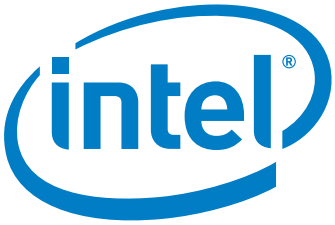
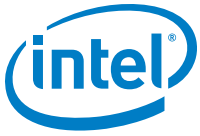


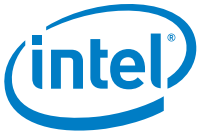
Intel® Performance Scaled Messaging 2 (PSM2)

Programmer's Guide

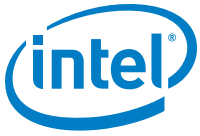
November 2015

Order No.: H76473-1.0



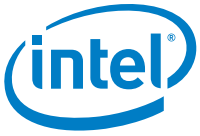


Contents



Figures

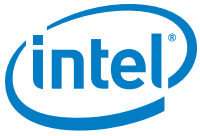
N/A



Tables

Revision History

Date	Revision	Description
November 2015	1.0	Document has been updated for Revision 1.0. Starting with this release, the Intel® PSM2 API library is a stand-alone package with its own documentation.
August 2015	0.7	Document has been updated for Revision 0.7. Added Chapter 5.0, "Intel® PSM2 Sample Program."
April 2015	0.5	First release of the PSM2 manual with Intel® Omni-Path extensions.



1.0 Introduction

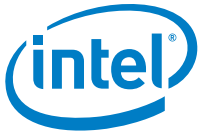
This manual is a reference for programmers working with the Intel® PSM2 Application Programming Interface (API). The Performance Scaled Messaging 2 API (PSM2 API) is a low-level user-level communications interface.

Note: In the previous release, the Intel® PSM API was updated to include PSM2 support for the Intel® Omni-Path family of products. However, there has been an implementation change and starting with this release, the Intel® PSM2 API library is a stand-alone package with its own documentation. PSM2 has been extended from the Intel® True Scale PSM interface, but is not compatible unless used with the provided interface library. Refer to the *Intel® Omni-Path Fabric Host Software User Guide* for details.

1.1 Documentation Conventions

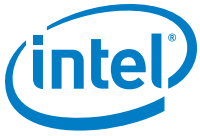
This guide uses the following documentation conventions:

- *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, and links to Web sites are also shown in blue. For example:
  - See "License Agreements" on page 7.
  - For more information, visit www.intel.com.
- Text in **bold** font indicates user interface elements such as a menu items, buttons, check boxes, or column headings. For example:
  - Click the **Start** button, point to **Programs**, point to **Accessories**, and then click **Command Prompt**.
- Text in `Courier` font indicates a file name, directory path, or command line text. For example:
  - Enter the following command: `sh ./install.bin`
- Key names and key strokes are indicated with UPPERCASE:
  - Press CTRL+P.
- Text in *italics* indicates terms, emphasis, variables, or document titles. For example:
  - For a complete listing of license agreements, refer to the *Intel Software End User License Agreement*.



1.2 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.



2.0 Intel® PSM2 Messaging API

The Intel® PSM2 (Performance Scaled Messaging 2) 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 it transparently implements two levels of communication: intra-node shared memory communication and inter-node communication.

2.1 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.

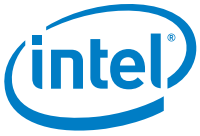
PSM2 is currently a single-threaded library. This means that you cannot make any concurrent PSM2 library calls. While threads may be a valid execution model for the wider set of potential PSM2 clients, applications should currently expect better effective use of Intel® Omni-Path resources (and hence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

Except where noted, PSM2 does not assume an SPMD (single program, multiple data) parallel model and extends to MPMD (multiple program, multiple data) 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.

2.2 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.

2.3 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 (Section 3.1, “Matched Queues Interface” on page 14). 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 that implement parallel communication models that deviate from the Matched Queue semantics.

2.4 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 Section 3.1.4, “MQ Progress Requirements” on page 17.

2.5 PSM2 Completion Semantics

PSM2 currently only implements the MQ component, which documents its own message completion semantics (see Section 3.1.3, “MQ Completion Semantics” on page 16).

2.6 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.

2.7 Environment Variables

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

2.7.1 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.

MPI users not using Intel® Omni-Path can set this to enable running in shared memory mode on a single node. It is automatically set for Intel® Omni-Path MPI.

Default: PSM2_DEVICES="self,shm,hfi"

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

2.7.2 PSM2_MEMORY

Memory usage mode. This scales the resource allocation according to normal (default) size clusters or large clusters. Normal is expected to be sufficient even on very large cluster sizes, but large is available in case some of the resources allocation is too restrictive. Although this case is unlikely, comprehensive error messages are apparent to large-scale users if normal is insufficient.

Default: PSM2_MEMORY=normal

2.7.3 PSM2_MQ_SENDREQS_MAX

This sets the maximum number of `isend` requests in flight.

Default: `PSM2_MQ_SENDREQS_MAX=1048576`

2.7.4 PSM2_MQ_RECVREQS_MAX

This sets the maximum number of `irecv` requests in flight.

Default: `PSM2_MQ_RECVREQS_MAX=1048576`

2.7.5 PSM2_MQ_RNDV_HFI_THRESH

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

Default: `PSM2_MQ_RNDV_HFI_THRESH=64000`

2.7.6 PSM2_MQ_RNDV_SHM_THRESH

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

Default: `PSM2_MQ_RNDV_SHM_THRESH=16000`

2.7.7 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. Either `PSM2_RANKS_PER_CONTEXT` or `PSM2_SHAREDCONTEXTS_MAX` can be used in a particular job, but not both. If both are used and the settings are incompatible, then PSM2 reports an error and the job fails to start up.

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.

2.7.8 PSM2_RCVTHREAD

PSM2 uses a communication thread to help parts of the Intel® Omni-Path MPI protocol ensure communication progress. 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`

2.7.9 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
```

2.7.10 PSM2_SHAREDCONTEXTS_MAX

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_SHAREDCONTEXTS_MAX` environment variable. The default is to use all possible contexts.

Default: `PSM2_SHAREDCONTEXTS_MAX=8`

2.7.11 PSM2_TID

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

Default: `PSM2_TID=0x1`

2.7.12 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`

2.8 HFI Environment Variables

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

2.8.1 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.

2.8.2 HFI_NO_CPUAFFINITY

Prevents PSM2 from setting affinity. By default, no affinity is set.

Default: `HFI_NO_CPUAFFINITY=NO`

2.8.3 HFI_UNIT

Device Unit number. Used to restrict the number of contexts used on a 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. By default, `HFI_UNIT` is unset; all available contexts from all units are autodetected and used, and are made available to MPI jobs.

`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).

3.0 Intel® PSM2 Component Documentation

3.1 Matched Queues Interface

The 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.

3.1.1 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 = 0x3333000300000000;
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.

3.1.2 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.

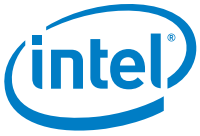
Whenever a match occurs, whether the message is expected or unexpected, you must ensure that the message is not truncated. Message truncation occurs when the size of the preposted buffer is less than the size of the incoming matched message. MQ correctly handles message truncation by always copying the 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.

3.1.3 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.

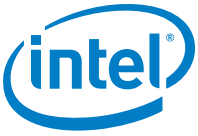


3.1.4 MQ Progress Requirements

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 process 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.



4.0 Intel® PSM2 Component Functional Documentation

4.1 PSM2 Initialization and Maintenance

4.1.1 Data Structures

`struct psm2_optkey`

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

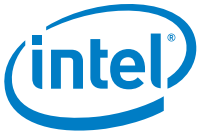
Data Fields:

`uint32_t key`	Option key.
`void * value`	Key value.

4.1.2 Defines

Table 1. 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.



4.1.3 Typedefs

Table 2. 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.

4.1.4 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 3. Error Type Enumerators (Sheet 1 of 2)

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.

Table 3. Error Type Enumerators (Sheet 2 of 2)

Enumerator	Description
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.
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.

4.1.5 Functions

Table 4. Initialization and Maintenance Functions

Function	Description
`psm2_init (int *api_verno_major, int *api_verno_minor)`	Initialize PSM2 interface. For details, see: Section 4.1.5.1.
`psm2_finalize (void)`	Finalize PSM2 interface. For details, see: Section 4.1.5.2.
`psm2_error_register_handler (psm2_ep_t ep, const psm2_ep_errhandler_t errhandler)`	PSM2 error handler registration. For details, see: Section 4.1.5.3.
`psm2_error_defer (psm2_error_token_t err_token)`	PSM2 deferred error handler. For details, see: Section 4.1.5.4.
`psm2_error_get_string (psm2_error_t error)`	Get generic error string from error. For details, see: Section 4.1.5.5.

4.1.5.1 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;
}
```

4.1.5.2 psm2_finalize

Syntax:

```
psm2_error_t psm2_finalize (void)
```

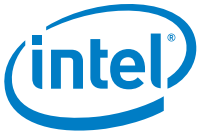
Finalize PSM2 interface. Single call to finalize PSM2 and close all unclosed endpoints.

Postcondition:

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

Returns:

`PSM2_OK`
: Always returns `PSM2_OK`.



4.1.5.3 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 Section 2.6, “PSM2 Error Handling” on page 9).

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`.

4.1.5.4 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.

4.1.5.5 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.

4.2 PSM2 Device Endpoint Management

4.2.1 Data Structures

`struct 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.

Each option is documented in `psm2_ep_open`.

Data Fields:

`int64_t timeout`	Timeout in nanoseconds to open device.
`int unit`	Intel® Omni-Path Unit ID to open on.
`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).

4.2.2 Defines

Table 5. 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.

4.2.3 Typedefs

Table 6. 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. 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.

4.2.4 Functions

Table 7. Endpoint Functions (Sheet 1 of 2)

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_map_nid_hostname(int num, const uint64_t *nids, const char **hostnames)`	Provide a mapping from network ID (LID) to hostnames. For details, see: Section 4.2.4.1.
`psm2_ep_num_devunits (uint32_t *num_units)`	List the number of available Intel® Omni-Path units. For details, see: Section 4.2.4.2.
`psm2_uuid_generate (psm2_uuid_t uuid_out)`	Utility to generate UUIDs for psm2_ep_open. For details, see: Section 4.2.4.3.
`psm2_ep_open_opts_get_defaults (struct psm2_ep_open_opts *opts);`	Endpoint open default options. For details, see: Section 4.2.4.4.
`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: Section 4.2.4.5.
`psm2_ep_epid_share_memory (psm2_ep_t ep, psm2_epid_t epid, int *result)`	Endpoint shared memory query. For details, see: Section 4.2.4.6.

Table 7. Endpoint Functions (Sheet 2 of 2)

Function	Description
psm2_ep_close (psm2_ep_t ep, int mode, int64_t timeout)	Close endpoint. For details, see: Section 4.2.4.7.
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: Section 4.2.4.8.
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: Section 4.2.4.9.
psm2_poll (psm2_ep_t ep)	Ensure endpoint communication progress. For details, see: Section 4.2.4.10.
psm2_epaddr_setlabel (psm2_epaddr_t epaddr, const char *epaddr_label_string)	Set a user-determined ep address label. For details, see: Section 4.2.4.11.

4.2.4.1 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.

4.2.4.2 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`.

4.2.4.3 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.

4.2.4.4 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.

4.2.4.5 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).

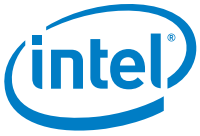
Parameters:

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 Section 4.1.5.1, "psm2_init" on page 21.

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 uid 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`.

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 15 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`.

Warning:

Currently, PSM2 limits you to calling `psm2_ep_open` only once per process; subsequent calls fail. Multiple endpoints per process may be enabled in a future release.

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;
}
```

4.2.4.6 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 endpiont.

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.

4.2.4.7 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`.

`timeout`
: How long to wait in nanoseconds if mode is `PSM2_EP_CLOSE_GRACEFUL`, 0 waits forever. If mode is `PSM2_EP_CLOSE_FORCE`, this parameter is ignored.

The following errors are 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.

`PSM2_EP_CLOSE_TIMEOUT`
: Endpoint could not be successfully closed within timeout.

4.2.4.8 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;
}
```

4.2.4.9 psm2_ep_disconnect

Syntax:

psm2_error_t 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. Function to non-collectively disconnect a connection to a set of endpoint addresses and free the endpoint addresses. 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 amount 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;
}
```

4.2.4.10 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 17 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.

4.2.4.11 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`.

4.3 PSM2 Matched Queues

4.3.1 Modules

PSM2 Matched Queue Options.

4.3.2 Data Structures

Table 8. Matched Queues Data Structures

Data Structure	Description
psm2_mq_status	MQ Non-blocking operation status structure. For details, see: Section 4.3.2.1.
psm2_mq_stats	MQ statistics structure. For details, see: Section 4.3.2.2.
psm2_tag_t	MQ 96-bit tag structure For details, see: Section 4.3.2.3.
psm2_mq_status2_t	MQ status structure for 96-bit (psm2_tag_t) non-blocking operations. For details, see: Section 4.3.2.4.

4.3.2.1 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.
`int32_t msg_source`	Sender's registered source ID (receive reqs only).
`void *context`	User-associated context for send or receive.

4.3.2.2 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.
`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.

4.3.2.3 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].

4.3.2.4 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).
uint32_t nbytes	Actual number of bytes transfered (receive reqs only).
psm2_error_t error_code	MQ error code for communication operation.
void * context	User-associated context for send or receive.

4.3.3 Defines

Table 9. Matched Queues Defines

Define	Description
#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.

4.3.4 Typedefs

Table 10. Matched Queues 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).

4.3.5 Functions

Table 11. Matched Queue Functions (Sheet 1 of 2)

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: Section 4.3.5.1.
`psm2_mq_finalize (psm2_mq_t mq)`	Finalize (close) an MQ handle. For details, see: Section 4.3.5.2.
`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: Section 4.3.5.3.
`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: Section 4.3.5.4.
`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: Section 4.3.5.5.
`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: Section 4.3.5.6.
`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: Section 4.3.5.7.
`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: Section 4.3.5.8.
`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: Section 4.3.5.9.
`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: Section 4.3.5.10.
`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: Section 4.3.5.11.

Table 11. Matched Queue Functions (Sheet 2 of 2)

Function	Description
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: Section 4.3.5.12.
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: Section 4.3.5.13.
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: Section 4.3.5.14.
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: Section 4.3.5.15.
psm2_mq_wait (psm2_mq_req_t *request, psm2_mq_status_t *status)	Wait until a non-blocking request completes. For details, see: Section 4.3.5.16.
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: Section 4.3.5.17.
psm2_mq_test (psm2_mq_req_t *request, psm2_mq_status_t *status)	Test if a non-blocking request is complete. For details, see: Section 4.3.5.18.
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: Section 4.3.5.19.
psm2_mq_cancel (psm2_mq_req_t *req)	Cancel a preposted request. For details, see: Section 4.3.5.20.
psm2_mq_get_stats (psm2_mq_t mq, psm2_mq_stats_t *stats)	Retrieve statistics from an instantiated MQ. For details, see: Section 4.3.5.21.

4.3.5.1 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.

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

Return values:

`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;
}
```

4.3.5.2 psm2_mq_finalize

Syntax:

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

Finalize (close) an MQ handle. The following error code is returned. Other errors are handled by the PSM2 error handler (psm2_error_register_handler).

Return values:

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

4.3.5.3 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 Section 3.1.1, “MQ Tag Matching” on page 14.

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.

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.

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

Return values:

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

4.3.5.4 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 Section 3.1.1, “MQ Tag Matching” on page 14.

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.

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

Return values:

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

4.3.5.5 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.

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

Return values:

`PSM2_OK`
The message has been successfully sent.

4.3.5.6 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`.

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

Return values:

`PSM2_OK`
The message has been successfully sent.

4.3.5.7 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`.

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

Return values:

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;
}
```

4.3.5.8 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.

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

Return values:

PSM2_OK
The message has been successfully initiated.

4.3.5.9 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 if 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.

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

Return values:

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

`PSM2_MQ_INCOMPLETE`
The iprobe is unsuccessful and status is unchanged.

4.3.5.10 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 if 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.

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

Return values:

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

PSM2_MQ_INCOMPLETE
The iprobe2 is unsuccessful and status is unchanged.

4.3.5.11 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.

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

Return values:

PSM2_OK
The improbe is successful and status is updated if non-NULL.

PSM2_MQ_INCOMPLETE
The improbe is unsuccessful and status is unchanged.

4.3.5.12 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.

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

Return values:

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

`PSM2_MQ_INCOMPLETE`
The improbe2 is unsuccessful and status is unchanged.

4.3.5.13 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).

PSM2_OK
: The function is successful and status is updated if non-NULL.

PSM2_MQ_INCOMPLETE
: The function is unsuccessful and status is unchanged.

4.3.5.14 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.

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

Return values:

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);
}
```

4.3.5.15 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.

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

Return values:

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.

4.3.5.16 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 Section 3.1.4, “MQ Progress Requirements” on page 17.

Parameters:

request
MQ non-blocking request.

status
Updated if non-NULL when request successfully completes.

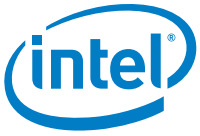
Precondition:

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.

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

Return values:

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

4.3.5.17 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 Section 3.1.4, “MQ Progress Requirements” on page 17.

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.

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

Return values:

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

4.3.5.18 psm2_mq_test

Syntax:

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

Test if 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 Section 3.1.4, “MQ Progress Requirements” on page 17.

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.

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

Return values:

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_mq_t mq, psm2_mq_req_t *allreqs, int nreqs)
{
    int i;
    void *context = NULL;

    // Ensure progress only once
    psm2_poll(mq);

    // 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;
}
```

4.3.5.19 psm2_mq_test2

Syntax:

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

Test if 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 Section 3.1.4, “MQ Progress Requirements” on page 17.

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.

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

Return values:

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.

4.3.5.20 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.

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

Return values:

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.

4.3.5.21 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.

4.4 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;
}
```

4.4.1 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.

4.4.2 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: Section 4.4.2.1.
`psm2_mq_setopt (psm2_mq_t mq, int option, const void *value)`	Set an MQ option. For details, see: Section 4.4.2.2.

4.4.2.1 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.

4.4.2.2 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`

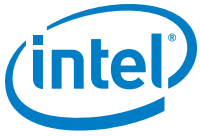
`value`
Pointer to storage that contains the value to be updated for the supplied option number. You must ensure that the pointer points to a memory location with a correct size.

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).



5.0 Intel® PSM2 Sample Program

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

5.1 Prerequisites

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

5.2 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
   ```

5.3 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;
}
```