
- `psm2_mq_ipeek` and `psm2_mq_ipeek2` ensure progress if and only if the MQ's completion queue is empty. These functions do not ensure progress as long as the completion queue is non-empty. If you always aggressively process all elements of the MQ completion queue as part of your own progress engine, you indirectly always ensure MQ progress. The `ipeek` or `ipeek2` mechanism is the preferred way for ensuring progress when many non-blocking requests are in flight, since these functions return requests in the order in which they complete. Depending on how communication is initiated and completed, this may be preferable to calling other progress functions on individual requests.
- `psm2_mq_iprobe`, `psm2_mq_iprobe2`, `psm2_mq_improbe`, and `psm2_mq_improbe2` ensure progress if matching request wasn't found after the first attempt.



3.0 Intel® PSM2 Component Functional Documentation

3.1 PSM2 Initialization and Maintenance

3.1.1 Data Structures

`struct psm2_optkey`

Option key/pair structure. Currently only used in MQ.

Data Fields:

uint32_t key	Option key.
void * value	Key value.

3.1.2 Defines

Table 2. Initialization and Maintenance Defines

Define	Description
`#define PSM2_VERNO`	Header-defined Version number.
`#define PSM2_VERNO_MAJOR`	Header-defined Major Version Number.
`#define PSM2_VERNO_MINOR`	Header-defined Minor Version Number.
`#define PSM2_ERRHANDLER_DEFAULT`	Legacy value; included for backwards compatibility. Use `PSM2_ERRHANDLER_PSM_HANDLER` instead.
`#define PSM2_ERRHANDLER_NOP`	Legacy value; included for backwards compatibility. Use `PSM2_ERRHANDLER_NO_HANDLER` instead.
`#define PSM2_ERRHANDLER_PSM_HANDLER`	PSM2 error handler as explained in PSM2 Error Handling.
`#define PSM2_ERRHANDLER_NO_HANDLER`	Bypasses the default PSM2 error handler and returns all errors (this is the default).
`#define PSM2_ERRSTRING_MAXLEN`	Maximum error string length.

3.1.3 Typedefs

Table 3. Initialization and Maintenance Typedefs

Typedef	Description
`typedef enum psm2_error`	See also: `psm2_error`.
`typedef psm2_error_token *psm2_error_token_t`	Error handling opaque token. A token is required for users that register their own handlers and wish to defer further error handling to PSM2.
`typedef psm2_error_t(*psm2_ep_errhandler_t) (psm2_ep_t ep, const psm2_error_t error, const char *error_string, psm2_error_token_t token)`	Error handling function. Users can handle errors explicitly instead of relying on PSM2's own error handler. There is one global error handler and error handlers that can be individually set for each opened endpoint. By default, endpoints inherit the global handler registered at the time of open. **Parameters:** • `ep` Handle associated to the endpoint over which the error occurred or NULL if the error is being handled by the global error handler. • `error` PSM2 error identifier. • `error_string` A descriptive error string of maximum length `PSM2_ERRSTRING_MAXLEN`. • `token` Opaque PSM2 token associated with the particular event that generated the error. The token can be used to extract the error string and can be passed to `psm2_error_defer` to defer any remaining or unhandled error handling to PSM2. **Postcondition:** If the error handler returns, the error returned is propagated to the caller.

3.1.4 Enumerations

```
enum psm2_error {PSM2_OK, PSM2_OK_NO_PROGRESS, PSM2_PARAM_ERR,
PSM2_NO_MEMORY, PSM2_INIT_NOT_INIT, PSM2_INIT_BAD_API_VERSION,
PSM2_NO_AFFINITY, PSM2_INTERNAL_ERR, PSM2_SHMEM_SEGMENT_ERR,
PSM2_OPT_READONLY, PSM2_TIMEOUT, PSM2_TOO_MANY_ENDPOINTS,
PSM2_IS_FINALIZED, PSM2_EP_WAS_CLOSED, PSM2_EP_NO_DEVICE,
PSM2_EP_UNIT_NOT_FOUND, PSM2_EP_DEVICE_FAILURE,
PSM2_EP_NO_PORTS_AVAIL, PSM2_EP_NO_NETWORK,
PSM2_EP_INVALID_UUID_KEY, PSM2_EPID_UNKNOWN,
PSM2_EPID_UNREACHABLE, PSM2_EPID_INVALID_NODE,
PSM2_EPID_INVALID_MTU, PSM2_EPID_INVALID_UUID_KEY,
PSM2_EPID_INVALID_VERSION, PSM2_EPID_INVALID_CONNECT,
PSM2_EPID_ALREADY_CONNECTED,PSM2_EPID_NETWORK_ERROR,
PSM2_MQ_INCOMPLETE, PSM2_MQ_TRUNCATION, PSM2_ERROR_LAST}
```

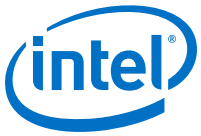


Table 4. Error Type Enumerators

Enumerator	Description
PSM2_OK	Interface-wide "ok", guaranteed to be 0.
PSM2_OK_NO_PROGRESS	No events progressed on psm2_poll (not fatal).
PSM2_PARAM_ERR	Error in a function parameter.
PSM2_NO_MEMORY	PSM2 ran out of memory.
PSM2_INIT_NOT_INIT	PSM2 has not been initialized by psm2_init.
PSM2_INIT_BAD_API_VERSION	API version passed in psm2_init is incompatible.
PSM2_NO_AFFINITY	PSM2 Could not set affinity.
PSM2_INTERNAL_ERR	PSM2 Unresolved internal error.
PSM2_SHMEM_SEGMENT_ERR	PSM2 could not set up shared memory segment.
PSM2_OPT_READONLY	PSM2 option is a read-only option.
PSM2_TIMEOUT	PSM2 operation timed out.
PSM2_TOO_MANY_ENDPOINTS	Too many endpoints.
PSM2_IS_FINALIZED	PSM2 is finalized.
PSM2_EP_WAS_CLOSED	Endpoint was closed.
PSM2_EP_NO_DEVICE	PSM2 Could not find an Intel® Omni-Path Unit.
PSM2_EP_UNIT_NOT_FOUND	User passed a bad unit number.
PSM2_EP_DEVICE_FAILURE	Failure in initializing endpoint.
PSM2_EP_NO_PORTS_AVAIL	No free ports could be obtained.
PSM2_EP_NO_NETWORK	Could not detect network connectivity.
PSM2_EP_INVALID_UUID_KEY	Invalid Unique job-wide UUID Key.
PSM2_EPID_UNKNOWN	Endpoint connect status unknown (because of other failures or if connect attempt timed out).
PSM2_EPID_UNREACHABLE	Endpoint could not be reached by any PSM2 component.
PSM2_EPID_INVALID_NODE	At least one of the connecting nodes was incompatible in endianess.
PSM2_EPID_INVALID_MTU	At least one of the connecting nodes provided an invalid MTU.
PSM2_EPID_INVALID_UUID_KEY	At least one of the connecting nodes provided a bad key.
PSM2_EPID_INVALID_VERSION	At least one of the connecting nodes is running an incompatible PSM2 protocol version.
PSM2_EPID_INVALID_CONNECT	At least one node provided garbled information.
PSM2_EPID_ALREADY_CONNECTED	EPID was already connected.
PSM2_EPID_NETWORK_ERROR	EPID is duplicated, network connectivity problem.

continued...

Enumerator	Description
PSM2_MQ_INCOMPLETE	MQ Non-blocking request is incomplete.
PSM2_MQ_TRUNCATION	MQ Message has been truncated at the receiver.
PSM2_ERROR_LAST	Reserved Value, indicates highest ENUM value for psm2_error.

3.1.5 Functions

Table 5. Initialization and Maintenance Functions

Function	Description
psm2_init (int *api_verno_major, int *api_verno_minor)	Initialize PSM2 interface. For details, see psm2_init.
psm2_finalize (void)	Finalize PSM2 interface. For details, see psm2_finalize.
psm2_error_register_handler (psm2_ep_t ep, const psm2_ep_errhandler_t errhandler)	PSM2 error handler registration. For details, see psm2_error_register_handler.
psm2_error_defer (psm2_error_token_t err_token)	PSM2 deferred error handler. For details, see psm2_error_defer.
psm2_error_get_string (psm2_error_t error)	Get generic error string from error. For details, see psm2_error_get_string.

3.1.5.1 psm2_init

Syntax

```
psm2_error_t psm2_init (int *api_verno_major, int
*api_verno_minor)
```

Call to initialize the PSM2 library for a desired API revision number.

Parameters

`api_verno_major` As input, a pointer to an integer that holds PSM2_VERNO_MAJOR. As output, the pointer is updated with the major revision number of the loaded library.

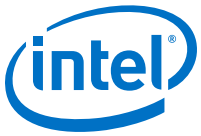
`api_verno_minor` As input, a pointer to an integer that holds PSM2_VERNO_MINOR. As output, the pointer is updated with the minor revision number of the loaded library.

Precondition

You have not called any other PSM2 library call except psm2_error_register_handler to register a global error handler.

Warning

PSM2 initialization is a precondition for all functions used in the PSM2 library.



Returns

PSM2_OK	The PSM2 interface could be opened and the desired API revision can be provided.
PSM2_INIT_BAD_API_VERSION	The PSM2 library is not compatible with the desired API version.

Example

```
// In this example, we want to handle our own errors before doing init,
// since we don't want a fatal error if Intel® Omni-Path is not found.
// Note that @ref psm2_error_register_handler
// (and @ref psm2_uuid_generate)
// are the only functions that can be called before @ref psm2_init

int try_to_initialize_psm() {
    int verno_major = PSM2_VERNO_MAJOR;
    int verno_minor = PSM2_VERNO_MINOR;
    int err = psm2_error_register_handler(NULL, //Global handler
           PSM2_ERRHANDLER_NO_HANDLER);//return errors
    if (err) {
        fprintf(stderr, "Couldn't register global handler: %s\n",
                        psm2_error_get_string(err));
    return -1;
    }

    err = psm2_init(&verno_major, &verno_minor);
    if (err || verno_major > PSM2_VERNO_MAJOR) {
        if (err)
            fprintf(stderr, "PSM2 initialization failure: %s\n",
                                    psm2_error_get_string(err));
            else
                fprintf(stderr, "PSM2 loaded an unexpected/unsupported "
                                "version (%d.%d)\n", verno_major, verno_minor);
            return -1;
    }

    // We were able to initialize PSM2 but defer all further error
    // handling since most of the errors beyond this point are fatal.

    int err = psm2_error_register_handler(NULL, // Global handler
           PSM2_ERRHANDLER_PSM_HANDLER); //
    if (err) {
        fprintf(stderr, "Couldn't register global errhandler: %s\n",
                        psm2_error_get_string(err));
        return -1;
    }
    return 1;
}
```

3.1.5.2 psm2_finalize

Syntax

```
psm2_error_t psm2_finalize (void)
```

Finalize PSM2 interface. Single call to finalize PSM2 and close all unclosed endpoints.

Postcondition

You guarantee not to make any further PSM2 calls, including `psm2_init`.

Returns

`PSM2_OK` Always returns PSM2_OK.

3.1.5.3 psm2_error_register_handler

Syntax

```
psm2_error_t psm2_error_register_handler (psm2_ep_t ep, const
psm2_ep_errhandler_t errhandler)
```

PSM2 error handler registration. Function to register error handlers on a global basis and on a per-endpoint basis. `PSM2_ERRHANDLER_PSM_HANDLER` and `PSM2_ERRHANDLER_NO_HANDLER` are special pre-defined handlers to respectively enable use of the default PSM2-internal handler or the no-handler that disables registered error handling and returns all errors to the caller (both are documented in PSM2 Error Handling on page 14).

Parameters

`ep` Handle of the endpoint over which the error handler should be registered. With `ep` set to NULL, the behavior of the global error handler can be controlled.

`errhandler` Handler to register. Can be a user-specific error handling function or `PSM2_ERRHANDLER_PSM_HANDLER` or `PSM2_ERRHANDLER_NO_HANDLER`.

Remarks

When `ep` is set to NULL, this is the only function that can be called before `psm2_init`.

3.1.5.4 psm2_error_defer

Syntax

```
psm2_error_t psm2_error_defer (psm2_error_token_t err_token)
```

PSM2 deferred error handler.

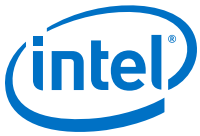
Function to handle fatal PSM2 errors if no error handler is installed or if you wish to defer further error handling to PSM2. Depending on the type of error, PSM2 may or may not return from the function call.

Parameters

`err_token` Error token initially passed to error handler.

Precondition

The function is called because PSM2 is designated to handle an error case.



Postcondition

The function may or may not return depending on the error.

3.1.5.5 psm2_error_get_string

Syntax

```
const char* psm2_error_get_string (psm2_error_t error)
```

Get generic error string from error. Function to return the default error string associated to a PSM2 error. While a more detailed and precise error string is usually available within error handlers, this function is available to obtain an error string out of an error handler context or when a no-op error handler is registered.

Parameters

`error` PSM2 error.

3.2 PSM2 Device Endpoint Management

3.2.1 Data Structures

3.2.1.1 psm2_ep_open_opts

Endpoint Open Options. These options are available for opening a PSM2 endpoint. Each is individually documented. Setting each option to -1 or passing NULL as the options parameter in `psm2_ep_open` instructs PSM2 to use implementation-defined defaults.

Additional details are documented in the `psm2_ep_open` Options section.

Data Fields:

Field	Description
`int64_t timeout`	Timeout in nanoseconds to open device.
`int unit`	Intel® Omni-Path Unit ID to open on. **Note:** The Intel® PSM2 implementation has a limit of four (4) HFIs.
`int affinity`	How PSM2 should set affinity.
`int shm_mbytes`	Megabytes used for intra-node communication.
`int sendbufs_num`	Preallocated send buffers.
`uint64_t network_pkey`	Network Protection Key (v1.01).
`int port`	Intel® Omni-Path port to use. Range = 1 to N.
`int outsl`	Intel® Omni-Path SL to use when sending packets.
`uint64_t service_id`	Intel® Omni-Path Service ID to use for endpoint.

continued...

Field	Description
psm2_path_res_t path_res_type	Path resolution type.
int senddesc_num	Preallocated send descriptors.
int imm_size	Immediate data size for endpoint.

3.2.2 Defines

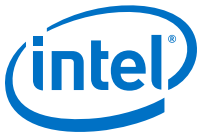
Table 6. Endpoint Defines

Define	Description
#define PSM2_EP_OPEN_AFFINITY_SKIP	Disable setting affinity.
#define PSM2_EP_OPEN_AFFINITY_SET	Enable setting affinity unless already set.
#define PSM2_EP_OPEN_AFFINITY_FORCE	Enable setting affinity regardless of current affinity setting.
#define PSM2_EP_OPEN_PKEY_DEFAULT	Default protection key.
#define PSM2_EP_CLOSE_GRACEFUL	Graceful close mode in psm2_ep_close.
#define PSM2_EP_CLOSE_FORCE	Forceful close mode in psm2_ep_close.

3.2.3 Typedefs

Table 7. Endpoint Typedefs

Typedef	Description
typedef psm2_ep *psm2_ep_t	Local endpoint handle (opaque). Handle is returned when a new local endpoint is created. The handle is a local handle to be used in all communication functions and is not intended to globally identify the opened endpoint in any way. All open endpoint handles can be globally identified using the endpoint id integral type (psm2_epid_t) and all communication must use an endpoint address (psm2_epaddr_t) that can be obtained by connecting a local endpoint to one or more endpoint identifiers.
typedef uint64_t psm2_epid_t	Endpoint ID. Integral type of size 8 bytes that can be used to globally identify a successfully opened endpoint. Although the contents of the endpoint id integral type remains opaque, unique network ID and Intel® Omni-Path port number can be extracted using psm2_epid_nid and psm2_epid_port.
typedef psm2_epaddr *psm2_epaddr_t	Endpoint Address (opaque). Remote endpoint addresses are created when you bind an endpoint ID to a particular endpoint handle using psm2_ep_connect. A given endpoint address is only guaranteed to be valid over a single endpoint.
typedef uint8_t psm2_uuid_t[16]	PSM2 Unique UID (UUID). PSM2 type equivalent to the DCE-1 uuid_t, used to uniquely identify an endpoint within a particular job. Since PSM2 does not participate in job allocation and management, you must generate a unique ID to associate endpoints to a particular parallel or collective job. See also: psm2_uuid_generate.



3.2.4 Functions

Table 8. Endpoint Functions

Function	Description
`psm2_epid_nid (psm2_epid_t epid)`	Get Endpoint identifier's Unique Network ID.
`psm2_epid_port (psm2_epid_t epid)`	Get Endpoint identifier's Intel® Omni-Path port.
`psm2_epid_context (psm2_epid_t epid)`	Get Endpoint identifier's Intel® Omni-Path context number.
`psm2_map_nid_hostname(int num, const uint64_t *nids, const char **hostnames)`	Provide a mapping from network ID (LID) to hostnames. For details, see psm2_map_nid_hostname.
`psm2_ep_num_devunits (uint32_t *num_units)`	List the number of available Intel® Omni-Path units. For details, see psm2_ep_num_devunits.
`psm2_uuid_generate (psm2_uuid_t uuid_out)`	Utility to generate UUIDs for `psm2_ep_open`. For details, see psm2_uuid_generate.
`psm2_ep_open_opts_get_defaults (struct psm2_ep_open_opts *opts);`	Endpoint open default options. For details, see psm2_ep_open_opts_get_defaults.
`psm2_ep_open (const psm2_uuid_t unique_job_key, const struct psm2_ep_open_opts *opts, psm2_ep_t *ep, psm2_epid_t *epid)`	Intel® Omni-Path endpoint creation. For details, see psm2_ep_open.
`psm2_ep_epid_share_memory (psm2_ep_t ep, psm2_epid_t epid, int *result)`	Endpoint shared memory query. For details, see psm2_ep_epid_share_memory.
`psm2_ep_close (psm2_ep_t ep, int mode, int64_t timeout)`	Close endpoint. For details, see psm2_ep_close.
`psm2_ep_connect (psm2_ep_t ep, int num_of_epid, const psm2_epid_t *array_of_epid, const int *array_of_epid_mask, psm2_error_t *array_of_errors, psm2_epaddr_t *array_of_epaddr, int64_t timeout)`	Connect one or more remote endpoints to a local endpoint. For details, see psm2_ep_connect.
`psm2_ep_disconnect (psm2_ep_t ep, int num_of_epaddr, const psm2_epaddr_t *array_of_epaddr, const int *array_of_epaddr_mask, psm2_error_t *array_of_errors, int64_t timeout)`	Disconnect one or more remote endpoints from a local endpoint. For details, see psm2_ep_disconnect.
`psm2_poll (psm2_ep_t ep)`	Ensure endpoint communication progress. For details, see psm2_poll.
`psm2_epaddr_setlabel (psm2_epaddr_t epaddr, const char *epaddr_label_string)`	Set a user-determined `ep` address label. For details, see psm2_epaddr_setlabel.
`psm2_ep_query (int *num_of_epinfo, psm2_epinfo_t *array_of_epinfo)`	Query PSM2 for endpoint information. For details, see psm2_ep_query.
`psm2_ep_epid_lookup (psm2_epid_t epid, psm2_epconn_t *epconn)`	Query PSM2 for endpoint connections. For details, see psm2_ep_epid_lookup.
`psm2_ep_epid_lookup2 (psm2_ep_t ep, psm2_epid_t epid, psm2_epconn_t *epconn)`	Query specified PSM2 endpoint for its connections. For details, see psm2_ep_epid_lookup2.
`psm2_epaddr_to_epid (psm2_epaddr_to_epid)`	Get PSM2 epid for given epaddr. For details, see psm2_epaddr_to_epid.

3.2.4.1 psm2_map_nid_hostname

Syntax

```
psm2_error_t psm2_map_nid_hostname(int num, const uint64_t *nids, const char
**hostnames)
```

Provide a mapping from Network ID (LID) to hostnames.

Since PSM2 does not assume or rely on the availability of an external network ID-to-hostname mapping service, users can provide one or more of these mappings. The `psm2_map_nid_hostname` function allows a list of network ids to be associated with hostnames.

This function is not mandatory for correct operation but may allow PSM2 to provide better diagnostics when remote endpoints are unavailable and can otherwise only be identified by their Network ID.

Parameters

`num`	Number elements in `nid` and `hostnames` arrays.
`nids`	User-provided array of network IDs (that is, Intel® Omni-Path LIDs), should be obtained by calling `psm2_epid_nid` on each epid.
`hostnames`	User-provided array of hostnames (array of NULL-terminated strings) where each hostname index maps to the provided nid hostname.

Warning

Duplicate nids may be provided in the input `nids` array, only the first corresponding hostname is remembered.

Precondition

You may or may not have already provided a hostname mappings.

Postcondition

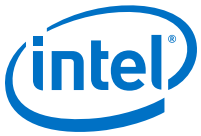
You may free any dynamically allocated memory passed to the function.

3.2.4.2 psm2_ep_num_devunits

Syntax

```
psm2_error_t psm2_ep_num_devunits (uint32_t *num_units)
```

List the number of available Intel® Omni-Path units. Function used to determine the amount of locally available Intel® Omni-Path units. For N units, valid unit numbers in `psm2_ep_open` are 0 to N-1.



Returns

`PSM2_OK` Unless you have not called `psm2_init`.

3.2.4.3 psm2_uuid_generate

Syntax

```
void psm2_uuid_generate (psm2_uuid_t uuid_out)
```

Utility to generate UUIDs for `psm2_ep_open`. Utility to generate UUIDs for `psm2_ep_open`. This function is available as a utility for generating unique job-wide ids. See discussion in `psm2_ep_open` for further information.

Remarks

This function does not require PSM2 to be initialized.

3.2.4.4 psm2_ep_open_opts_get_defaults

Syntax

```
psm2_error_t psm2_ep_open_opts_get_defaults (struct psm2_ep_open_opts *opts);
```

Function used to initialize the set of endpoint options to their default values for use in `psm2_ep_open`.

Parameters

`opts` Endpoint Open options.

Warning

For portable operation, you should always call this function prior to calling `psm2_ep_open`.

Returns

`PSM2_OK`	If result could be updated.
`PSM2_INIT_NOT_INIT`	If PSM2 has not been initialized.
`PSM2_PARAM_ERR`	If user passes invalid parameters to the API.

3.2.4.5 psm2_ep_open

Syntax

```
psm2_error_t psm2_ep_open (const psm2_uuid_t unique_job_key, const
struct psm2_ep_open_opts *opts, psm2_ep_t *ep, psm2_epid_t *epid)
```

Endpoint creation.

Function used to create a new local communication endpoint on an Intel® Omni-Path HFI. The returned endpoint handle is required in all PSM2 communication operations, as PSM2 can manage communication over multiple endpoints. An opened endpoint has no global context until you connect the endpoint to other global endpoints by way of `psm2_ep_connect`. All local endpoint handles are globally identified by endpoint IDs (`psm2_epid_t`) which are also returned when an endpoint is opened. It is assumed that you can provide an out-of-band mechanism to distribute the endpoint IDs in order to establish connections between endpoints (see `psm2_ep_connect` for more information).

Parameters

`unique_job_key`	Endpoint key, to uniquely identify the endpoint's job. You must ensure that the key is globally unique over a period long enough to prevent duplicate keys over the same set of endpoints (see additional details in the following paragraphs).
`opts`	Open options of type `psm2_ep_open_opts` (see `psm2_ep_open_opts_get_defaults`). Note that this parameter can also be NULL. Refer to the example in `psm2_init`.
`ep`	User-supplied storage to return a pointer to the newly created endpoint. The returned pointer of type `psm2_ep_t` is a local handle and cannot be used to globally identify the endpoint.
`epid`	User-supplied storage to return the endpoint ID associated to the newly created local endpoint returned in the `ep` handle. The endpoint ID is an integral type suitable for uniquely identifying the local endpoint.

PSM2 does not internally verify the consistency of the uuid. You must ensure that the uuid is unique enough not to collide with other currently-running jobs. Use one of the following mechanisms to obtain a uuid:

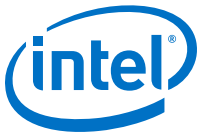
1. Use the supplied `psm2_uuid_generate` utility.
2. Use an OS or library-specific uuid generation utility that complies with OSF DCE 1.1, such as `uuid_generate` on Linux* or `uuid_create` on FreeBSD*.

   See: http://www.opengroup.org/onlinepubs/009629399/uuid_create.htm.
3. Manually pack a 16-byte string using a utility such as /dev/random or other source with enough entropy and proper seeding to prevent two nodes from generating the same `uuid_t`.

Options

The following options are relevant when opening an endpoint:

- `timeout` establishes the amount of nanoseconds to wait before failing to open a port (with -1, defaults to 30 secs).
- `unit` sets the unit number to use to open a port (with -1, PSM2 determines the best unit to open the port). If `HFI_UNIT` is set in the environment, this setting is ignored.



- `affinity` enables or disables PSM2 setting processor affinity. The option can be controlled to either disable (`PSM2_EP_OPEN_AFFINITY_SKIP`) or enable the affinity setting only if it is already unset (`PSM2_EP_OPEN_AFFINITY_SET`) or regardless of affinity begin set or not (`PSM2_EP_OPEN_AFFINITY_FORCE`). If `HFI_NO_CPUAFFINITY` is set in the environment, this setting is ignored.
- `shm_mbytes` sets a maximum amount of megabytes that can be allocated to each local endpoint ID connected through this endpoint (with -1, defaults to 10 MB).
- `sendbufs_num` sets the number of send buffers that can be pre-allocated for communication (with -1, defaults to 512 buffers of MTU size).
- `network_pkey` sets the protection key to employ for point-to-point PSM2 communication. Unless a specific value is used, this parameter should be set to `PSM2_EP_OPEN_PKEY_DEFAULT`.
- `port` sets the Intel® Omni-Path port to use. Range = 1 to N.
- `outsl` sets the Intel® Omni-Path SL to use when sending packets. Range = 0 to 31. Check with your network administrator for details.
- `service_id` sets the Intel® Omni-Path Service ID to use for an endpoint. Used for path resolution. Default is 0x1000117500000000ULL

  See `PSM2_IB_SERVICE_ID` for more details.
- `path_res_type` sets the path resolution type. Values include:
  - `PSM2_PATH_RES_NONE` (default)
  - `PSM2_PATH_RES_OPP`
  - `PSM2_PATH_RES_UMAD`

  See `PSM2_PATH_REC` for more details.
- `senddesc_num` sets preallocated send descriptors. Default = 1048576 (1 Million).

  See `PSM2_MQ_RNDV_HFI_THRESH` for more details.
- `imm_size` sets the immediate data send size not requiring a buffer. Default = 128 bytes.

Postcondition

Depending on the environment variable `PSM2_MULTI_EP` being set and its contents, support for opening multiple endpoints is either enabled or disabled.

Warning

By default, PSM2 limits the user to calling `psm2_ep_open` only once per process and subsequent calls will fail. To enable creation of multiple endpoints per process, you must properly set the environment variable `PSM2_MULTI_EP` before calling `psm2_init`.

Returns

`PSM2_PARAM_ERR`	If user passes invalid parameters to the API.

Example

```
// In order to open an endpoint and participate in a job, each endpoint has
// to be distributed a unique 16-byte UUID key from an out-of-band source.
// Presumably this can come from the parallel spawning utility either
// indirectly through an implementors own spawning interface or as in this
// example, the UUID is set as a string in an environment variable
// propagated to all endpoints in the job.

int try_to_open_psm2_endpoint(psm2_ep_t *ep, // output endpoint handle
    psm2_epid_t *epid, // output endpoint identifier
    int unit) // unit of our choice
{
    psm2_ep_open_opts epopts;
    psm2_uuid_t job_uuid;
    char *c;

    // Let PSM2 assign its default values to the endpoint options.
    psm2_ep_open_opts_get_defaults(&epopts);

    // We want a stricter timeout and a specific unit
    epopts.timeout = 15*1e9; // 15 second timeout
    epopts.unit = unit; // We want a specific unit, -1 would let PSM2
                        // choose the unit for us.
    // We've already set affinity, don't let PSM2 do so if it wants to.
    if (epopts.affinity == PSM2_EP_OPEN_AFFINITY_SET)
        epopts.affinity = PSM2_EP_OPEN_AFFINITY_SKIP;

    // ENDPOINT_UUID is set to the same value in the environment of all the
    // processes that wish to communicate over PSM2 and was generated by
    // the process spawning utility.
    c = getenv("ENDPOINT_UUID");
    if (c && *c)
        implementor_string_to_16byte_packing(c, job_uuid);
    else {
        fprintf(stderr, "Can't find UUID for endpoint\n);
        return -1;
    }

    // Assume we don't want to handle errors here.
    psm2_ep_open(job_uuid, &epopts, ep, epid);
    return 1;
}
```

3.2.4.6 psm2_ep_epid_share_memory

Syntax

```
psm2_error_t psm2_ep_epid_share_memory (psm2_ep_t ep, psm2_epid_t
epid, int *result)
```

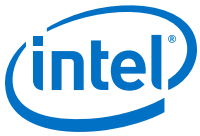
Endpoint shared memory query. Function used to determine if a remote endpoint shares memory with a currently opened local endpoint.

Parameters

`ep` Endpoint handle.

`epid` Endpoint ID.

`result` Is non-zero if the remote endpoint shares memory with the local endpoint `ep`, or zero otherwise.



Returns

PSM2_OK	If result could be updated.
PSM2_EPID_UNKNOWN	If the epid is not recognized.

3.2.4.7 psm2_ep_close

Syntax

```
psm2_error_t psm2_ep_close (psm2_ep_t ep, int mode, int64_t timeout)
```

Close endpoint.

Parameters

ep	Endpoint handle.
mode	One of PSM2_EP_CLOSE_GRACEFUL or PSM2_EP_CLOSE_FORCE. If mode is PSM2_EP_CLOSE_GRACEFUL, before closing the endpoint, the function attempts to disconnect from any other endpoints that are connected, and also waits for connected endpoints to disconnect. If the timeout is reached and there are still unresolved open connections, the endpoint is closed as if mode was set to PSM2_EP_CLOSE_FORCE. If mode is PSM2_EP_CLOSE_FORCE, the endpoint is closed without ensuring that any open connections are successfully disconnected.
timeout	How long to wait in nanoseconds for negotiated disconnects to succeed. If mode is PSM2_EP_CLOSE_GRACEFUL, 0 waits forever. -1 lets the function decide using an internal heuristic. If mode is PSM2_EP_CLOSE_FORCE, this parameter is ignored.

The following error is returned, others are handled by the per-endpoint error handler:

Returns

PSM2_OK	Endpoint was successfully closed without force or successfully closed with force within the supplied timeout.

3.2.4.8 psm2_ep_connect

Syntax

```
psm2_error_t psm2_ep_connect (psm2_ep_t ep, int num_of_epid, const
psm2_epid_t *array_of_epid, const int *array_of_epid_mask,
psm2_error_t *array_of_errors, psm2_epaddr_t *array_of_epaddr,
int64_t timeout)
```

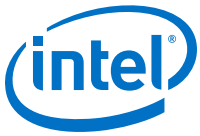
Connect one or more remote endpoints to a local endpoint. Function to non-collectively establish a connection to a set of endpoint IDs and translate endpoint IDs into endpoint addresses. Establishing a remote connection with a set of remote endpoint IDs does not imply a collective operation and you are free to connect unequal sets on each process. Similarly, a given endpoint address does not imply that a pairwise communication context exists between the local endpoint and remote endpoint.

Parameters

`ep`	Endpoint handle.
`num_of_epid`	The number of endpoints to connect to, which also establishes the amount of elements contained in all of the function's array-based parameters.
`array_of_epid`	User-allocated array that contains `num_of_epid` valid endpoint identifiers. Each endpoint id (or `epid`) has been obtained through an out-of-band mechanism and each endpoint must have been opened with the same uuid key.
`array_of_epid_mask`	User-allocated array that contains `num_of_epid` integers. This array of masks allows users to select which of the epids in `array_of_epid` should be connected. If the integer at index i is zero, PSM2 does not attempt to connect to the epid at index i in `array_of_epid`. If this parameter is NULL, PSM2 tries to connect to each epid.
`array_of_errors`	User-allocated array of at least `num_of_epid` elements. If the function does not return `PSM2_OK`, this array can be consulted for each endpoint not masked off by `array_of_epid_mask` to know why the endpoint could not be connected. Endpoints that could not be connected because of an unrelated failure are marked as `PSM2_EPID_UNKNOWN`. If the function returns `PSM2_OK`, the errors for all endpoints also contain `PSM2_OK`.
`array_of_epaddr`	User-allocated array of at least `num_of_epid` elements of type `psm2_epaddr_t`. Each successfully connected endpoint is updated with an endpoint address handle that corresponds to the endpoint id at the same index in `array_of_epid`. Handles are only updated if the endpoint could be connected and if its error in `array_of_errors` is `PSM2_OK`.
`timeout`	Timeout in nanoseconds after which connection attempts are abandoned. Setting this value to 0 disables timeout and waits until all endpoints have been successfully connected or until an error is detected.

Precondition

You have opened a local endpoint and obtained a list of endpoint IDs to connect to a given endpoint handle using an out-of-band mechanism not provided by PSM2.



Postcondition

If the connect is successful, `array_of_epaddr` is updated with valid endpoint addresses.

If unsuccessful, you can query the return status of each individual remote endpoint in `array_of_errors`.

You can call into `psm2_ep_connect` many times with the same endpoint ID and the function is guaranteed to return the same output parameters. PSM2 does not keep any reference to the arrays passed into the function and the caller is free to deallocate them.

The error value with the highest importance is returned by the function if some portion of the communication failed. Users should always refer to individual errors in `array_of_errors` whenever the function cannot return `PSM2_OK`.

Returns

`PSM2_OK` The entire set of endpoint IDs were successfully connected and endpoint addresses are available for all endpoint IDs.

Example

```
int connect_endpoints(psm2_ep_t ep, int numep, const psm2_epid_t
                                  *array_of_epid, psm2_epaddr_t
**array_of_epaddr_out)
{
    psm2_error_t *errors = (psm2_error_t *)
                           calloc(numep, sizeof(psm2_error_t));
    if (errors == NULL)

        return -1;

    psm2_epaddr_t *all_epaddrs =
            (psm2_epaddr_t *) calloc(numep, sizeof(psm2_epaddr_t));
    if (all_epaddrs == NULL)
       return -1;
    psm2_ep_connect(ep, numep, array_of_epid,
            NULL, // We want to connect all epids, no mask needed
            errors,
            all_epaddrs,
            30*e9); // 30 second timeout, <1 ns is forever
    *array_of_epaddr_out = all_epaddrs; free(errors);
    return 1;
}
```

3.2.4.9 psm2_ep_disconnect

Syntax

```
psm2_error_t psm2_ep_disconnect (psm2_ep_t ep, int num_of_epaddr,
psm2_epaddr_t *array_of_epaddr, const int *array_of_epaddr_mask,
psm2_error_t *array_of_errors, int64_t timeout)
```

Disconnect one or more remote endpoints from a local endpoint. Function to non-collectively disconnect a connection to a set of endpoint addresses and free each of the endpoint addresses if there are no incoming connections to that endpoint address.

After disconnecting, the application cannot send messages to the remote processes again and PSM2 is restored back to the state before calling psm2_ep_connect. The application must call psm2_ep_connect to establish the connections again.

Parameters

ep	Endpoint handle.
num_of_epaddr	The number of endpoint addresses to disconnect from, which also indicates the amount of elements contained in all of the function's array-based parameters.
array_of_epaddr	User-allocated array that contains num_of_epaddr valid endpoint addresses. Each endpoint address (or epaddr) has been obtained through a previous psm2_ep_connect call.
array_of_epaddr_mask	User-allocated array that contains num_of_epaddr integers. This array of masks allows users to select which of the epaddresses in array_of_epaddr should be disconnected. If the integer at index i is zero, PSM2 does not attempt to disconnect to the epaddr at index i in array_of_epaddr. If this parameter is NULL, PSM2 tries to disconnect all epaddr in array_of_epaddr.
array_of_errors	User-allocated array of at least num_of_epaddr elements. If the function does not return PSM2_OK, this array can be consulted for each endpoint address not masked off by array_of_epaddr_mask to know why the endpoint could not be disconnected. Any endpoint address that could not be disconnected because of an unrelated failure is marked as PSM2_EPID_UNKNOWN. If the function returns PSM2_OK, the errors for all endpoint addresses also contain PSM2_OK.
timeout	Timeout in nanoseconds after which disconnection attempts are abandoned. Setting this value to 0 disables timeout and waits until all endpoints have been successfully disconnected or until an error is detected.

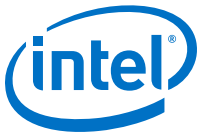
Precondition

You have established the connections with previous psm2_ep_connect calls.

Postcondition

If the disconnect is successful, the corresponding epaddr in array_of_epaddr is reset to NULL pointer.

If unsuccessful, you can query the return status of each individual remote endpoint in array_of_errors.



PSM2 does not keep any reference to the arrays passed into the function and the caller is free to deallocate them.

The error value with the highest importance is returned by the function if some portion of the communication failed. Refer to individual errors in `array_of_errors` whenever the function cannot return `PSM2_OK`.

Returns

`PSM2_OK` The entire set of endpoint IDs were successfully disconnected and endpoint addresses are freed by PSM2.

Example

```
int disconnect_endpoints(psm2_ep_t ep, int num_epaddr, const psm2_epaddr_t
            *array_of_epaddr)
{
    psm2_error_t *errors = (psm2_error_t *)
            calloc(num_epaddr, sizeof(psm2_error_t));
    if (errors == NULL)
        return -1;

    psm2_ep_disconnect(ep, num_epaddr, array_of_epaddr,
            NULL, // We want to disconnect all epaddrs, no mask needed,
            errors,
            30*e9); // 30 second timeout, <1 ns is forever

    free(errors);
    return 1;
}
```

3.2.4.10 psm2_poll

Syntax

```
psm2_error_t psm2_poll (psm2_ep_t ep)
```

Ensure endpoint communication progress.

Function to ensure progress for all PSM2 components instantiated on an endpoint (currently, this only includes the MQ component). The function never blocks and is typically required in two cases:

- Allowing all PSM2 components instantiated over a given endpoint to make communication progress. Refer to MQ Progress Requirements on page 26 for a detailed discussion on MQ-level progress issues.
- Cases where users write their own synchronization primitives that depend on remote communication, such as spinning on a memory location whose new value depends on ongoing communication.

The poll function does not block, but you can rely on the `PSM2_OK_NO_PROGRESS` return value to control polling behavior in terms of frequency (poll until an event happens) or execution environment (poll for a while but yield to other threads of CPUs are oversubscribed).

Returns

PSM2_OK	Some communication events were progressed.
PSM2_OK_NO_PROGRESS	Polling did not yield any communication progress.

3.2.4.11 psm2_epaddr_setlabel

Syntax

```
void psm2_epaddr_setlabel (psm2_epaddr_t epaddr, const char *epaddr_label_string)
```

Set a user-determined ep address label.

Parameters

epaddr	Endpoint address, obtained from psm2_ep_connect.
epaddr_label_string	User-allocated string to print when identifying endpoint in error handling or other verbose printing. You must allocate the NULL-terminated string since PSM2 only keeps a pointer to the label. If you do not explicitly set a label for each endpoint, endpoints identify themselves as hostname:port.

3.2.4.12 psm2_ep_query

Syntax

```
psm2_error_t psm2_ep_query(int *num_of_epinfo, psm2_epinfo_t *array_of_epinfo)
```

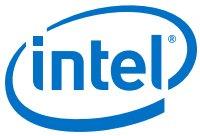
Function to query PSM2 for endpoint information. This allows retrieval of endpoint information in cases where the caller does not have access to the results of psm2_ep_open. In the default single-rail mode, PSM2 uses a single endpoint. If either multi-rail mode or multi-endpoint mode is enabled, PSM2 uses multiple endpoints.

Parameters

num_of_epinfo	On input, sizes the available number of entries in array_of_epinfo. On output, specifies the returned number of entries in array_of_epinfo.
array_of_epinfo	Returns endpoint information structures.

Precondition

PSM2 is initialized and the endpoint has been opened.



Returns

PSM2_OK	Indicates success.
PSM2_PARAM_ERR	If input num_if_epinfo is less than or equal to zero.
PSM2_EP_WAS_CLOSED	If PSM2 endpoint is closed or does not exist.

3.2.4.13 psm2_ep_epid_lookup

Syntax

```
psm2_error_t psm2_ep_epid_lookup(psm2_epid_t epid, psm2_epconn_t *epconn)
```

Function to query PSM2 for endpoint connections. This allows retrieval of endpoint connections in cases where the caller does not have access to the results of psm2_ep_connect. The epid values can be found using psm2_ep_query so that each PSM2 process can determine its own epid. These values can then be distributed across the PSM2 process so that each PSM process knows the epid for all other PSM2 processes.

Parameters

epid	Endpoint ID of a PSM2 process.
epconn	Returns connection information for the specified PSM2 process.

Precondition

PSM2 is initialized and the endpoint has been connected to this epid.

Returns

PSM2_OK	Indicates success.
PSM2_EP_WAS_CLOSED	If PSM2 endpoint is closed or does not exist.
PSM2_EPID_UNKNOWN	If the epid is not recognized.

3.2.4.14 psm2_ep_epid_lookup2

Syntax

```
psm2_error_t psm2_ep_epid_lookup2(psm2_ep_t ep, psm2_epid_t epid, psm2_epconn_t
*epconn)
```

Function to query PSM2 endpoint for its connections.

Note: This function is similar to psm2_ep_epid_lookup, however, it contains an extra endpoint parameter which limits the lookup to that single ep.

Parameters

ep	PSM2 endpoint handle.
epid	Endpoint ID of a PSM2 process.
epconn	Returns connection information for the specified PSM2 process.

Returns

PSM2_OK	Indicates success.
PSM2_EP_WAS_CLOSED	If PSM2 endpoint is closed or does not exist.
PSM2_EPID_UNKNOWN	If the epid is not recognized.
PSM2_PARAM_ERR	If output epconn is NULL.

3.2.4.15 psm2_epaddr_to_epid

Syntax

```
psm2_error_t psm2_epaddr_to_epid(psm2_epaddr_t epaddr, psm2_epid_t *epid)
```

Get PSM2 epid for given epaddr.

Parameters

epaddr	Endpoint address.
epid	Returns endpoint ID of a PSM2 process.

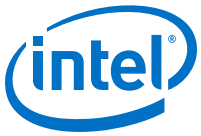
Returns

PSM2_OK	Indicates success.
PSM2_PARAM_ERR	If input epaddr or output epconn is NULL.

3.3 PSM2 Matched Queues

3.3.1 Modules

PSM2 Matched Queue Options.



3.3.2 Data Structures

Table 9. Matched Queues Data Structures

Data Structure	Description
psm2_mq_status	MQ Non-blocking operation status structure. For details, see psm2_mq_status on page 49.
psm2_mq_stats	MQ statistics structure. For details, see MQ Statistics Structure on page 49.
psm2_tag_t	MQ 96-bit tag structure For details, see psm2_tag_t on page 50.
psm2_mq_status2_t	MQ status structure for 96-bit (psm2_tag_t) non-blocking operations. For details, see psm2_mq_status2 on page 50.

3.3.2.1 psm2_mq_status

```
struct psm2_mq_status
```

MQ Non-blocking operation status structure

Message completion status for asynchronous communication operations. For wait and test functions, MQ fills in the structure upon completion. Upon completion, receive requests fill in every field of the status structure while send requests only return a valid `error_code` and context pointer.

Data Fields:

Field	Description
uint64_t msg_tag	Sender's original message tag (receive reqs only).
uint32_t msg_length	Sender's original message length (receive reqs only).
uint32_t nbytes	Actual number of bytes transferred (receive reqs only).
psm2_error_t error_code	MQ error code for communication operation.
void *context	User-associated context for send or receive.

3.3.2.2 MQ Statistics Structure

```
struct psm2_mq_stats
```

MQ statistics structure

Data Fields:

Field	Description
uint64_t rx_user_bytes	Bytes received into a matched user buffer.
uint64_t rx_user_num	Messages received into a matched user buffer.

continued...

Field	Description
uint64_t rx_sys_bytes	Bytes received into an unmatched system buffer.
uint64_t rx_sys_num	Messages received into an unmatched system buffer.
uint64_t tx_num	Total Messages transmitted (shm and hfi).
uint64_t tx_eager_num	Messages transmitted eagerly.
uint64_t tx_eager_bytes	Bytes transmitted eagerly.
uint64_t tx_rndv_num	Messages transmitted using expected TID mechanism.
uint64_t tx_rndv_bytes	Bytes transmitted using expected TID mechanism.
uint64_t tx_shm_num	Messages transmitted (shm only).
uint64_t rx_shm_num	Messages received through shm.
uint64_t rx_sysbuf_num	Number of system buffers allocated.
uint64_t rx_sysbuf_bytes	Bytes allocated for system buffers
uint64_t _reserved[16]	Internally reserved for future use.

3.3.2.3 psm2_tag_t

```
struct psm2_tag_t
```

MQ 96-bit tag structure

Data Fields:

Field	Description
uint32_t tag[3]	Message tag bits. The backwards-compatible 64-bit component of the tag is stored in tag[0] and tag[1].

3.3.2.4 psm2_mq_status2

```
struct psm2_mq_status2
```

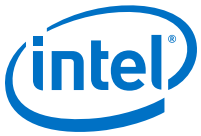
MQ Non-blocking operation status structure

Message completion status for asynchronous communication operations. For wait and test functions, MQ fills in the structure upon completion. Upon completion, receive requests fill in every field of the status structure while send requests only return a valid `error_code` and context pointer.

Data Fields:

Field	Description
psm2_epaddr_t msg_peer	Remote peer's epaddr.
psm2_mq_tag_t msg_tag	Sender's original message tag.
uint32_t msg_length	Sender's original message length (receive reqs only).

continued...



Field	Description
uint32_t nbytes	Actual number of bytes transferred (receive reqs only).
psm2_error_t error_code	MQ error code for communication operation.
void * context	User-associated context for send or receive.

3.3.3 Defines

Table 10. Matched Queues Defines

#define PSM2_MQ_ORDERMASK_NONE	Used to initialize MQ and disable all MQ message ordering guarantees (this mask may prevent the use of MQ to maintain matched message envelope delivery required in MPI).
#define PSM2_MQ_ORDERMASK_ALL	Used to initialize MQ with no message ordering hints, which forces MQ to maintain order over all messages.
#define PSM2_MQ_FLAG_SENDSYNC	MQ Send Force synchronous send.
#define PSM2_MQ_REQINVALID	MQ request completion value.
#define PSM2_MQ_NUM_STATS	How many stats are currently used in psm2_mq_stats.
#define PSM2_MQ_ANY_ADDR	psm2_epaddr_t that matches any epaddr in the MQ.

3.3.4 Typedefs

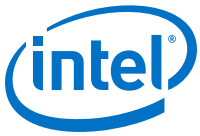
Typedef	Description
typedef psm2_mq *psm2_mq_t	MQ handle (opaque). Handle returned when a new Matched Queue is created (psm2_mq_init).
typedef struct psm2_mq_status psm2_mq_status_t	MQ Non-blocking operation status for 64-bit tagged operations. Message completion status for asynchronous communication operations. For wait and test functions, MQ fills in the structure upon completion. Other than error_code and context guaranteed to be valid for send and recv operations, other struct members are only defined for posted receives.
typedef struct psm2_mq_status2 psm2_mq_status_t	MQ Non-blocking operation status for 96-bit tagged operations. Message completion status for asynchronous communication operations. For wait and test functions, MQ fills in the structure upon completion. Other than error_code and context guaranteed to be valid for send and recv operations, other struct members are only defined for posted receives.
typedef struct psm2_mq_stats psm2_mq_stats_t	Statistics for messages send and received over a given MQ.
typedef psm2_mq_req *psm2_mq_req_t	PSM2 Communication handle (opaque).

3.3.5 Functions

Table 11. Matched Queue Functions

Function	Description
`psm2_mq_init (psm2_ep_t ep, uint64_t tag_order_mask, const struct psm2_optkey *opts, int numopts, psm2_mq_t *mq)`	Initialize the MQ component for MQ communication. For details, see psm2_mq_init.
`psm2_mq_finalize (psm2_mq_t mq)`	Finalize (close) an MQ handle. For details, see psm2_mq_finalize.
`psm2_mq_irecv (psm2_mq_t mq, uint64_t rtag, uint64_t rtagsel, uint32_t flags, void *buf, uint32_t len, void *context, psm2_mq_req_t *req)`	Post a receive to a Matched Queue with tag selection criteria. For details, see psm2_mq_irecv.
`psm2_mq_irecv2 (psm2_mq_t mq, psm2_epaddr_t src, psm2_mq_tag_t *rtag, psm2_mq_tag_t *rtagsel, uint32_t flags, void *buf, uint32_t len, void *context, psm2_mq_req_t *req)`	Post a receive to a Matched Queue with tag selection criteria, it only matches message from the specified src process. Source matching is optional. Uses 96-bit psm2_mq_tag_t instead of 64-bit tag. For details, see psm2_mq_irecv2.
`psm2_mq_send (psm2_mq_t mq, psm2_epaddr_t dest, uint32_t flags, uint64_t stag, const void *buf, uint32_t len)`	Send a blocking MQ message. For details, see psm2_mq_send.
`psm2_mq_send2 (psm2_mq_t mq, psm2_epaddr_t dest, uint32_t flags, psm2_mq_tag_t *stag, const void *buf, uint32_t len)`	Send a blocking MQ message. For details, see psm2_mq_send2.
`psm2_mq_isend (psm2_mq_t mq, psm2_epaddr_t dest, uint32_t flags, uint64_t stag, const void *buf, uint32_t len, void *context, psm2_mq_req_t *req)`	Send a non-blocking MQ message. For details, see psm2_mq_isend.
`psm2_mq_isend2 (psm2_mq_t mq, psm2_epaddr_t dest, uint32_t flags, psm2_mq_tag_t *stag, const void *buf, uint32_t len, void *context, psm2_mq_req_t *req)`	Send a non-blocking MQ message. For details, see psm2_mq_isend2.
`psm2_mq_iprobe (psm2_mq_t mq, uint64_t rtag, uint64_t rtagsel, psm2_mq_status_t *status)`	Try to probe if a message is received to match tag selection criteria. For details, see psm2_mq_iprobe.
`psm2_mq_iprobe2 (psm2_mq_t mq, psm2_epaddr_t src, psm2_mq_tag_t *rtag, psm2_mq_tag_t *rtagsel, psm2_mq_status2_t *status)`	Try to probe if a message from the specified src process is received to match tag selection criteria. Source matching is optional. Uses 96-bit psm2_mq_tag_t instead of 64-bit tag. For details, see psm2_mq_iprobe2.
`psm2_mq_improbe (psm2_mq_t mq, uint64_t rtag, uint64_t rtagsel, psm2_mq_req_t *req, psm2_mq_status_t *status)`	Probe for a matching message, and if found, remove the message from the MQ; the message can be retrieved through the req. For details, see psm2_mq_improbe.
`psm2_mq_improbe2 (psm2_mq_t mq, psm2_epaddr_t src, psm2_mq_tag_t *rtag, psm2_mq_tag_t *rtagsel, psm2_mq_req_t *req, psm2_mq_status2_t *status)`	Probe for a matching message, and if found, remove the message from the MQ; the message can be retrieved through the req. For details, see psm2_mq_improbe2.

continued...



Function	Description
`psm2_mq_imrecv(psm2_mq_t mq, uintew_t flags, void *buf, uint32_t len, void *context, psm2_mq_req_t *req)`	Retrieves both 64-bit and 96-bit tagged messages, through the psm2_mq_req_t, matched by a previous call to psm2_mq_improbe() or psm2_mq_improbe2(). For details, see psm2_mq_imrecv.
`psm2_mq_ipeek (psm2_mq_t mq, psm2_mq_req_t *req, psm2_mq_status_t *status)`	Query for non-blocking requests ready for completion. For details, see psm2_mq_ipeek.
`psm2_mq_ipeek2 (psm2_mq_t mq, psm2_mq_req_t *req, psm2_mq_status2_t *status)`	Query for 96-bit psm2_mq_tag_t nonblocking requests ready for completion. For details, see psm2_mq_ipeek2.
`psm2_mq_wait (psm2_mq_req_t *request, psm2_mq_status_t *status)`	Wait until a non-blocking request completes. For details, see psm2_mq_wait.
`psm2_mq_wait2 (psm2_mq_req_t *request, psm2_mq_status2_t *status)`	Wait until a 96-bit psm2_mq_tag_t non-blocking request completes. For details, see psm2_mq_wait2.
`psm2_mq_test (psm2_mq_req_t *request, psm2_mq_status_t *status)`	Test if a non-blocking request is complete. For details, see psm2_mq_test.
`psm2_mq_test2 (psm2_mq_req_t *request, psm2_mq_status2_t *status)`	Test if a 96-bit psm2_mq_tag_t non-blocking request completes. For details, see psm2_mq_test2.
`psm2_mq_cancel (psm2_mq_req_t *req)`	Cancel a preposted request. For details, see psm2_mq_cancel.
`psm2_mq_get_stats (psm2_mq_t mq, psm2_mq_stats_t *stats)`	Retrieve statistics from an instantiated MQ. For details, see psm2_mq_get_stats.

3.3.5.1 psm2_mq_init

Syntax

```
psm2_error_t psm2_mq_init (psm2_ep_t ep, uint64_t tag_order_mask,
const struct psm2_optkey *opts, int numopts, psm2_mq_t *mq)
```

Initialize the MQ component for MQ communication. This function provides the Matched Queue handle necessary to perform all Matched Queue communication operations.

Parameters

`ep` Endpoint over which to initialize Matched Queue.

`tag_order_mask` Order mask hint to let MQ know what bits of the send tag are required to maintain MQ message order. In MPI parlance, this mask sets the bits that store the context (or communicator ID). You can choose to pass `PSM2_MQ_ORDERMASK_NONE` or `PSM2_MQ_ORDERMASK_ALL` to tell MQ to respectively provide no ordering guarantees or to provide ordering over all messages by ignoring the contexts of the send tags.

`opts` Set of options for Matched Queue.

numopts	Number of options passed.
mq	User-supplied storage to return the Matched Queue handle associated to the newly created Matched Queue.

Remarks

This function can be called many times to retrieve the MQ handle associated to an endpoint, but options are only considered the first time the function is called.

Postcondition

You obtain a handle to an instantiated Match Queue.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (psm2_error_register_handler).

PSM2_OK	A new Matched Queue has been instantiated across all the members of the group.

Example

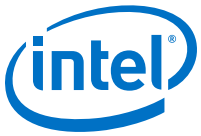
```
int try_open_endpoint_and_initialize_mq(
        psm2_ep_t *ep, // endpoint handle
        psm2_epid_t *epid, // unique endpoint ID
        psm2_uuid_t job_uuid, // unique job uuid, for ep_open
        psm2_mq_t *mq, // MQ handle initialized on endpoint 'ep'
        uint64_t communicator_bits) // Where we store our communicator or
                        // context bits in the 64-bit tag.
{
    // Simplifed open, see psm2_ep_open documentation for more info
    psm2_ep_open(job_uuid,
                 NULL, // no options
                 ep, epid);

    // We initialize a matched queue by telling PSM2 the bits that are
    // order-significant in the tag. Point-to-point ordering is not
    // maintained between senders where the communicator bits are not
    // the same.
    psm2_mq_init(ep,
                 communicator_bits,
                 NULL, // no other MQ options
                 0,    // 0 options passed
                 mq);  // newly initialized matched Queue

    return 1;
}
```

3.3.5.2 psm2_mq_finalize

Syntax

```
psm2_error_t psm2_mq_finalize (psm2_mq_t mq)
```

Finalize (close) an MQ handle.



Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK` A given Matched Queue has been freed and use of the future use of the handle produces undefined results.

3.3.5.3 psm2_mq_irecv

Syntax

```
psm2_error_t psm2_mq_irecv (psm2_mq_t mq, uint64_t rtag,
uint64_t rtagsel, uint32_t flags, void *buf, uint32_t len,
void *context, psm2_mq_req_t *req)
```

Post a receive to a Matched Queue with tag selection criteria. Function to receive a non- blocking MQ message by providing a preposted buffer. For every MQ message received on a particular MQ, the `tag` and `tagsel` parameters are used against the incoming message's send tag as described in MQ Tag Matching on page 23.

Parameters

`mq` Matched Queue handle.

`rtag` Receive tag.

`rtagsel` Receive tag selector.

`flags` Receive flags (None currently supported).

`buf` Receive buffer.

`len` Receive buffer length.

`context` User context pointer, available in `psm2_mq_status_t` upon completion.

`req` PSM2 MQ Request handle created by the preposted receive, to be used for explicitly controlling message receive completion.

Precondition

The supplied receive buffer is given to MQ to match against incoming messages unless it is cancelled via `psm2_mq_cancel` before any match occurs.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (psm2_error_register_handler).

PSM2_OK The receive buffer has successfully been posted to the MQ.

3.3.5.4 psm2_mq_irecv2

Syntax

```
psm2_error_t psm2_mq_irecv2 (psm2_mq_t mq, psm2_epaddr_t src,
psm2_mq_tag_t *rtag, psm2_mq_tag_t *rtagsel, uint32_t flags, void
*buf, uint32_t len, void *context, psm2_mq_req_t *req)
```

Post a receive to a Matched Queue with source and tag selection criteria. Function to receive a nonblocking MQ message by providing a preposted buffer. Only for every MQ message received from the specified source process on a particular MQ, the src, tag, and tagsel parameters are used against the incoming message's send tag as described in MQ Tag Matching on page 23.

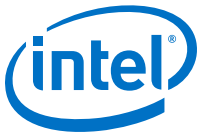
If argument src is NULL pointer, then every MQ message received from any process is used to do the matching, which is equivalent to psm2_mq_irecv.

Parameters

mq	Matched Queue handle.
src	Source EP address; PSM2_MQ_ANY_ADDR can allow a match on any sender.
rtag	Receive tag pointer.
rtagsel	Receive tag selector pointer.
flags	Receive flags (None currently supported).
buf	Receive buffer.
len	Receive buffer length.
context	User context pointer, available in psm2_mq_status2_t upon completion.
req	PSM2 MQ Request handle created by the preposted receive, to be used for explicitly controlling message receive completion.

Postcondition

The supplied receive buffer is given to MQ to match against incoming messages unless it is cancelled via psm2_mq_cancel before any match occurs.



Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK` The receive buffer has successfully been posted to the MQ.

3.3.5.5 psm2_mq_send

Syntax

```
psm2_error_t psm2_mq_send (psm2_mq_t mq, psm2_epaddr_t dest,
uint32_t flags, uint64_t stag, const void *buf, uint32_t len)
```

Send a blocking MQ message. Function to send a blocking MQ message, whereby the message is locally complete and the source data can be modified upon return.

Parameters

`mq` Matched Queue handle.

`dest` Destination EP address.

`flags` Message flags, currently:

`PSM2_MQ_FLAG_SENDSYNC` tells PSM2 to send the message synchronously, meaning that the message is not sent until the receiver acknowledges that it has matched the send with a receive buffer.

`stag` Message Send Tag.

`buf` Source buffer pointer.

`len` Length of message starting at `buf`.

Postcondition

The source buffer is reusable and the send is locally complete.

Note: This send function has been implemented to best suit `MPI_Send`.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (psm2_error_register_handler).

PSM2_OK The message has been successfully sent.

3.3.5.6 psm2_mq_send2

Syntax

```
psm2_error_t psm2_mq_send2 (psm2_mq_t mq, psm2_epaddr_t dest,
uint32_t flags, psm2_mq_tag_t *stag, const void *buf, uint32_t len)
```

Send a blocking MQ message. Function to send a blocking MQ message, whereby the message is locally complete and the source data can be modified upon return.

Parameters

mq Matched Queue handle.

dest Destination EP address.

flags Message flags, currently:

PSM2_MQ_FLAG_SENDSYNC tells PSM2 to send the message synchronously, meaning that the message is not sent until the receiver acknowledges that it has matched the send with a receive buffer.

stag Message Send Tag pointer.

buf Source buffer pointer.

len Length of message starting at buf.

Postcondition

The source buffer is reusable and the send is locally complete.

Note: This send function has been implemented to best suit MPI_Send.

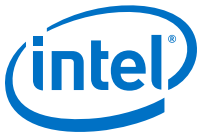
Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (psm2_error_register_handler).

PSM2_OK The message has been successfully sent.



3.3.5.7 psm2_mq_isend

Syntax

```
psm2_error_t psm2_mq_isend (psm2_mq_t mq, psm2_epaddr_t dest,
uint32_t flags, uint64_t stag, const void *buf, uint32_t len,
void *context, psm2_mq_req_t *req)
```

Send a non-blocking MQ message. Function to initiate the send of a non-blocking MQ message. You must ensure that the source data remains unmodified until the send is locally completed through a call such as `psm2_mq_wait` or `psm2_mq_test`.

Parameters

`mq` Matched Queue handle.

`dest` Destination EP address.

`flags` Message flags, currently:

`PSM2_MQ_FLAG_SENDSYNC` tells PSM2 to send the message synchronously, meaning that the message is not sent until the receiver acknowledges that it has matched the send with a receive buffer.

`stag` Message Send Tag.

`buf` Source buffer pointer.

`len` Length of message starting at `buf`.

`context` Optional user-provided pointer available in `psm2_mq_status_t` when the send is locally completed.

`req` PSM2 MQ Request handle created by the non-blocking send, to be used for explicitly controlling message completion.

Postcondition

The source buffer is not reusable and the send is not locally complete until its request is completed by either `psm2_mq_test` or `psm2_mq_wait`.

Note: This send function has been implemented to suit `MPI_Isend`.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

PSM2_OK The message has been successfully initiated.

Example

```
psm2_mq_req_t
non_blocking_send(const psm2_mq_t mq, psm2_epaddr_t dest_ep,
            const void *buf, uint32_t len,
            int context_id, int send_tag, const my_request_t *req)
{
    psm2_mq_req_t req_mq;
    // Set up our send tag, assume that "my_rank" is global and
    // represents the rank of this process in the job
    uint64_t tag = (((context_id & 0xffff) << 48) |
                    ((my_rank & 0xffff) << 32) |
                    ((send_tag & 0xffffffff)) );

    psm2_mq_isend(mq, dest_ep,
                  0, // no flags
                  tag,
                  buf,
                  len,
                  req, // this req is available in psm2_mq_status_t when one
                       // of the synchronization functions is called.
                  &req_mq);
    return req_mq;
}
```

3.3.5.8 psm2_mq_isend2

Syntax

```
psm2_error_t psm2_mq_isend2 (psm2_mq_t mq, psm2_epaddr_t dest,
uint32_t flags, psm2_mq_tag_t *stag, const void *buf, uint32_t len,
void *context, psm2_mq_req_t *req)
```

Send a non-blocking MQ message. Function to initiate the send of a non-blocking MQ message. You must ensure that the source data remains unmodified until the send is locally completed through a call such as psm2_mq_wait2 or psm2_mq_test2.

Parameters

mq Matched Queue handle.

dest Destination EP address.

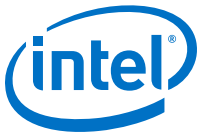
flags Message flags, currently:

PSM2_MQ_FLAG_SENDSYNC tells PSM2 to send the message synchronously, meaning that the message is not sent until the receiver acknowledges that it has matched the send with a receive buffer.

stag Message Send Tag pointer.

buf Source buffer pointer.

len Length of message starting at buf.



`context` Optional user-provided pointer available in `psm2_mq_status2_t` when the send is locally completed.

`req` PSM2 MQ Request handle created by the non-blocking send, to be used for explicitly controlling message completion.

Postcondition

The source buffer is not reusable and the send is not locally complete until its request is completed by either `psm2_mq_test2` or `psm2_mq_wait2`.

Note: This send function has been implemented to suit `MPI_Isend`.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK` The message has been successfully initiated.

3.3.5.9 psm2_mq_iprobe

Syntax

```
psm2_error_t psm2_mq_iprobe (psm2_mq_t mq, uint64_t rtag,
uint64_t rtagsel, psm2_mq_status_t *status)
```

Try to probe if a message is received to match tag selection criteria.

Function to verify whether a message matching the supplied tag and tag selectors has been received. The function is not fully matched until you provide a buffer with the successfully matching tag selection criteria through `psm2_mq_irecv`. Probing for messages may be useful if the size of the message to be received is unknown, in which case its size is available in the `msg_length` member of the returned status.

Parameters

`mq` Matched Queue handle.

`rtag` Message receive tag.

`rtagsel` Message receive tag selector.

`status` Upon return, status is filled with information regarding the matching send.

Remarks

- Function ensures progress if matching request was not found after the first attempt.
- This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error codes are returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The iprobe is successful and status is updated if non-NULL.
`PSM2_MQ_INCOMPLETE`	The iprobe is unsuccessful and status is unchanged.

3.3.5.10 psm2_mq_iprobe2

Syntax

```
psm2_error_t psm2_mq_iprobe2 (psm2_mq_t mq, psm2_epaddr_t src,
psm2_mq_tag_t *rtag, psm2_mq_tag_t *rtagsel, psm2_mq_status2_t *status);
```

Try to probe if a message is received to match tag selection criteria. If `src` is `PSM2_MQ_ANY_ADDR`, messages from all remote processes are used for the matching.

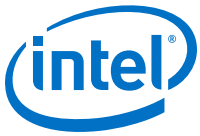
Function to verify whether a message matching the supplied tag and tag selectors has been received. The function is not fully matched until you provide a buffer with the successfully matching tag selection criteria through `psm2_mq_irecv2`. Probing for messages may be useful if the size of the message to be received is unknown, in which case its size is available in the `msg_length` member of the returned status.

Parameters

`mq`	Matched Queue handle.
`src`	Source EP address.
`rtag`	Message receive tag pointer.
`rtagsel`	Message receive tag selector pointer.
`status`	Upon return, status is filled with information regarding the matching send.

Remarks

- Function ensures progress if matching request was not found after the first attempt.
- This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.



Returns

The following error codes are returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The iprobe2 is successful and status is updated if non-NULL.
`PSM2_MQ_INCOMPLETE`	The iprobe2 is unsuccessful and status is unchanged.

3.3.5.11 psm2_mq_improbe

Syntax

```
psm2_mq_improbe (psm2_mq_t mq, uint64_t rtag, uint64_t rtagsel,
psm2_mq_req_t *req, psm2_mq_status_t *status)
```

Probe for a matching message, and if found, remove the message from the MQ; the message can be retrieved through the `req`.

Parameters

`mq`	Matched Queue handle.
`rtag`	Message receive tag.
`rtagsel`	Message receive tag selector.
`req`	PSM2 MQ Request handle, to be used for receiving the matched message.
`status`	Upon return, status is filled with information regarding the matching send.

Remarks

- Function ensures progress if matching request was not found after the first attempt.
- This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error codes are returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The improbe is successful and status is updated if non-NULL.
`PSM2_MQ_INCOMPLETE`	The improbe is unsuccessful and status is unchanged.

3.3.5.12 psm2_mq_improbe2

Syntax

```
psm2_mq_improbe2 (psm2_mq_t mq, psm2_epaddr_t src, psm2_mq_tag_t *rtag,
psm2_mq_tag_t *rtagsel, psm2_mq_req_t *req, psm2_mq_status2_t *status)
```

Probe for a matching message, and if found, remove the message from the MQ; the message can be retrieved through the `req`.

Parameters

`mq`	Matched Queue handle.
`rtag`	Message receive tag pointer.
`rtagsel`	Message receive tag selector pointer.
`req`	PSM2 MQ Request handle, to be used for receiving the matched message.
`status`	Upon return, status is filled with information regarding the matching send.

Remarks

- Function ensures progress if matching request was not found after the first attempt.
- This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error codes are returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The improbe2 is successful and status is updated if non-NULL.
`PSM2_MQ_INCOMPLETE`	The improbe2 is unsuccessful and status is unchanged.

3.3.5.13 psm2_mq_imrecv

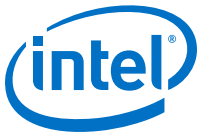
Syntax

```
psm2_mq_imrecv (psm2_mq_t mq, uintew_t flags, void *buf,
uint32_t len, void *context, psm2_mq_req_t *req)
```

`psm2_mq_imrecv()` retrieves both 64-bit and 96-bit tagged messages through the req handle returned by the appropriate improbe function.

Parameters

`mq`	Matched Queue handle.



`flags`	Receive flags (None currently supported).
`buf`	Receive buffer.
`len`	Receive buffer length.
`context`	User context pointer, available in `psm2_mq_status_t` upon completion.
`req`	PSM2 MQ Request handle created by the preposted receive, to be used for explicitly controlling message receive completion.

The following error codes are returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

`PSM2_OK`	The function is successful and status is updated if non-NULL.
`PSM2_MQ_INCOMPLETE`	The function is unsuccessful and status is unchanged.

3.3.5.14 psm2_mq_ipeek

Syntax

```
psm2_error_t psm2_mq_ipeek (psm2_mq_t mq, psm2_mq_req_t *req, psm2_mq_status_t
*status)
```

Query for non-blocking requests ready for completion.

Function to query a particular MQ for non-blocking requests that are ready for completion. Requests "ready for completion" are not actually considered complete by MQ until they are returned to the MQ library through `psm2_mq_wait` or `psm2_mq_test`.

If you can deal with consuming request completions in the order in which they complete, this function can be used both for completions and for ensuring progress. The latter requirement is satisfied when you peek an empty completion queue as a side effect of always aggressively peeking and completing all of an MQ's requests ready for completion.

Parameters

`mq`	Matched Queue handle.
`req`	MQ non-blocking request.

`status` Optional MQ status, can be NULL.

Postcondition

You have ensured progress if the function returns `PSM2_MQ_INCOMPLETE`.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

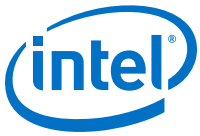
Returns

The following error codes are returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The peek is successful and req is updated with a request ready for completion. If status is non-NULL, it is also updated.
`PSM2_MQ_INCOMPLETE`	The peek is not successful, meaning that there are no further requests ready for completion. The contents of req and status remain unchanged.

Example

```
// Example that uses psm2_mq_ipeek to make progress instead of psm2_poll
// We return the amount of non-blocking requests that we've completed
int main_progress_loop(psm2_mq_t mq)
{
    int num_completed = 0;
    psm2_mq_req_t req;
    psm2_mq_status_t status;
    psm2_error_t err;
    my_request_t *myreq;

    do {
        err = psm2_mq_ipeek(mq, &req,
                            NULL); // No need for status in ipeek here
        if (err == PSM2_MQ_INCOMPLETE)
            return num_completed;
        else if (err != PSM2_OK)
            goto errh; num_completed++;

        // We obtained 'req' at the head of the completion queue.
        // We can now free the request with PSM2 and obtain our
        // original request from the status' context
        err = psm2_mq_test(&req, // is marked as invalid
                           &status); // we need the status
        myreq = (my_request_t *) status.context;

        // handle the completion for myreq whether myreq is a
        // posted receive or a non-blocking send.

    }
    while (1);
}
```



3.3.5.15 psm2_mq_ipeek2

Syntax

```
psm2_error_t psm2_mq_ipeek2 (psm2_mq_t mq, psm2_mq_req_t *req, psm2_mq_status2_t
*status)
```

Query for non-blocking requests ready for completion.

Function to query a particular MQ for non-blocking requests that are ready for completion. Requests "ready for completion" are not actually considered complete by MQ until they are returned to the MQ library through `psm2_mq_wait2` or `psm2_mq_test2`.

If you can deal with consuming request completions in the order in which they complete, this function can be used both for completions and for ensuring progress. The latter requirement is satisfied when you peek an empty completion queue as a side effect of always aggressively peeking and completing all of an MQ's requests ready for completion.

Parameters

`mq`	Matched Queue handle.
`req`	MQ non-blocking request.
`status`	Optional MQ status, can be NULL.

Postcondition

You have ensured progress if the function returns `PSM2_MQ_INCOMPLETE`.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error codes are returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The peek is successful and `req` is updated with a request ready for completion. If `status` is non-NULL, it is also updated.
`PSM2_MQ_INCOMPLETE`	The peek is not successful, meaning that there are no further requests ready for completion. The contents of `req` and `status` remain unchanged.

3.3.5.16 psm2_mq_wait

Syntax

```
psm2_error_t psm2_mq_wait (psm2_mq_req_t *request, psm2_mq_status_t *status)
```

Wait until a non-blocking request completes. Function to wait on requests created from either preposted receive buffers or non-blocking sends. This is the only blocking function in the MQ interface and it polls until the request is complete as per the progress semantics explained in MQ Progress Requirements on page 26.

Parameters

`request` MQ non-blocking request.

`status` Updated if non-NULL when request successfully completes.

Precondition

You have obtained a valid MQ request by calling `psm2_mq_isend` or `psm2_mq_irecv` and you pass a pointer to enough storage to write the output of a `psm2_mq_status_t` or NULL if status is to be ignored.

Since MQ internally ensures progress, you need not ensure that progress is made prior to calling this function.

Postcondition

The request is assigned the value `PSM2_MQ_REQINVALID` and all associated MQ request storage is released back to the MQ library.

Remarks

This function ensures progress on the endpoint as long as the request is incomplete. The `status` can be NULL, in which case no status is written upon completion. If `request` is `PSM2_MQ_REQINVALID`, the function returns immediately.

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

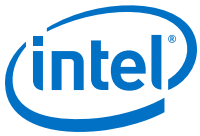
The following error code is returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK` The request is complete or the value of `request` was `PSM2_MQ_REQINVALID`.

3.3.5.17 psm2_mq_wait2

Syntax

```
psm2_error_t psm2_mq_wait2 (psm2_mq_req_t *request, psm2_mq_status2_t *status)
```



Wait until a non-blocking request completes. Function to wait on requests created from either preposted receive buffers or non-blocking sends. This is the only blocking function in the MQ interface and it polls until the request is complete as per the progress semantics explained in MQ Progress Requirements on page 26.

Parameters

`request` MQ non-blocking request.

`status` Updated if non-NULL when request successfully completes.

Precondition

You have obtained a valid MQ request by calling `psm2_mq_isend2` or `psm2_mq_irecv2` and you pass a pointer to enough storage to write the output of a `psm2_mq_status2_t` or NULL if status is to be ignored.

Since MQ internally ensures progress, you need not ensure that progress is made prior to calling this function.

Postcondition

The request is assigned the value `PSM2_MQ_REQINVALID` and all associated MQ request storage is released back to the MQ library.

Remarks

This function ensures progress on the endpoint as long as the request is incomplete. The `status` can be NULL, in which case no status is written upon completion. If `request` is `PSM2_MQ_REQINVALID`, the function returns immediately.

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following error code is returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK` The request is complete or the value of `request` was `PSM2_MQ_REQINVALID`.

3.3.5.18 psm2_mq_test

Syntax

```
psm2_error_t psm2_mq_test (psm2_mq_req_t *request, psm2_mq_status_t *status)
```

Test whether a non-blocking request is complete. Function to test requests created from either preposted receive buffers or non-blocking sends for completion. Unlike `psm2_mq_wait`, this function tests requests for completion and never ensures progress directly or indirectly. If you choose to exclusively test requests for completion, you must ensure progress, using functions described in MQ Progress Requirements on page 26.

It can be useful to construct higher-level completion tests over arrays to test some, all, or any request that has completed. If you are testing arrays of requests for completion, Intel recommends that you only ensure progress once, for better performance.

Parameters

`request` MQ non-blocking request.

`status` Updated if non-NULL and the request successfully completes.

Precondition

You obtain a valid MQ request by calling `psm2_mq_isend` or psm2_mq_irecv and pass a pointer to enough storage to write the output of a `psm2_mq_status_t` or NULL if status is to be ignored.

You must ensure progress on the Matched Queue if `psm2_mq_test` is exclusively used for guaranteeing request completions.

Postcondition

If the request is complete, the request is assigned the value `PSM2_MQ_REQINVALID` and all associated MQ request storage is released back to the MQ library. If the request is incomplete, the contents of `request` are unchanged.

You must ensure progress on the Matched Queue if `psm2_mq_test` is exclusively used for guaranteeing request completions.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

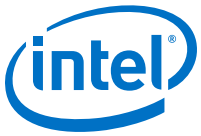
Returns

The following two errors are always returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The request is complete or the value of request was `PSM2_MQ_REQINVALID`.
`PSM2_MQ_INCOMPLETE`	The request is not complete and `request` is unchanged.

Example

```
// Function that returns the first completed request in an array
// of requests.
void * user_testany(psm2_ep_t ep, psm2_mq_req_t *allreqs, int nreqs)
{
    int i;
    void *context = NULL;

    // Ensure progress only once
    psm2_poll(ep);
```



```
    // Test for at least one completion and return its context
    psm2_mq_status_t stat;
    for (i = 0; i < nreqs; i++) {
        if (psm2_mq_test(&allreqs[i], &stat) == PSM2_OK) {
            context = stat.context;
            break;
        }
    }
    return context;
}
```

3.3.5.19 psm2_mq_test2

Syntax

```
psm2_error_t psm2_mq_test2 (psm2_mq_req_t *request, psm2_mq_status2_t *status)
```

Test whether a non-blocking request is complete. Function to test requests created from either preposted receive buffers or non-blocking sends for completion. Unlike psm2_mq_wait2, this function tests request for completion and never ensures progress directly or indirectly. If you choose to exclusively test requests for completion, you must ensure progress, using functions described in MQ Progress Requirements on page 26.

It can be useful to construct higher-level completion tests over arrays to test some, all, or any request that has completed. If you are testing arrays of requests for completion, Intel recommends that you only ensure progress once, for better performance.

Parameters

`request` MQ non-blocking request.

`status` Updated if non-NULL and the request successfully completes.

Precondition

You obtain a valid MQ request by calling `psm2_mq_isend2` or `psm2_mq_irecv2` and pass a pointer to enough storage to write the output of a `psm2_mq_status2_t` or NULL if status is to be ignored.

You must ensure progress on the Matched Queue if `psm2_mq_test2` is exclusively used for guaranteeing request completions.

Postcondition

If the request is complete, the request is assigned the value `PSM2_MQ_REQINVALID` and all associated MQ request storage is released back to the MQ library. If the request is incomplete, the contents of `request` are unchanged.

You must ensure progress on the Matched Queue if `psm2_mq_test2` is exclusively used for guaranteeing request completions.

Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

The following two errors are always returned. Other errors are handled by the PSM2 error handler (`psm2_error_register_handler`).

`PSM2_OK`	The request is complete or the value of `request` was `PSM2_MQ_REQINVALID`.
`PSM2_MQ_INCOMPLETE`	The request is not complete and `request` is unchanged.

3.3.5.20 psm2_mq_cancel

Syntax

```
psm2_error_t psm2_mq_cancel (psm2_mq_req_t *req)
```

Cancel a preposted request. Function to cancel a preposted receive request returned by `psm2_mq_irecv`.

It is currently illegal to cancel a send request initiated with `psm2_mq_isend`.

Precondition

You have obtained a valid MQ request by calling `psm2_mq_isend` or `psm2_mq_irecv` and you pass a pointer to enough storage to write the output of a `psm2_mq_status_t` or NULL if status is to be ignored.

Postcondition

Whether the cancel is successful or not, you return the request to the library using `psm2_mq_test` or `psm2_mq_wait`.

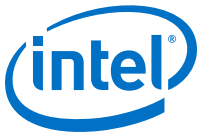
Remarks

This function may be called simultaneously from multiple threads as long as different MQ arguments are used in each of the calls.

Returns

Only the following errors can be returned directly, without being handled by the error handler (`psm2_error_register_handler`).

`PSM2_OK`	The request could be successfully cancelled such that the preposted receive buffer could be removed from the preposted receive queue before a match occurred. The associated request remains unchanged and you must still return the storage to the MQ library.



PSM2_MQ_INCOMPLETE	The request could not be successfully cancelled since the preposted receive buffer has already matched an incoming message. The request remains unchanged.

3.3.5.21 psm2_mq_get_stats

Syntax

```
psm2_mq_get_stats (psm2_mq_t mq, psm2_mq_stats_t *stats)
```

Retrieve statistics from an instantiated MQ.

Parameters

mq	Matched Queue handle.
stats	MQ Stats handle.

3.3.6 PSM2 Matched Queue Options

MQ options can be modified at any point at runtime, unless otherwise noted. The following example shows how to retrieve the current message size at which messages are sent as synchronous.

```
uint32_t get_hfirv_size(psm2_mq_t mq)
{
    uint32_t rvsize;
    psm2_getopt(mq, PSM2_MQ_RNDV_HFI_SZ, &rvsize);
    return rvsize;
}
```

3.3.6.1 Defines

Table 12. Matched Queue Options Defines

Define	Description
#define PSM2_MQ_RNDV_HFI_SZ	[uint32_t] Size at which to start enabling rendezvous messaging for Intel® Omni-Path messages. If unset, defaults to values between 56000 and 72000 depending on the system configuration.
#define PSM2_MQ_RNDV_SHM_SZ	[uint32_t] Size at which to start enabling rendezvous messaging for shared memory (intra-node) messages. If unset, defaults to 64000 bytes.
#define PSM2_MQ_MAX_SYSBUF_MBYTES	[uint32_t] Maximum amount of bytes to allocate for unexpected messages. Messages that would cause memory allocation to exceed this amount are dropped.

3.3.6.2 Functions

Table 13. Matched Queue Options Functions

Function	Description
`psm2_mq_getopt (psm2_mq_t mq, int option, void *value)`	Get an MQ option. For details, see: psm2_mq_getopt.
`psm2_mq_setopt (psm2_mq_t mq, int option, const void *value)`	Set an MQ option. For details, see: psm2_mq_setopt.

3.3.6.2.1 psm2_mq_getopt

Syntax

```
psm2_error_t psm2_mq_getopt (psm2_mq_t mq, int option, void *value)
```

Get an MQ option. Function to retrieve the value of an MQ option.

Parameters

`mq` Matched Queue handle.

`option` Index of option to retrieve. Possible values are:

- `PSM2_MQ_RNDV_HFI_SZ`
- `PSM2_MQ_RNDV_SHM_SZ`
- `PSM2_MQ_MAX_SYSBUF_MBYTES`

`value` Pointer to storage that can be used to store the value of the option to be set. You must ensure that the pointer points to a memory location large enough to accommodate the value associated to the type. Each option documents the size associated to its value.

Returns

`PSM2_OK` If option could be retrieved.

`PSM2_PARAM_ERR` If the option is not a valid option number.

3.3.6.2.2 psm2_mq_setopt

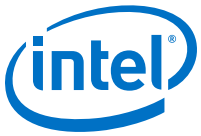
Syntax

```
psm2_error_t psm2_mq_setopt (psm2_mq_t mq, int option, const void *value)
```

Set an MQ option. Function to set the value of an MQ option.

Parameters

`mq` Matched Queue handle.



`option` Index of option to retrieve. Possible values are:

- `PSM2_MQ_RNDV_HFI_SZ`
- `PSM2_MQ_RNDV_SHM_SZ`
- `PSM2_MQ_MAX_SYSBUF_MBYTES`

`value` Pointer to storage that contains the value to be updated for the supplied option number. You must ensure that the pointer points to a memory location with a correct size.

Returns

`PSM2_OK`	If option could be retrieved.
`PSM2_PARAM_ERR`	If the option is not a valid option number.
`PSM2_OPT_READONLY`	If the option to be set is a read-only option (currently no MQ options are read- only).

4.0 Intel® PSM2 Sample Program

This section describes a sample program that can be used to verify basic PSM2 functionality, similar to *Hello World* code.

4.1 Prerequisites

To run the sample program, you need a built copy of PSM2 in your local directory.

4.2 Setting Up the Program

1. Start two instances of this program from the same working directory. These processes can execute on the same host, or on two hosts connected with Intel® Omni-Path Architecture (Intel® OPA).
2. Compile using this command:

```
gcc psm2-demo.c -o psm2-demo -lpsm2
```

3. Run one instance as a server process using the command:

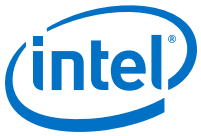
```
./psm2-demo -s
```

4. Run the other instance as a client process using the command:

```
./psm2-demo
```

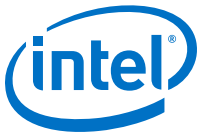
4.3 Sample Code

```
/*
    PSM2 example program.
    Start two instances of this program from the same working directory.
    These processes can execute on the same host, or on two hosts connected with
    OPA.

    Compile with: gcc psm2-demo.c -o psm2-demo -lpsm2
    Run as:  ./psm2-demo -s # this is the server process
        and: ./psm2-demo    # this is the client process

    Copyright(c) 2015 Intel Corporation.
*/
#include <stdio.h>
#include <psm2.h>         /* required for core PSM2 functions */
#include <psm2_mq.h>      /* required for PSM2 MQ functions (send, recv, etc) */
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

#define BUFFER_LENGTH 80
#define CONNECT_ARRAY_SIZE 8
```



```
void die(char *msg, int rc){
    fprintf(stderr, "%s: %d\n", msg, rc);
    exit(1);
}

/* Helper functions to find the server's PSM2 endpoint identifier (epid). */
psm2_epid_t find_server(){
    FILE           *fp = NULL;
    psm2_epid_t server_epid = 0;

    printf("PSM2 client waiting for epid mapping file to appear...\n");
    while (!fp){
        sleep(1);
        fp = fopen("psm2-demo-server-epid", "r");
    }
    fscanf(fp, "%lx", &server_epid);
    fclose(fp);
    printf("PSM2 client found server epid = 0x%lx\n", server_epid);
    return server_epid;
}

void write_epid_to_file(psm2_epid_t myepid) {
    FILE    *fp;

    fp = fopen("psm2-demo-server-epid", "w");
    if (!fp){
        fprintf(stderr,
            "Exiting, couldn't write server's epid mapping file: ");
        die(strerror(errno), errno);
    }
    fprintf(fp, "0x%lx", myepid);
    fclose(fp);
    printf("PSM2 server wrote epid = 0x%lx to file.\n", myepid);
    return;
}

int main(int argc, char **argv){
    struct psm2_ep_open_opts o;
    psm2_uuid_t uuid;
    psm2_ep_t    myep;
    psm2_epid_t myepid;
    psm2_epid_t server_epid;
    psm2_epid_t epid_array[CONNECT_ARRAY_SIZE];
    int            epid_array_mask[CONNECT_ARRAY_SIZE];
    psm2_error_t    epid_connect_errors[CONNECT_ARRAY_SIZE];
    psm2_epaddr_t epaddr_array[CONNECT_ARRAY_SIZE];

    int    rc;
    int    ver_major = PSM2_VERNO_MAJOR;
    int    ver_minor = PSM2_VERNO_MINOR;
    char       msgbuf[BUFFER_LENGTH];
    psm2_mq_t    q;
    psm2_mq_req_t req_mq;
    int    is_server = 0;

    if (argc > 2){
        die("To run in server mode, invoke as ./psm2-demo -s\n" \
            "or run in client mode, invoke as ./psm2-demo\n" \
            "Wrong number of args", argc);
    }

    is_server = argc - 1; /* Assume any command line argument is -s */

    memset(uuid, 0, sizeof(psm2_uuid_t)); /* Use a UUID of zero */

    /* Try to initialize PSM2 with the requested library version.
     *    In this example, given the use of the PSM2_VERNO_MAJOR and MINOR
     *    as defined in the PSM2 headers, ensure that we are linking with
     *    the same version of PSM2 as we compiled against. */
```

```
    if ((rc = psm2_init(&ver_major, &ver_minor)) != PSM2_OK){
    die("couldn't init", rc);
    }
    printf("PSM2 init done.\n");

    /* Setup the endpoint options struct */
    if ((rc = psm2_ep_open_opts_get_defaults(&o)) != PSM2_OK){
        die("couldn't set default opts", rc);
    }
    printf("PSM2 opts_get_defaults done.\n");

    /* Attempt to open a PSM2 endpoint. This allocates hardware resources. */
    if ((rc = psm2_ep_open(uuid, &o, &myep, &myepid)) != PSM2_OK){
        die("couldn't psm2_ep_open()", rc);
    }
    printf("PSM2 endpoint open done.\n");

    if (is_server){
        write_epid_to_file(myepid);
    } else {
        server_epid = find_server();
    }

    if (is_server){
        /* Server does nothing here. A connection does not have to be
         * established to receive messages. */
        printf("PSM2 server up.\n");
    } else {
        /* Setup connection request info */
        /* PSM2 can connect to a single epid per request,
         *    or an arbitrary number of epids in a single connect call.
         *    For this example, use part of an array of
         *    connection requests. */
        memset(epid_array_mask, 0, sizeof(int) * CONNECT_ARRAY_SIZE);
        epid_array[0] = server_epid;
        epid_array_mask[0] = 1;

        /* Begin the connection process.
         *    note that if a requested epid is not responding,
         *    the connect call will still return OK.
         *    The errors array will contain the state of individual
         *    connection requests. */
        if ((rc = psm2_ep_connect(myep,
                    CONNECT_ARRAY_SIZE,
                    epid_array,
                    epid_array_mask,
                    epid_connect_errors,
                    epaddr_array,
                    0 /* no timeout */
                )) != PSM2_OK){
            die("couldn't ep_connect", rc);
        }
        printf("PSM2 connect request processed.\n");

        /* Now check if our connection to the server is ready */
        if (epid_connect_errors[0] != PSM2_OK){
            die("couldn't connect to server", epid_connect_errors[0]);
        }
        printf("PSM2 client-server connection established.\n");
    }

    /* Setup our PSM2 message queue */
    if ((rc = psm2_mq_init(myep, PSM2_MQ_ORDERMASK_NONE, NULL, 0, &q))
        != PSM2_OK){
        die("couldn't initialize PSM2 MQ", rc);
    }
    printf("PSM2 MQ init done.\n");

    if (is_server){
        /* Post the receive request */
        if ((rc = psm2_mq_irecv(q,
```



```
                        0xABCD,         /* message tag */
                        (uint64_t)-1,   /* message tag mask */
                        0,              /* no flags */
                        msgbuf, BUFFER_LENGTH,
                        NULL,           /* no context to add */
                        &req_mq         /* track irecv status */
                    )) != PSM2_OK){
                die("couldn't post psm2_mq_irecv()", rc);
            }
            printf("PSM2 MQ irecv() posted\n");

            /* Wait until the message arrives */
            if ((rc = psm2_mq_wait(&req_mq, NULL)) != PSM2_OK){
                die("couldn't wait for the irecv", rc);
            }
            printf("PSM2 MQ wait() done.\n");
            printf("Message from client:\n");
            printf("%s", msgbuf);

            unlink("psm2-demo-server-epid");
        } else {
            /* Say hello */
            snprintf(msgbuf, BUFFER_LENGTH,
                "Hello world from epid=0x%lx, pid=%d.\n",
                myepid, getpid());

            if ((rc = psm2_mq_send(q,
                        epaddr_array[0], /* destination epaddr */
                        0,               /* no flags */
                        0xABCD,          /* tag */
                        msgbuf, BUFFER_LENGTH
                    )) != PSM2_OK){
                die("couldn't post psm2_mq_isend", rc);
            }
            printf("PSM2 MQ send() done.\n");
        }

        /* Close down the MQ */
        if ((rc = psm2_mq_finalize(q)) != PSM2_OK){
            die("couldn't psm2_mq_finalize()", rc);
        }
        printf("PSM2 MQ finalized.\n");

        /* Close our ep, releasing all hardware resources.
         * Try to close all connections properly */
        if ((rc = psm2_ep_close(myep, PSM2_EP_CLOSE_GRACEFUL,
                    0 /* no timeout */)) != PSM2_OK){
            die("couldn't psm2_ep_close()", rc);
        }
        printf("PSM2 ep closed.\n");

        /* Release all local PSM2 resources */
        if ((rc = psm2_finalize()) != PSM2_OK){
            die("couldn't psm2_finalize()", rc);
        }
        printf("PSM2 shut down, exiting.\n");

        return 0;
    }
```