PMX Specification
– DRAFT –

Atsushi HORI

May, 2008

Contents

List of Figures

List of Tables

PROGRESS

Table 1: Progress

Function	Composite	Ethernet	MX	Infiniband	Ethernet-HXB
OpenArgs	-	done	not yet	not yet	
OpenMember	-	done	done	done	
Reset	done	done	done	not yet (?)	
Initialize	done	done	done	done	
Start	done	done	done	done	
Stop	done	done	done	done	
Close	done	done	done	done	
GetSendBuffer	done	done	done	done	
GetSendDescInfo	done	done	done	done	
TruncateBuffer	done	done	done	done	
KeepSendDesc	done	done	done	done	
Send	done	done	done	done	
ReleaseSendDesc	done	done	done	done	
IsSendDone	done	done	done	done	
Receive	done	done	done	done	
ReleaseReceiveBuffer	done	done	done	done	
BeforeSelect	done	done	not yet	not yet	
AfterSelect	done	done	not yet	not yet	
Break	done	not yet	not yet	not yet	
Waive	done	not yet	not yet	not yet	
Continue	done	not yet	not yet	not yet	
NotCommunicating	done	done	done	not yet	
Export	not yet	not yet	not yet	not yet	
Unexport	not yet	not yet	not yet	not yet	
Write	not yet	not yet	not yet	not yet	
IsWriteDone	not yet	not yet	not yet	not yet	
Read	not yet	not yet	not yet	not yet	
IsReadDone	not yet	not yet	not yet	not yet	
Save	done	not yet	not yet	not yet	
ReopenMember	done	not yet	not yet	not yet	
Dump	done	done	done	not yet	

Preface

SCore (pronounced as [es-core]) was designed to be an operating system for clusters for high performance computation from the beginning. Thus SCore software philosophy and architecture are unique and very different from the other cluster management software or parallel programming environment for clusters.

Chapter 1

Overview

PMv2 was designed with a mind in which every software layer can be developed for PMv2 so that the hardware performance can be excluded as much as possible. Nowadays, however, it is getting difficult to develop everything from a scratch.

In PMv2, SCore-D used to manage cluster resources which are hosts, processors and network. To enable this, PM contexts are shared with SCore-D and user processes. However, this context sharing was the headache to implment PM on top of existing communication library, such as Myricom MX and the Verb layer of Infiniband. So the context sharing is NOT required to implment PM library in PMX.

This does not imply that SCore with PMX will have less functionalities, such as gang-scjeduling and checkpoint/restart of the ones with PMv2. And this is the most challenge of designing PMX.

Third Party Communication Layer In PMv2, everything is developed by SCore development team. However, it is getting difficult to develop everything. So PMX is designed to utilize those third party communication layer.

Multi Core Since the multi-core cluster (used to be called SMP cluster) is getting popular nowadays. In PMv2, the multi-core was in their minds but they did not expect to have more than 4 cores in a host. Although there is some argument to the number of cores in a host, but power problem of silicon is strongly pushing the multi-core technology.

MPI2 Dynamic process creation problem.

Here in this document, PMX, a new API design for PMv2, is proposed so that the above two major problems are to be solved.

1.1 Changes

- PMX functions are prefixed by “pmx” instead of “pm” in PMv2.
- Function-level compatibility (user only) is preserved
- Defining PM basic types
- The size of PM context can be dynamic.
- Remove the privilege abstraction and operations
- Changing the sharing semantics between user process and SCore-Dprocess

- Merging migration functions with checkpoint functions
- Making global clock functions obsolete
- Remove `pmGetFd()` function
- PM/Composite is a specil PM device in PMX.

1.2 Terminology

Context

A *PM context* is an end-point derived from a PM device. A PM context is an end-point. The PM contexts which are bound with the same *key* number can communicate with the others. The order of messages sent from a PM context to another PM context must be preserved at the PM context receiving.

Key

A *PM key* is a binding of PM contexts. A PM context can only communicate with the other PM contexts bound with the same PM key value.

Composite Context and Member Context

Composite is a special PM device which is not an actual network device but a pseudo device having routing table. In PMX, PM/Composite is merged with PM/Shmem which is a shared memory device for intra-host (inter-process) communication. Member context is the context which is designed to be a member of the Composite context. User program must always allocate at least one PM/Composite context and actual communication is done by member context hold by the PM/Composite context.

1.3 Compatibility with PMv2

1.4 PM Features

- PMX is an API supporting multiple protocols (multiple network devices). The multiple protocol support is done by PM/Composite, a pseudo PM device, and actual PM device need not to take care about the multiple protocol handling.
- PMX supports gang scheduling, checkpoint/restart and migration.
- A PM device can be implemented as user-level or kernel-level communication.
- PMX supports heterogeneous processors (byte orders)

1.5 Common Rules

No Blocking Any PMX function must not block.

1.6 Composite Context and Member Context

There is a special PM device, called PM/Composite. In PMv2, PM/Composite is essentially a routing table indexed by destination node number. In PMX, PM/Composite has the same routing table, however, it is merged with PM/Shmem. It is assumed that PM/Composite is the PM device which is seen by user programs directly, it has some common features to all other PM devices, such as the compatibility functions with PMv2 and locking for multi-threaded programming. This will be explained in Subsection 1.6.1.

When to try to send a message, PM/Composite finds an table entry corresponding to the destination node, and the entry holds actual PM device and the sending request is forwarded to the PM device in the entry (Figure 1.1). When to receive a message, PM/Composite try to find a received message among the PM devices registered to PM/Composite device.

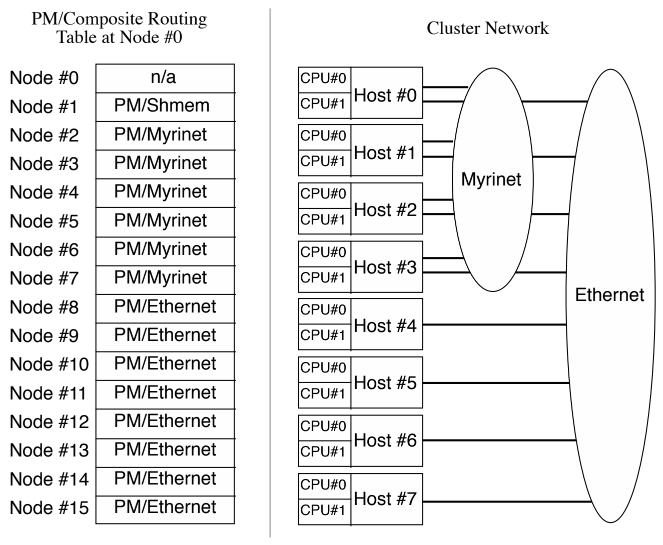

Figure 1.1: Basic Idea of PM/Composite

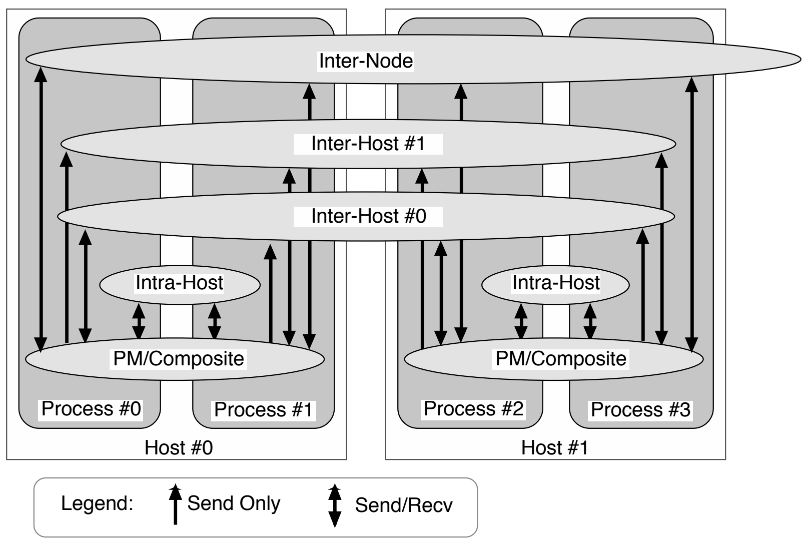

Figure 1.2: Example of Composite PM Contexts

1.6.1 Common Fetures

PM/Composite in PMX has common featrues to the other PM devices so that the other PM device does not have to care about.

Argument Check The passed arguments to the PMXfunctions are checked as far as PM/Composite can do.

State Transition PM context has a state and the transition of the state must be obeyed by the rule described in Section 1.7. Everytime a program wants to change the state of PM/Composite context and its member contexts, PM/Composite checks if the state transition is legal or not. Thus the other PM contexts do not have to check it.

Compatibility with PMv2 PM/Composite supports PMv2 functions, `pmGetMtu()`, `pmGetSelf()`, `pmGetSendBuffer()`, `pmTruncatreBuffer()`, `pmSend()`, `pmReceive()` and `pmReleaseReceiveBuffer()` functions for compatibility.

1.7 State Transition

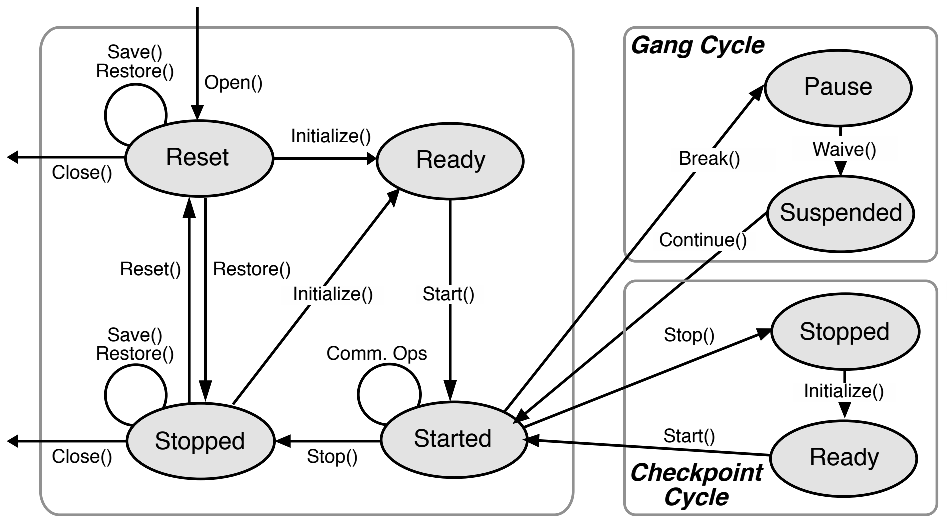

Figure 1.3: State Transition of a PM context

1.8 Buffer Descriptor

The buffer descriptor is newly introduced feature of PMX, so that the lock granularity can be minimized and PM NFO (Network Fail Over) device can be implemented safely.

The descriptor is created and retuned a descriptor by calling the `pmxGetSendBuffer()`, `pmxReceive()`, `pmxRead()` and `pmxWrite()` functions. Further, it allows to check if sending or

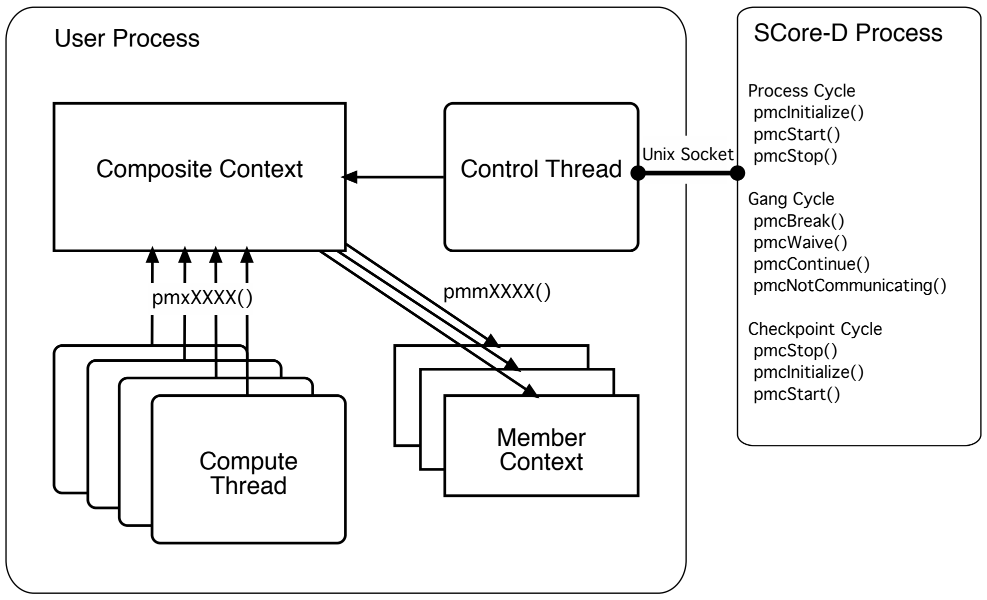

Figure 1.4: Pthread and Control Port

receiving is succeeded. This message level status check is needed to implement NFO (Network Fail Over), where messages which are failed to send must be re-send by using the other PM context(s) from the different process possibly.

When communicating processes are checkpointed and restarted, calling the `pmReopen()` function guarantees to have the same descriptor for the outstanding messages.

This uniqueness of the descriptor must be guaranteed with the descriptors which are created by calling the same PMX function. For example, the descriptor created by the `pmxGetSendBuffer()` function may happen to return the same descriptor created by calling the `pmxReceive()` function, but they are strictly distinguished in the program.

Table 1.1: Lifetime of Buffer Descriptor

	Creation	Destruction	Note
Two-Sided Receive	`pmxReceive()`	`pmxReleaseReceiveBuffer()`	
Two-Sided Send	`pmxGetSendBuffer()`	`pmxSend()`	
Two-Sided Send	`pmxGetSendBuffer()`	`pmxReleaseSendDesc()`	when `pmxKeepSendDesc()` is c
One-sided Read	`pmxRead()`	`pmxIsReadDone()`	
One-sided Write	`pmxWrite()`	`pmxIsWriteDone()`	

Implementation Note on Buffer Descriptor

The least-significant `PMX_DESC_SHIFT` bits of the descriptor value are reserved for PM/Composite.

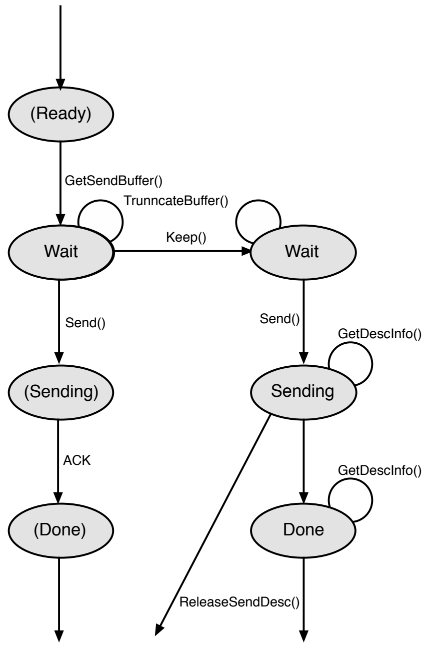

Figure 1.5: State Transition of Send Descriptor

1.9 Gang Scheduling and Checkpointing

The `pmxBreak()`, `pmxWaive()` and `pmxContinue()` operations must be designed so that the operation can affect all the send and receive threads. This is because of that those operation functions are called by SCore-D targetting the PM contexts allocated for user process(es).

The function call cycle of `pmxBreak()`, `pmxWaive()` and `pmxContinue()` for gang scheduling must be transparent to target user process(es). Conversely speaking, those function can be called in the middle of communication in the user process. Lost of messages or receiving the same message twice or more should not happen, during the preemption cycle.

The same rule must be applied to the checkpoint and restart. When a checkpoint is to be taken, the function call cycle of `pmxStop()`, `pmxSave()`, `pmxInitialize()`, and `pmxStart()` will take place. When restarting the checkpointed process, the function call sequence of `pmxReopen()`, `pmxInitialize()` and `pmxStart()` will take place. Unlike the above gang scheduling case, those functions are called in the user processes.

If the implementation of those operations is impossible due to the limitation of underlying communication library, then those operation should not be implemented.

1.10 Error Reporting

All PMX functions return an integer error value and return `PM_SUCCESS` when they succeed.

PM communication is asynchrounous and there is no error reporting variable. When an error happens on any asynchrounous communication operation, one-sided and two-sided, the error can be reported as the return value of any PM functions, whenever the implementation detects the error on the same shared context. Thus, for example, the `pmxSend()` function may report an error which is caused by the previous `pmxSend()` function call.

In the case where multiple errors happen before the chance to report the error. In this case, it is up to the implementation to report which error.

Implementation Note on Error Reporting

When a communication error happens, there is no way to recover nor proceed the parallel execution in most cases. And somtimes it is very difficult to specify which function call caused the error. Thus this "lazy" error reporting will not cause any problem and can reduce the overhead caused by "strict" error reporting.

Chapter 2

Composite Context

2.1 Composite Context Creation

2.1.1 pmxOpenContext()

```
   int pmxOpen( pm_key_t key, pm_flags_t flags, pmNode nodes[],
                pm_node_t nnodes, pm_node_t nodeno,
                pmContext *pmcp, int *fdp )
```

The `pmxOpen()` function creates a PM/Composite context.

When the `fdp` argument is set to `NULL`, then the `pmxOpen()` function will not create a pthread to control the created context by a supervisor process (SCore-D). Otherwise a Unix socket and a pthread are created, and the socket is to be passed to the supervisor process to control the context ouside of the process calling the `pmxOpen()` function. As the result of having the control port, the `pmxInitialize()`, `pmxStart()`, PFUNCpmxStop, `pmxBreak()`, `pmxWaive()` and `pmxContinue()` functions are unable to be called from the process calling this function so that the context is to be controlled by the supervisor process (see also 2.2).

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EIO`	Internal error
others	The `mmap()` function may return with.

```
typedef struct pm_node {
  char          *hostname;
  pm_node_t     procno;
} pmNode;
```

Figure 2.1: `pmNode` Definition

2.1.2 pmxAddMember()

```
   int pmxAddMember( pmContext *pmc, int argc, char **argv, pmNode nodes[] );
```

The `pmxAddMember()` function creates a member context and add the created member context to the composite context specified with the `pmc` argument. The member context is created by calling the `pmmOpenMember()` member function and the created member context is returned to the `pmp` argument and registered to the composite context.

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`ERANGE`	Node number in the `tab` is out of range
`EBADF`	File descriptor in bad state
`EIO`	Internal error
others	The `mmap()` function may return with.

2.1.3 `pmxCompositeGetConfig()`

```
int pmxCompositeGetConfig( pmCompositeContext *pcc,
                           pm_key_t *keyp,
                           pm_flags_t *flagsp,
                           pm_node_t *nnodesp,
                           pm_node_t *nprocsp,
                           pm_node_t *procnop )
```

Return Values

`PM_SUCCESS`	Succeeded

2.1.4 `pmxCompositeAddRoute()`

```
int pmxCompositeAddRoute( pmCompositeContext *pcc,
                          pm_node_t ndc,
                          pm_node_t ndm,
                          pmMemberContext *pmm,
                          pm_size_t tsmtu,
                          pm_size_t osmtu )
```

Return Values

`PM_SUCCESS`	Succeeded
`ERANGE`	Succeeded
`EBUSY`	Succeeded

2.2 Control Port

```
int    pmcInitialize(int fd);
int    pmcStart(int fd);
int    pmcStop(int fd);
int    pmcClose(int fd);
int    pmcBreak(int fd);
int    pmcWaive(int fd);
int    pmcContinue(int fd);
```

2.3 Composite Context Operations

2.3.1 pmxGetMtu()

```
int pmxGetMtu( pmContext *pmc, pm_node_t node, pm_size_t *mtu_twoside,
pm_size_t *mtu_oneside );
```

The `pmxGetMtu()` function returns the upper limit of the send messages size. The `node` can be `PM_NODE_ANY` and returns the same number as the `mtu_twoside` member variable in the `pmAttribute` structure obtained by the `pmxGetAttribute()` function.

The `mtu_twoside` and/or the `mtu_oneside` variable can be set to `NULL`.

Return Values

PM_SUCCESS	Succeeded
EINVAL	Illegal arguments
EIO	Internal error

2.3.2 pmxGetAttribute()

The `pmxGetAttribute()` returns the attributes of the specified PM context.

```
int pmxGetAttribute( pmContext *pmc, pmtAttribute *attrp );
```

Return Values

PM_SUCCESS	Succeeded
EIO	Internal error

2.3.3 pmxReset()

```
int pmxReset( pmContext *pmc );
```

Reset the PM context. If there are some messages in the receive and/or send buffer, those messages will be lost.

Return Values

`PM_SUCCESS`	Succeeded
`EIO`	Internal error

2.3.4 `pmxInitialize()` and `pmxStart()`

```
int pmxInitialize( pmContext *pmc, pm_flags_t flag );
int pmxStart( pmContext *pmc );
```

The `pmxInitialize()` and `pmxStart()` functions creates and starts the receive and send threads. Those two function must be called in a process in which actual PM communication takes place. If a contex is shared between processes to communicate, eash process must call those functions.

The end of calling the `pmxInitialize()` must be barrier-synchronized. The nodes specified in the `pmxOpenContext()` *MAY* not call those functions at the start up. This is because the node is allocated but there is no process at the time, but the process *MAY* be invoked in the future.

Table 2.1: `pmxInitialize()` Option Bits

Symbol	Note
`PM_OPT_BLOCKING_RECV`	Enabling blocking receive
`PM_OPT_NO_RECV`	Disbaling receiving

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Device already opened
`EPROTO`	Illegal state transition
`ENOLINK`	Device has no link
`EHOSTUNREACH`	there is an unreachable node
`EIO`	Internal error

2.3.5 `pmxStop()`

```
int pmxStop( pmContext *pmc );
```

Stop and destroy receive and send threads corresponding to the calling process. Those threads can be restarted or recreated by calling the `pmxInitialize()` and `pmxStart()` functions.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Device already opened
`EPROTO`	Illegal state transition
`EAGAIN`	There is at least one thread communicating
`EIO`	Internal error

2.3.6 pmxBreak() and pmxWaive()

```
int pmxBreak( pmContext pmc );
int pmxWaive( pmContext pmc );
```

Stop receive and send threads for network preemption (gang scheduling). The end of calling the `pmxBreak()` must be barrier-synchronized.

Return Values

PM_SUCCESS	Succeeded
EBUSY	Device already opened
EPROTO	Illegal state transition
EAGAIN	There is at least one thread communicating
EIO	Internal error

2.3.7 pmxContinue()

```
int pmxContinue( pmContext pmc );
```

Resume the suspended receive and send threads which are suspended by calling the `pmxBreak()` and `pmxWaive()` functions.

Return Values

PM_SUCCESS	Succeeded
EBUSY	Device already opened
EPROTO	Illegal state transition
EIO	Internal error

2.3.8 pmxClose()

```
int pmxClose( pmContext *pmc );
```

The `pmxClose()` function closes the PM context which was created by calling the `pmxOpenContext()`. After returning this function, the specified PM context will be invalid to access.

Return Values

PM_SUCCESS	Succeeded
EBUSY	Device already opened
EPROTO	Illegal state transition
EIO	Internal error

2.3.9 pmxNotCommunicating()

The `pmxNotCommunicating()` function returns `PM_SUCCESS` if there is no outstanding two-sided and one-sided communication. Otherwise it returns `EBUSY`.

```
int pmxNotCommunicating( pmContext *pmc );
```

The `pmxNotCommunicating()` function must be designed and implmented so that the function can be called on the PM context which is not called the `pmxInitialize()` and the `pmxStart()` functions, but those functions are already called by the process which is different from the process calling the `pmxNotCommunicating()` function.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	There are some outstanding communications

2.4 Two-Sided Communication

Sending a Message

2.4.1 `pmxGetSendBuffer()`

To send a message, the `pmxGetSendBuffer()` function is called to allocate a send buffer.

```
int pmxGetSendBuffer( pmContext *pmc, pm_node_t node, void **bufp,
                      pm_size_t length, pm_desc_t *descp );
```

It returns the address of the allocated buffer and the send buffer descriptor. The `pmxGetSendBuffer()` function can be called by different processes or threads on the same shared PM context.

The returned buffer address is aligned to machine dependent address so that the buffer can be casted to any data type.

The send buffer information can be obtained by calling the `pmxGetSendDescInfo()` function on the send buffer descriptor returned by the `pmxGetSendBuffer()` function.

It is not allowed to send a message to the node itself, nor to send a message having zero length.

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. Therefor the `pmxGetSendbuffer()` should have a lock during its execution. And when the context is already locked, the `pmxGetSendBuffer()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`ENOBUFS`	No buffer space available
`EBUSY`	Already locked
`EINVAL`	Illegal arguments
`EPROTO`	Illegal state transition
`EMSGSIZE`	Message too long or zero message size
`ERANGE`	Illegal node number
`EIO`	Internal error

```
/*
 * PM Descriptor Info.
 */
typedef struct pm_send_desc_info {
        int             status;         /* status */
        pm_addr_t       addr;           /* buffer address */
        pm_size_t       length;         /* message length */
} pmSendDescInfo;
```

Figure 2.2: `pmSendDescInfo` Definition

2.4.2 `pmxGetSendDescInfo()`

```
   int pmxGetSendDescInfo( pmContext *pmc, pm_desc_t desc,
                           pmSendDescInfo *infop );
```

And the `pmSendDescInfo` structure is defined as above. If the corresponding message is already sent or not found in the PM context, then the `pmxGetSendDescInfo()` function returns `EBADR`.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EIO`	Internal error

2.4.3 `pmxTruncateBuffer()`

```
   int pmxTruncateBuffer( pmContext *pmc, pm_desc_t desc,
                          pm_size_t length );
```

The `pmxTruncateBuffer()` function can shrinks the send buffer length to the specified length. The `pmxTruncateBuffer()` function can be called any number of times in the execution between the call of the `pmxGetSendBuffer()` function and the call of the `pmxSend()` function, and shrinked buffer can be expanded up to the length which is specified by the `pmxGetSendBuffer()` function. Setting the `length` of zero results in returning an error (`EINVAL`).

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.4.4 `pmxKeepSendDesc()`

```
   int pmxKeepSendDesc( pmContext *pmc, pm_desc_t desc );
```

The `pmxReleaseSendDesc()` function destroys the send descriptor which is kept by calling the `pmSend()` function.

If the `pmxKeepSendDesc()` function is called, then the buffer descriptor is kept and the corresponding buffer region must *not* be reclaimed, until the descriptor is released with calling the `pmxReleaseSendDesc()` functoion.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EIO`	Internal error

2.4.5 `pmxSend()`

Once the content of the send message is fixed, then the `pmxSend()` function is called to send the message. After calling the `pmxSend()` function, the modification on the content of the allocated send buffer may *not* be refelected to the received message.

```
int pmxSend( pmContext *pmc, pm_desc_t desc );
```

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. In some implementation, a lock during the `pmxSend()` execution may be required. And when the context is already locked, the `pmxSend()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.4.6 `pmxReleaseSendDesc()`

```
int pmxReleaseSendDesc( pmContext *pmc, pm_desc_t desc );
```

The `pmxReleaseSendDesc()` function destroys the send descriptor which is kept by calling the `pmSend()` function.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.4.7 pmxIsSendDone()

```
int pmxIsSendDone( pmContext *pmc );
```

Return Values

PM_SUCCESS	Succeeded
EBUSY	Not yet
EIO	Internal error

2.4.8 Order of Sending Messages

In the program shown in Figure 2.3, the messages will be sent in the order of message A, B and C. This order is the same order of calling the `pmxGetSendBuffer()` function.

```
pmxGetSendBuffer( pmc, dest, &mess_a, &desc_a );
pmxGetSendBuffer( pmc, dest, &mess_b, &desc_b );
pmxGetSendBuffer( pmc, dest, &mess_c, &desc_c );
...
pmxSend( pmc, &desc_b, 1 );
pmxSend( pmc, &desc_c, 1 );
pmxSend( pmc, &desc_a, 1 );
```

Figure 2.3: Order of Sending Messages

Receiving a Message

2.4.9 pmxReceive()

The `pmxReceive()` function tries to get a received message.

```
int pmxReceive( pmContext *pmc, void **bufp, pm_size_t *sizep,
                pm_desc_t *descp );
```

The `pmxReceive()` function returns the address of the received message and the receive buffer descriptor, if there is a received message.

The returned buffer address is aligned to machine dependent address so that the buffer can be casted to any data type.

Return Values

PM_SUCCESS	Succeeded
EINVAL	Illegal arguments
EBUSY	Already locked
EPROTO	Illegal state transition
ENOBUFS	No message to receive available
ESHUTDOWN	Receiving is disabled
EIO	Internal error

2.4.10 pmxReleaseReceiveBuffer()

```
int pmxReleaseReceiveBuffer( pmContext *pmc, pm_desc_t desc );
```

The descriptor is released and corresponding buffer region is reclaimed when the `pmxReleaseReceiveBuffer()` is called. There is no way and no need of getting information from the receive buffer descriptor.

Return Values

PM_SUCCESS	Succeeded
EBUSY	Already locked
EPROTO	Illegal state transition
EBADR	No such descriptor
EIO	Internal error

Blocking Receive

2.4.11 pmxBeforeSelect()

```
int pmxBeforeSelect( pmContext *pmc, fd_set *fds, int *maxfdp,
                              sigset_t *sigmask );
```

The `pmxBeforeSelect()` function tells PM context that user program is going to block to wait for incoming messages. It returns the file descriptor(s) in `fds`, the maximum number of the file descriptor, and the signal mask of Linux so that those variables can be passed to the `pselect()` function. There can be the case where user sets those variables and the `pmxBeforeSelect()` function never resets them.

The `pmxBeforeSelect()` function is designed so that the wait can be implemented with using the Linux `pselect()` function. The `fds` and the `maxfdp` variables are set by the `pmxBeforeSelect()` function so that they can be passed to the Linux `pselect()` function. It should be noticed that the `pmxBeforeSelect()` function does not always return with the file descriptor(s) to be `pselect()`ed. In this case, the `maxfdp` will be returned unchanged and the `fds` and `fdmaxp` may not be suitable for passing the `pselect()` function.

Also the `sigmask` can be passed so that some specific signal(s) can unblock `pselect()` function. To avoid the race condition, if a Linux signal is used to unblock the waiting, then the signal must be blocked in the `pmxBeforeSelect()` function, and can be unblocked in the `pmxAfterSelect()` function.

The `pmxBeforeSelect()` and the `pmxAfterSelect()` functions must be called in pair always. Since it may be very difficult to avoid the race condition in some implementations, the `pmxReceive()` function must be called before calling the `pselect()` function so that the messages received in prior to the call of the `pmxBeforeSelect()` function are extracted from the receive buffer. The `pmxBeforeSelect()` function call never guarantees the presence of received message(s) when the `pselect()` function returns with a positive integer larger than zero. User program must be programmed to handle the case where the `pselect()` function tells there seems to be some received messages, but actually there is no received messages. In addition, the `pselect()` may be blocked even if there are some messages which can be be received. Thus it is not recommended to set the infinite or very larger timeout duration (by setting `NULL`of the timeout parameter).

The returned set of the file descriptors *MAY* be changed while the execution of a parallel process, especially when the parallel process spawns (adds) the other processes or some processes in the parallel process terminates. Thus the set of file descriptors must be obtained by the function every time `pselect()` is called.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.4.12 `pmxAfterSelect()`

```
int pmxAfterSelect( pmContext *pmc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.4.13 Order of Received Messages

In the program shown in Figure 2.4, the messages will be sent in the order of message A, B and C. This order is the same order of calling the `pmxReceive()`.

```
pmxReceive( pmc, &mess_a, &size_a, &desc_a );
pmxReceive( pmc, &mess_b, &size_b, &desc_b );
pmxReceive( pmc, &mess_c, &size_c, &desc_c );
...
pmxReleaseReceiveBuffer( pmc, desc_b );
pmxReleaseReceiveBuffer( pmc, desc_c );
pmxReleaseReceiveBuffer( pmc, desc_a );
```

Figure 2.4: Order of Received Messages

2.4.14 Blocking Receive Example

Deadlock Detection

In the SCore-D deadlock detection, checking the existence of outstanding messages which are in the buffers of a PM context and are not yet received by a user process nor sent to the other process, is the key. Figure 2.6 shows the possible problem on counting the outstanding messages. On the node K SCore-D samples a PM context but it can not found message in the receive buffer, and on the node L SCore-D can not find any message in the send buffer, however, a message actually exists. This can not happen on PMv2 because any actual message sending is prohibited at the time of sampling.

This problem can be avoided by doing the check twice. When the first check fails to find some outstanding messages, then do the check again. If the both checks tell the absence of the outstanding message, then there is possibility of deadlock.

```
#include <sys/select.h>
#include <signal.h>
#include <errno.h>
#include <pm.h>

int blocking_receive( pmContext *pmc, void **bufp, pm_size_t *lenp,
                      pm_desc_t *descp ) {
  struct timespec timeout;
  sigset_t sigmask;
  fd_set fds;
  int fd_max;
  int cc0, cc1;

  FD_ZERO( &fds );
  fd_max = 0;   // must be set to zero everytime this function is called
  if( sigprocmask( 0, NULL, &sigmask ) < 0 ) {  // get current signal mask
    return( errno );
  }
  cc0 = pmxBeforeSelect( pmc, &fds, &fd_max, &sigmask );
  if( cc0 != PM_SUCCESS ) return( cc0 );
  while( 1 ) {
    //  Race: an interrupt or a signal may be delivered here.
    //          When it happens, pselect() returns immediately
    //          but the coresponding message is already pmxReceive()d.
    if( ( cc1 = pmxReceive( pmc, bufp, lenp, descp ) ) == PM_SUCCESS ) break;
    if( cc1 == EBUSY ) continue;
    if( cc1 != ENOBUFS ) return( cc1 ); // Error !!
    // The return value of pselect() may NOT be accurate
    // because of the race condition above.
    if( fd_max == 0 ) break;
    timeout.tv_sec  = 0;
    timeout.tv_nsec = 10 * 1000 * 1000; // 10 msec
    (void) pselect( fd_max, &fds, NULL, NULL, &timeout, &sigmask );
  }
  if( ( cc0 = pmxAfterSelect( pmc ) ) != PM_SUCCESS ) return( cc0 );
  return( cc1 );
}
```

Figure 2.5: `blocking_receive()`

2.5 One-Sided Communication

In PMX, there is no implicit rule on the order of inidividual one-sided communications. There is no rule on the order of two-sided communications and one-sided communications.

Figure 2.7 shows the normal one-sided communication scheme. Most of one-sided communication requires the pindown of a memory segment so that the physical memory address of the segment would not be changed. It is implementation-dependent that how the memory segment is locked and how the locked segment is expressed in its API. Since PMX is supporting multiple protocols (multiple underlying communication libraries), the low-level implementation should be hide fromthe PMX API.

The `pmxExport()` function locks the specified memory segment by calling an appropriate function of low-level library, and the segment ID is sent to the node on which the segment ID will be used for some one-sided communication. The `pmxExport()` function returns a handle of the segment. This handle can be used for the one-sided communication of PMX. The message to send the low-levl segment ID is hidden from user program (dashed arrrow in Figure 2.7, however, `pmxReceive()` function should be called to receive the hanlde so that the low-level can handle the segment ID properly (Figure 2.7).

A created memory handle can be exported to the other nodes any time. It is not allowed

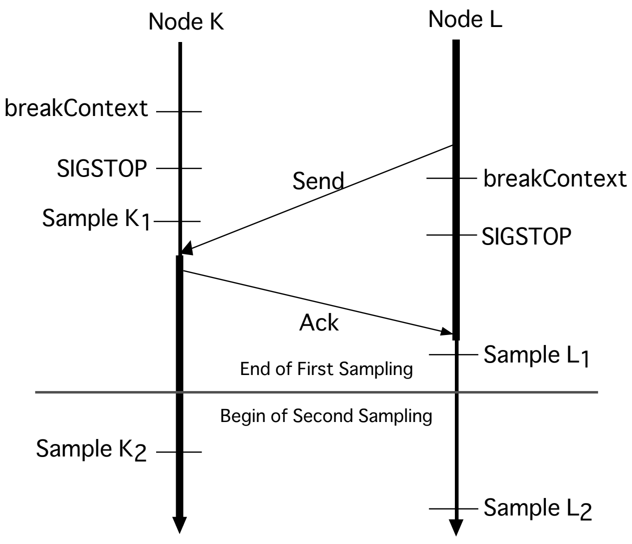

Figure 2.6: Counting Outstanding Messages

Table 2.2: PM One-Sided Communication Operations

Function Name	PMv2
`pmxIsReadDone()`	Changed
`pmxIsWriteDone()`	Changed
`pmxExport()`	`pmMlock()`
`pmxUnexport()`	`pmMunlock()`
`pmxRead()`	Changed
`pmxWrite()`	Changed

to forward the imported handle to another node. The `pmxUnexport()` function destruct the handle and exported handle on the othet node(s) is also annuled. Only the node which creates and exports the handle can un-export the handle.

The PM shared context should have a table which holds the segment ID of low-level communication library and the handle of PMX. However, the ID of the communication library might be different from the one after the process is restarted from a checkpoint. Thus, it is required to reconstruct the segment ID table by calling the `pmxExport()` function to the every entry in the table when the program is restarted from a checkpoint while in the process of the `pmxStart()` function.

In the near future it is expected that the memory segment locking will not be required by having the micro-TLB on NIC hardware or similar mechanism. In this case, the `pmxExport()`

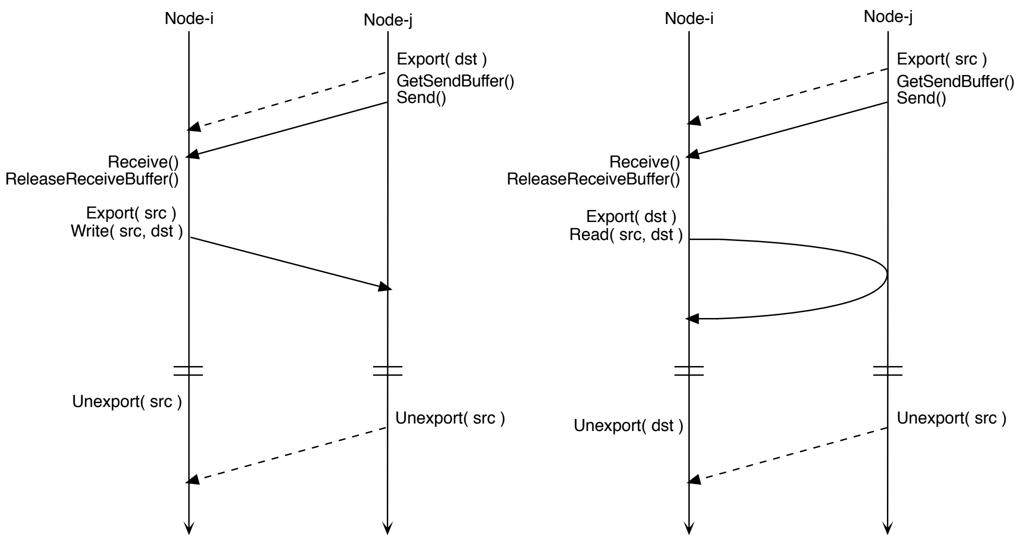

Figure 2.7: One-Sided Communication

```
typedef struct pm_address_handle {
  pm_node_t     node;           /* network byte order */
  pm_pid_t      pid;            /* network byte order */
  pm_addr_t     address;        /* network byte order */
  pm_off_t      length;         /* network byte order */
} pmAddrHandle;
```

Figure 2.8: `pmDeviceOps` Definition in `pm.h`

function will simply creates and returns the remote memory handle.

Implementation Note on Exit of Process

A process may exit at any time. Thus `pmxRead()` and `pmxWrite()` would be carefully designed so that accessing memory pages which were occupied by the exited process would never happen.

Issues on Order of One-Sided Communication

Strict order or relaxed order ?

2.5.1 `pmxExport()`

The `pmxExport()` function allows remote `node` to access the specified local memory region with the one-sided communication way. The `node` value can be `PM_NODE_ANY`.

Table 2.3: Problematic One-Sided Communication

	Node A	Node B	Node C
1	export a segment to Node B		
2	send the ID to Node C		
3			forward the ID to Node B
4		receive the ID	
5		export to another segment	
6		issue one-sided communication	

```
int pmxExport( pmContext *pmc, pm_addr_t addr,
               pm_size_t length, pm_node_t node, pmAddrHandle *handle );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.5.2 pmxUnexport()

The `pmxUnexport()` function destroys the memory handle which is created on the same host.

```
int pmxUnexport( pmContext *pmc, pmAddrHandle *handle );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.5.3 pmxRead()

```
int pmxRead( pmContext *pmc, pmAddrHandle src, pm_off_t soff,
             pmAddrHandle dest, pm_off_t doff,
             pm_size_t length, pm_desc_t *descp );
```

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. In some implementation, a lock during the `pmxRead()` execution may be required. And when the context is already locked, the `pmxRead()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EREMOTE`	Destination is remote
`EFAULT`	Bad address handle
`EIO`	Internal error

2.5.4 `pmxWrite()`

```
int pmxWrite( pmContext *pmc, pmAddrHandle src, pm_off_t soff,
               pmAddrHandle dest, pm_off_t doff,
               pm_size_t length, pm_desc_t *descp );
```

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. In some implementation, a lock during the `pmxWrite()` execution may be required. And when the context is already locked, the `pmxWrite()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EREMOTE`	Source is remote
`EFAULT`	Bad address handle
`EIO`	Internal error

2.5.5 `pmxIsReadDone()`

The `pmxIsReadDone()` function checks is the posted one-sided write operations succeeded or not. If they are succeeded, the function retunrs `PM_SUCCESS` otherwise returns `EBUSY`.

```
int pmxIsReadDone( pmContext *pmc, pm_desc_t desc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not yet finished
`EIO`	Internal error

2.5.6 `pmxIsWriteDone()`

The `pmxIsWriteDone()` function checks is the posted one-sided write operations succeeded or not. If they are succeeded, the function retunrs `PM_SUCCESS` otherwise returns `EBUSY`.

```
int pmxIsWriteDone( pmContext *pmc, pm_desc_t desc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not yet finished
`EIO`	Internal error

2.6 Checkpoint Support

Table 2.4: PM Checkpoint Operations

Function Name	PMv2
`pmxSave()`	Changed
`pmxReopen()`	Changed
`pmCheckpoint()`	Obsolete
`pmRestartSys()`	Obsolete
`pmRestartUser()`	Obsolete
`pmMigrateSys()`	Obsolete
`pmMigrateUser()`	Obsolete

Implementation Note on Hostnames

As shown in the Figure 1.3, the context has neither send nor receive thread at the time of calling the `pmxSave()` function. Those thread are re-created at the time of restating the process. Conversely speaking, there should be enough information to re-create those threads at restarting in the saved information created by the `pmxInitialize()` function.
It is the SCore-D's responsibility to have the same PM context having the same context number and channel number to be re-allocated and the same number of nodes and the same number of hosts to be re-allocated when the checkpointed process is restarted. It should be minded that there is no guarantee to re-allocate the same set of hosts. This could happen when the process is migrated.
Thus it is not a good idea to rely on hostnames to identify hosts (or nodes). Or the information depending on hostnames in the saved context must be compensated when the context is restored by calling the `pmxReopen()` function.

2.6.1 `pmxSave()`

```
int pmxSave( pmContext *pmc, void **savep );
```

If the context is located on a mmap segment which is shared with the other process(es), then the shared mmap region will not be restored at restart. To avoid this, the `pmxSave()` function copies the content of the shared region to an allocated memory area which can be checkpointed and restarted. After calling this function, the `savep` points the address of the (possibly malloc()ed) memory region which has enough information to recreate the context.

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EBUSY`	Not ready to save context
`EPROTO`	Illegal state transition
`ENOBUFS`	No buffer space available
`EIO`	Internal error

2.6.2 `pmxReopen()`

```
int pmxReopen( pmContext *pmc, pmNode nodes[], void *save );
```

Recreate context based on the information pointed by the `save` argument. It must be guaranteed that it must be recreated having the same number of nodes and numer of processes in a host. However, the `nodes` may vary from the one when the context was created and saved. This can happen when the process is migrated from a host to another.

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EBUSY`	Not ready to restore context
`EPROTO`	Illegal state transition
`EIO`	Internal error

Implementation Note on Soundness

There can be the case in which the saved context information *MAY* be changed accidentally or maliciously and the `pmxReopen()` fails and the process to restart may also fails as the result. However, the Linux kernel or the other process which shares the restored context should not be affected when this happens. It is the implementer's responsibility to check the content of the saved context.

2.7 Network Preemption Support

Table 2.5: PM Preemption Operations

Function Name	PMv2
`pmxBreak()`	`pmControlSend()`
`pmxWaive()`	New
`pmxContinue()`	New

2.7.1 pmxBreak()

```
int pmxBreak( pmContext *pmc );
```

The `pmxBreak()` function inhibits the state transitions of the messages in the send buffer(s) of the PM context. In addition to this, in some implementation, the function must be prepared for the following call of the `pmxWaive()` function so that the network can be preempted.

At the result of the freezing the states of sending messages, the `pmxGetSendBuffer()` and `pmSend()` functions must return `EBUSY` without any action.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not ready
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.7.2 pmxWaive()

```
int pmxWaive( pmContext *pmc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not ready
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.7.3 pmxContinue()

```
int pmxContinue( pmContext *pmc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not ready
`EPROTO`	Illegal state transition
`EIO`	Internal error

2.8 Debug

Table 2.6: PM Debug Supports

Function Name	PMv2
`pmxDump()`	

```
int pmxDump( pmContext *pmc, FILE *file );
```

Chapter 3

Member Context

3.1 SCore KVS

3.1.1 Environment Query

```
#include <score_kvs.h>
int score_environment( void );
```

The `score_environment()` function returns the SCore environment in which the program is running. The return value of `SCORE_LOCAL_ENVIRONMENT` means the program is running on local host. The value of `SCORE_SCOUT_ENVIRONMENT` means that the program is running on the SCOUT environment. The `SCORE_SCORED_ENVIRONMENT` and the `SCORE_SCRDMAN_ENVIRONMENT` values means the program is running under SCore-D.

3.1.2 Initilization

```
#include <score_kvs.h>
int scorekvs_initialize( void );

int score_self_proc;
int score_self_host;
int score_self_node;
int score_num_proc;
int score_num_host;
int score_num_node;
```

`scorekvs_initialize()` function must be called before calling any other SCore KVS functions (except `score_environment()`). When the `scorekvs_initialize()` function call succeeds, then the `score_self_proc`, `score_self_host`, `score_self_node`, `score_num_proc`, `score_num_host`, and `score_num_node` variables are set.

Return Values

`0`	Succeeded
`ENOMEM`	Not enough memory
others	Depending on environment

3.1.3 Scoreboard Information

```
#include <score_kvs.h>
char *scoreboard_get_value( char* name, char* attr );
```

No matter which SCore environment where the program is running, the content of SCore database (`scorehosts.db`) is put into the SCore KVS system and can be retrived by calling the `scorekvs_get_value()` function.

```
host0.score.net  core=2  speed=2200  network=ethernet
```

Figure 3.1: Example of SCore database description

```
#include <score_kvs.h>

char *core;
core = scoreboard_get_value( ȟost0.score.net'', čore'' );
```

Figure 3.2: `scoreboard_get_value()` function usage

If the database file has the record shown in Figure 3.1, then code fragment shown in Figure 3.2 is executed, then the variable `core` will hold the character string of "2."

The returned string of the `scoreboard_get_value()` function is allocated by calling the `malloc()` function. If there is no record nor attribute name specified in the argument, then the function returns `NULL`.

3.1.4 KVS

```
#include <score_kvs.h>
int scorekvs_put( char* key, char *value );
int scorekvs_get( char *key, char **valp )
int score_barrier( void );
```

The `scorekvs_put()` function puts the tuple of key and value pair to the SCore KVS. the `score_barrier()` function does the barrier synchronization and commits the put tuples so that the tuples are accesible from all nodes. When the `score_barrier()` function call succeeds, the tuples put before the barrier synchronization are accessible by calling the `scorekvs_get()` function. The `scorekvs_get()` function returns the value string which is allocated by the `malloc()` function.

Return Values

`0`	Succeeded
`ENOMEM`	Not enough memory
others	Depending on environment

3.1.5 Scoreboard Databasae Access

```
#include <score_kvs.h>
int scoreboard_get_value( char *key, char *attr, char **valp )
```

The content of scoreboard.db file can be accessed with the `scoreboard_get_value()` function. If an attribute has more than one value, then the list of values is obtained as a concatenated string where each value is delimitted by comma (,).

Return Values

`0`	Succeeded
`ENOMEM`	Not enough memory
others	Depending on environment

3.2 Member Context

```
#include <pm.h>
#include <pm_internal.h>

#define DEV     my_pmx_dev

PMX_PROTOTYPES(DEV);

static pmMemberOps ops = PMX_OPS(DEV);

typedef struct PMX_CTXTYPE(DEV) {
  pmMemberContext       pmc;
  /* private data may follow */

} PMX_CTXTYPE(DEV);
```

Figure 3.3: Member Context Definition

3.3 Template

3.3.1 `pmmOpenArgs()`

```
int pmmOpenArgs( char **man[], char **opt[] );
int pmmOpenMember( pmCompositeContext *pcc, int argc, char *argv[],
                pmNode nodes[], pmMemberContext *pmcp );
```

The `pmmOpen()` function creates a member context and the `pmmOpenArgs()` function returns the option names which can be accepted by the `pmmOpen()` function.

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EIO`	Internal error
others	The `mmap()` function may return with.

```
PMX_OpenMember(DEV, pcc, argc, argv, nodes, pmm) {
  /*
   * pmCompositeContext* pcc:   Composiet context
   * int argc:                  number of argument
   * char **argv:               argument vector
   * pmNode nodes[]:            list of hostname and process number pair
   */
  PMX_CTXTYPE(DEV)      *me;
  pm_key_t      key;
  pm_flags_t    flags;
  pm_node_t     nnodes, nprocs, procno;
  pm_node_t     nd, md;
  pm_size_t     tsmtu, osmtu;
  int           err;

  PMX_PUSH;
  PMX_CALL(err=pmxCompositeGetConfig(pcc,&key,&flags,&nnodes,&nprocs,&procno));
  if( PMX_IS_FAILED( err ) ) {
    PMX_ERETURN( err, "%s", pmxErrorString( err ) );
  }
  /*
   * create and setup member context (me)
   */
  *pmm = (pmMemberContext*) me;
  (*pmm)->ops = ops;
  for( nd=0; nd<nnodes; nd++ ) {
    /*
     * md:      node number of the member context
     * tsmtu:   MTU of two-sided communication
     * osmtu:   MTU of one-sided communication
     */
    PMX_CALL( err=pmxCompositeAddRoute(pcc,nd,md,*pmm,tsmtu,osmtu) );
    if( PMX_IS_FAILED( err ) ) {
      if( err == EBUSY ) {
        /* then the node entry is already set by the other member */
        /* and simple ignore this entry                           */
        continue;
      }
      /* free allocated member context, if needed */
      PMX_ERETURN( err, "%s", pmxErrorString( err ) );
    }
  }
  PMX_RETURN(PM_SUCCESS);
}
```

Figure 3.4: Code Template of `pmmOpenMember()`

3.3.2 pmmReset()

```
int pmmReset( pmContext *pmc );
```

Reset the PM context. If there are some messages in the receive and/or send buffer, those messages will be lost.

Return Values

PM_SUCCESS	Succeeded
EIO	Internal error

3.3.3 pmmInitialize() and pmmStart()

```
int pmmInitialize( pmContext *pmc, pm_flags_t flag );
int pmmStart( pmContext *pmc );
```

The `pmmInitialize()` and `pmmStart()` functions creates and starts the receive and send threads. Those two function must be called in a process in which actual PM communication takes place. If a contex is shared between processes to communicate, eash process must call those functions.

The end of calling the `pmmInitialize()` must be barrier-synchronized. The nodes specified in the `pmmOpenContext()` *MAY* not call those functions at the start up. This is because the node is allocated but there is no process at the time, but the process *MAY* be invoked in the future.

Implementation Note on The Case of Myrinet/MX

In the `pmmInitialize()` function, the end points of MX are created. In the `pmmStart()`, the `mx_connect()` function is called to connect end points. The barrier synchronization guarantees the existence of the end points on live hosts. Note that there can be the case where some processes on some hosts are not created and there are no end points on those hosts. This can happen when dynamic host allocation (creation) on MPI-2 takes place.

Return Values

PM_SUCCESS	Succeeded
EBUSY	Device already opened
EPROTO	Illegal state transition
ENOLINK	Device has no link
EHOSTUNREACH	there is an unreachable node
EIO	Internal error

3.3.4 pmmStop()

```
int pmmStop( pmContext *pmc );
```

Stop and destroy receive and send threads corresponding to the calling process. Those threads can be restarted or recreated by calling the `pmmInitialize()` and `pmmStart()` functions.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Device already opened
`EPROTO`	Illegal state transition
`EAGAIN`	There is at least one thread communicating
`EIO`	Internal error

3.3.5 `pmmBreak()` and `pmmWaive()`

```
int pmmBreak( pmContext pmc );
int pmmWaive( pmContext pmc );
```

Stop receive and send threads for network preemption (gang scheduling). The end of calling the `pmmBreak()` must be barrier-synchronized.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Device already opened
`EPROTO`	Illegal state transition
`EAGAIN`	There is at least one thread communicating
`EIO`	Internal error

3.3.6 `pmmContinue()`

```
int pmmContinue( pmContext pmc );
```

Resume the suspended receive and send threads which are suspended by calling the `pmmBreak()` and `pmmWaive()` functions.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Device already opened
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.7 `pmmClose()`

```
int pmmClose( pmContext *pmc );
```

The `pmmClose()` function closes the PM context which was created by calling the `pmmOpenContext()`. After returning this function, the specified PM context will be invalid to access.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Device already opened
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.8 `pmmNotCommunicating()`

The `pmmNotCommunicating()` function returns `PM_SUCCESS` if there is no outstanding two-sided and one-sided communication. Otherwise it returns `EBUSY`.

```
int pmmNotCommunicating( pmContext *pmc );
```

The `pmmNotCommunicating()` function must be designed and implmented so that the function can be called on the PM context which is not called the `pmmInitialize()` and the `pmmStart()` functions, but those functions are already called by the process which is different from the process calling the `pmmNotCommunicating()` function.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	There are some outstanding communications

3.3.9 `pmmGetSendBuffer()`

To send a message, the `pmmGetSendBuffer()` function is called to allocate a send buffer.

```
int pmmGetSendBuffer( pmContext *pmc, pm_node_t node, void **bufp,
                      pm_size_t length, pm_desc_t *descp );
```

It returns the address of the allocated buffer and the send buffer descriptor. The `pmmGetSendBuffer()` function can be called by different processes or threads on the same shared PM context.

The returned buffer address is aligned to machine dependent address so that the buffer can be casted to any data type.

The send buffer information can be obtained by calling the `pmmGetSendDescInfo()` function on the send buffer descriptor returned by the `pmmGetSendBuffer()` function.

It is not allowed to send a message to the node itself, nor to send a message having zero length.

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. Therefor the `pmmGetSendbuffer()` should have a lock during its execution.

And when the context is already locked, the `pmmGetSendBuffer()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`ENOBUFS`	No buffer space available
`EBUSY`	Already locked
`EINVAL`	Illegal arguments
`EPROTO`	Illegal state transition
`EMSGSIZE`	Message too long or zero message size
`ERANGE`	Illegal node number
`EIO`	Internal error

3.3.10 `pmmGetSendDescInfo()`

```
   int pmmGetSendDescInfo( pmContext *pmc, pm_desc_t desc,
                           pmSendDescInfo *infop );
```

```
/*
 * PM Descriptor Info.
 */
typedef struct pm_send_desc_info {
        int             status;         /* status */
        pm_addr_t       addr;           /* buffer address */
        pm_size_t       length;         /* message length */
} pmSendDescInfo;
```

Figure 3.5: `pmSendDescInfo` Definition

And the `pmSendDescInfo` structure is defined as above. If the corresponding message is already sent or not found in the PM context, then the `pmmGetSendDescInfo()` function returns `EBADR`.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EIO`	Internal error

3.3.11 `pmmTruncateBuffer()`

```
   int pmmTruncateBuffer( pmContext *pmc, pm_desc_t desc,
                           pm_size_t length );
```

The `pmmTruncateBuffer()` function can shrinks the send buffer length to the specified length. The `pmmTruncateBuffer()` function can be called any number of times in the execution between the call of the `pmmGetSendBuffer()` function and the call of the `pmmSend()` function, and shrinked buffer can be expanded up to the length which is specified by the `pmmGetSendBuffer()`

function. Setting the `length` of zero results in returning an error (`EINVAL`).

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.12 `pmmKeepSendDesc()`

```
int pmmKeepSendDesc( pmContext *pmc, pm_desc_t desc );
```

The `pmmReleaseSendDesc()` function destroys the send descriptor which is kept by calling the `pmSend()` function.

If the `pmmKeepSendDesc()` function is called, then the buffer descriptor is kept and the corresponding buffer region must *not* be reclaimed, until the descriptor is released with calling the `pmmReleaseSendDesc()` functoion.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EIO`	Internal error

3.3.13 `pmmSend()`

Once the content of the send message is fixed, then the `pmmSend()` function is called to send the message. After calling the `pmmSend()` function, the modification on the content of the allocated send buffer may *not* be refelected to the received message.

```
int pmmSend( pmContext *pmc, pm_desc_t desc );
```

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. In some implementation, a lock during the `pmmSend()` execution may be required. And when the context is already locked, the `pmmSend()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.14 `pmmReleaseSendDesc()`

```
int pmmReleaseSendDesc( pmContext *pmc, pm_desc_t desc );
```

The `pmmReleaseSendDesc()` function destroys the send descriptor which is kept by calling the `pmSend()` function.

Return Values

`PM_SUCCESS`	Succeeded
`EBADR`	No such descriptor
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.15 `pmmIsSendDone()`

```
   int pmmIsSendDone( pmContext *pmc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not yet
`EIO`	Internal error

3.3.16 `pmmReceive()`

The `pmmReceive()` function tries to get a received message.

```
   int pmmReceive( pmContext *pmc, void **bufp, pm_size_t *sizep,
                   pm_desc_t *descp );
```

The `pmmReceive()` function returns the address of the received message and the receive buffer descriptor, if there is a received message.

The returned buffer address is aligned to machine dependent address so that the buffer can be casted to any data type.

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`ENOBUFS`	No message to receive available
`ESHUTDOWN`	Receiving is disabled
`EIO`	Internal error

3.3.17 `pmmReleaseReceiveBuffer()`

```
   int pmmReleaseReceiveBuffer( pmContext *pmc, pm_desc_t desc );
```

The descriptor is released and corresponding buffer region is reclaimed when the `pmmReleaseReceiveBuffer()` is called. There is no way and no need of getting information from the receive buffer descriptor.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EBADR`	No such descriptor
`EIO`	Internal error

3.3.18 `pmmBeforeSelect()`

```
int pmmBeforeSelect( pmContext *pmc, fd_set *fds, int *maxfdp,
                     sigset_t *sigmask );
```

The `pmmBeforeSelect()` function tells PM context that user program is going to block to wait for incoming messages. It returns the file descriptor(s) in `fds`, the maximum number of the file descriptor, and the signal mask of Linux so that those variables can be passed to the `pselect()` function. There can be the case where user sets those variables and the `pmmBeforeSelect()` function never resets them.

The `pmmBeforeSelect()` function is designed so that the wait can be implemented with using the Linux `pselect()` function. The `fds` and the `maxfdp` variables are set by the `pmmBeforeSelect()` function so that they can be passed to the Linux `pselect()` function. It should be noticed that the `pmmBeforeSelect()` function does not always return with the file descriptor(s) to be `pselect()`ed. In this case, the `maxfdp` will be returned unchanged and the `fds` and `fdmaxp` may not be suitable for passing the `pselect()` function.

Also the `sigmask` can be passed so that some specific signal(s) can unblock `pselect()` function. To avoid the race condition, if a Linux signal is used to unblock the waiting, then the signal must be blocked in the `pmmBeforeSelect()` function, and can be unblocked in the `pmmAfterSelect()` function.

The `pmmBeforeSelect()` and the `pmmAfterSelect()` functions must be called in pair always. Since it may be very difficult to avoid the race condition in some implementations, the `pmmReceive()` function must be called before calling the `pselect()` function so that the messages received in prior to the call of the `pmmBeforeSelect()` function are extracted from the receive buffer. The `pmmBeforeSelect()` function call never guarantees the presence of received message(s) when the `pselect()` function returns with a positive integer larger than zero. User program must be programmed to handle the case where the `pselect()` function tells there seems to be some received messages, but actually there is no received messages. In addition, the `pselect()` may be blocked even if there are some messages which can be be received. Thus it is not recommended to set the infinite or very larger timeout duration (by setting NULL of the timeout parameter).

The returned set of the file descriptors *MAY* be changed while the execution of a parallel process, especially when the parallel process spawns (adds) the other processes or some processes in the parallel process terminates. Thus the set of file descriptors must be obtained by the function every time `pselect()` is called.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.19 `pmmAfterSelect()`

```
int pmmAfterSelect( pmContext *pmc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.20 `pmmExport()`

The `pmmExport()` function allows remote `node` to access the specified local memory region with the one-sided communication way. The `node` value can be `PM_NODE_ANY`.

```
int pmmExport( pmContext *pmc, pm_addr_t addr,
               pm_size_t length, pm_node_t node, pmAddrHandle *handle );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.21 `pmmUnexport()`

The `pmmUnexport()` function destroys the memory handle which is created on the same host.

```
int pmmUnexport( pmContext *pmc, pmAddrHandle *handle );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.22 pmmRead()

```
int pmmRead( pmContext *pmc, pmAddrHandle src, pm_off_t soff,
                pmAddrHandle dest, pm_off_t doff,
                pm_size_t length, pm_desc_t *descp );
```

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. In some implementation, a lock during the `pmmRead()` execution may be required. And when the context is already locked, the `pmmRead()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EREMOTE`	Destination is remote
`EFAULT`	Bad address handle
`EIO`	Internal error

3.3.23 pmmWrite()

```
int pmmWrite( pmContext *pmc, pmAddrHandle src, pm_off_t soff,
                pmAddrHandle dest, pm_off_t doff,
                pm_size_t length, pm_desc_t *descp );
```

If the PM device of the context is Inter-Host routing, then the context may be shared between processes in a host. In some implementation, a lock during the `pmmWrite()` execution may be required. And when the context is already locked, the `pmmWrite()` returns `EBUSY`.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Already locked
`EPROTO`	Illegal state transition
`EREMOTE`	Source is remote
`EFAULT`	Bad address handle
`EIO`	Internal error

3.3.24 pmmIsReadDone()

The `pmmIsReadDone()` function checks is the posted one-sided write operations succeeded or not. If they are succeeded, the function retunrs `PM_SUCCESS` otherwise returns `EBUSY`.

```
int pmmIsReadDone( pmContext *pmc, pm_desc_t desc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not yet finished
`EIO`	Internal error

3.3.25 `pmmIsWriteDone()`

The `pmmIsWriteDone()` function checks is the posted one-sided write operations succeeded or not. If they are succeeded, the function retunrs `PM_SUCCESS` otherwise returns `EBUSY`.

```
int pmmIsWriteDone( pmContext *pmc, pm_desc_t desc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not yet finished
`EIO`	Internal error

3.3.26 `pmmSave()`

```
int pmmSave( pmContext *pmc, void **savep );
```

If the context is located on a mmap segment which is shared with the other process(es), then the shared mmap region will not be restored at restart. To avoid this, the `pmmSave()` function copies the content of the shared region to an allocated memory area which can be checkpointed and restarted. After calling this function, the `savep` points the address of the (possibly malloc()ed) memory region which has enough information to recreate the context.

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EBUSY`	Not ready to save context
`EPROTO`	Illegal state transition
`ENOBUFS`	No buffer space available
`EIO`	Internal error

3.3.27 `pmmReopen()`

```
int pmmReopen( pmCompositeContext *pcc, pmContext *pmc,
               pmNode nodes[], void *save );
```

Recreate context based on the information pointed by the `save` argument. It must be guaranteed that it must be recreated having the same number of nodes and numer of processes in a host. However, the `nodes` may vary from the one when the context was created and saved. This can happen when the process is migrated from a host to another.

Return Values

`PM_SUCCESS`	Succeeded
`EINVAL`	Illegal arguments
`EBUSY`	Not ready to restore context
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.28 `pmmBreak()`

```
int pmmBreak( pmContext *pmc );
```

The `pmmBreak()` function inhibits the state transitions of the messages in the send buffer(s) of the PM context. In addition to this, in some implementation, the function must be prepared for the following call of the `pmmWaive()` function so that the network can be preempted.

At the result of the freezing the states of sending messages, the `pmmGetSendBuffer()` and `pmSend()` functions must return `EBUSY` without any action.

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not ready
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.29 `pmmWaive()`

```
int pmmWaive( pmContext *pmc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not ready
`EPROTO`	Illegal state transition
`EIO`	Internal error

3.3.30 `pmmContinue()`

```
int pmmContinue( pmContext *pmc );
```

Return Values

`PM_SUCCESS`	Succeeded
`EBUSY`	Not ready
`EPROTO`	Illegal state transition
`EIO`	Internal error

Chapter 4

Common Routines

4.1 PM Node Set

```
#include <pm_nodeset.h>
void     pmx_nodeset_clr(pmNodeset*);
int      pmx_nodeset_isset(pm_node_t, pmNodeset*);
void     pmx_nodeset_set(pm_node_t, pmNodeset*);
void     pmx_nodeset_zero(pm_node_t, pmNodeset*);
pm_node_t pmx_nodeset_find(pm_node_t, pm_node_t, pmNodeset*);
```

Figure 4.1: PM Node Set

4.2 Machine Dependent Operations

```
#include <pm_machdep.h>
void pmx_memory_barrier( void );

double pmx_get_time( void );
int pmx_poll_test( struct pmx_cputimer *ptc );
void pmx_poll_delay( struct pm_cputimer *ptc );
void pmx_poll_timer_reset( struct pm_cputimer *ptc );
void pmx_usec_delay( int usec );
```

Figure 4.2: PM Machine Dependent Functions

4.3 Spin Lock and Aotmic Op.

PM provides spin-lock functions shown in Figure 4.3.

4.4 Network Byte Order

The underlying protocol should be designed to be used in heterogeneous clusters.

```
#include <pm_internal.h>

int pmx_initlock(pmLock *lock);
int pmx_destroylock(pmLock *lock);
int pmx_trylock(pmLock *lock);
int pmx_unlock(pmLock *lock);
```

Figure 4.3: PM Spin-Lock and Aotmic Op.

```
#include <pm_byteorder.h>

uint16_t pmx_ntohs(uint16_t nshort);
uint32_t pmx_ntohl(uint32_t nlong);
uint64_t pmx_ntohll(uint64_t nlonglong);
uint16_t pmx_htons(uint16_t hshort);
uint32_t pmx_htonl(uint32_t hlong);
uint64_t pmx_htonll(uint64_t hlonglong);
```

Figure 4.4: Network Byte-Ordering Functions

4.5 File Descriptors

```
#include <pm_internal.h>
int pmx_open(const char *pathname, int flags, mode_t mode);
int pmx_close_on_exec( int );
int pmx_nonblock_mode( int );
```

Figure 4.5: File Descriptor Related Functions

4.6 Hostname

The `pmx_gethostname()` function returns FQDN (Full Qualified Domain Name). Unlike the `gethostname()` function, the `pmx_gethostname()` is thread safe.

4.7 Pthread

The `pmx_fork_pthread()` function creates a thread which is in the detached state (unable to join).

4.8 Temporary Files

The temporary files, including the files for `mmap()`, must be created under the directory of `/var/score/pm/`*devicename*`/` defined as `PM_DIR_PREFIX`.

```
#include <pm_internal.h>

int pmx_gethostname( char *name, pm_size_t len );
```

Figure 4.6: `pmx_gethostname()`

```
#include <pm_internal.h>

int pmx_fork_pthread( void *(*thread_func)(void*), void* argp );
```

Figure 4.7: `pmx_fork_pthread()`

4.9 Stack Variables

Any PM functions shall not allocate large amount of memory on stack.

4.10 Debug Support

Table 4.1: Value of the `PM_DEBUG` environment variable

Value	Tag	Output messages
0	-	No information will be displayed. (Default)
1	Error	Information of unrecoverable error
2	Warning	Above and information on temporary error
3	Info	Above and any information (even if succeeded)
Higher	-	More messages may be displayed depending on PM device

See 3.4 for the usage of the debug support macros below.

4.11 Misc.

```
   #include <pm.h>
   extern const char *pmErrorString(int);
```

```
#include <pm_internal.h>

PMX_PUSH;
PMX_MEMO( format ... );
PMX_CALL( funcall );
PMX_RETURN( value );
PMX_VRETURN();
PMX_ERETURN( value, format ... );
PMX_VRETURN( format ... );
```

Figure 4.8: Macros for debugging

Chapter 5

How PMXfunctions are called in real world

5.1 Initializing PM for SCore-D

Here is the calling sequence when SCore-D initializes PM context for the communication of SCore-Ditself.

Table 5.1: Calling Sequence at SCore-D Initialization

No.	Function	Note
1	`pmxOpenDevice()`	Open All Pm devices allocated
2	`pmxGetDeviceAttribute()`	Check device attributes
3	`pmxIsReachable()`	Check network coverage
4	`pmxOpenContext()`	Open a context for SCore-D comm.
5	`pmxReset()`	
6	`pmxInitialize()`	
7	barrier synchronization	
8	`pmxStart()`	
	`pmxBeforeSelect()`	
	`pmxReceive()`	
	`pmxReleaseReceiveBuffer()`	
	`pmxAfterSelect()`	

5.2 Initializing PM for User Processes

Here is the calling sequence to prepare PM contexts for user processes.

5.3 Normal Termination of User Processes

Here is the calling sequence to shutdown PM contexts for user processes.

5.4 Network Preemption

Here is the calling sequence to switch PM contexts for gang-scheduling.

Table 5.2: Calling Sequence for Initializing User contexts

No.	User	SCore-D	Note
1	`pmxOpen()`		
2	`pmxAddMember()`		
3	Pass Control FD to SCore-D		
4		`pmxInitialize()`	
5		barrier synchronization	
6		`pmxStart()`	

Table 5.3: Calling Sequence for Initializing User contexts

No.	SCore-D	User	Note
1		`pmxStop()`	
2		`pmxClose()`	
3		`exit()`	
4		barrier synchronization	
5		`pmxClose()`	

5.5 Checkpoint

Here is the calling sequence to checkpoint a user process.

5.6 Restart

Here is the calling sequence to restart user process form a checkpoint.

Table 5.4: Calling Sequence for Gang Scheduling

No.	SCore-D	Signal	Note
1	`pmxBreak()`		
2	barrier synchronization		
3	`pmxWaive()`		
4		`SIGSTOP`	Stop user process(es)
5	barrier synchronization		
6		`SIGCONT`	Start user process(es)
7	`pmxContinue()`		

Table 5.5: Calling Sequence for Checkpoint

No.	Signal	SCore-D	Parity	Note
1		`pmxStop()`		
2	`SIGQUIT`	`IFSTOPPED()`		
3			barrier synchronization	
4		`pmxSave()`		
5		`pmxReset()`		
6			`pmxInitialize()`	
7			barrier synchronization	
8			`pmxStart()`	
9	Checkpoint Parity			if needed
10			`pmxStop()`	
11			`pmxClose()`	
12		`pmxReopen()`		
13		`pmxInitialize()`		
14		barrier synchronization		
15		`pmxStart()`		

Table 5.6: Calling Sequence for Restart

No.	User	Note
1	`pmxOpen()`	
2	`pmxAddMember()`	
3	`pmxResopen()`	
4	`pmxInitialize()`	
5	barrier synchronization	
6	`pmxStart()`	

Chapter 6

Development, Test and Tuning

6.1 Test Programs

6.1.1 Functional Test

```
pmxtest <options> ... <netopts> ...

-v[erbose]
-iter[ation] <N>
-duration <SEC>
-composite <NPROCS>
-func[tion] [min|ts|os|scored|ckpt|compat|{all}]
-scbd <netname> | -device <devname>縲<opendevice_args>
```

6.1.2 Performance Test

Index