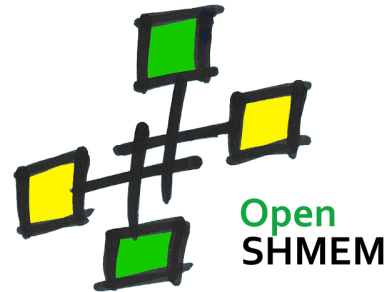


# OpenSHMEM

## Application Programming Interface



<http://www.openshmem.org/>

Version 1.2

15th March 2015

### Developed by

- High Performance Computing Tools group at the University of Houston  
<http://www.cs.uh.edu/~hpctools/>
- Extreme Scale Systems Center, Oak Ridge National Laboratory  
<http://www.csm.ornl.gov/essc/>

## Sponsored by

- U.S. Department of Defense (DoD)  
<http://www.defense.gov/>
- Oak Ridge National Laboratory (ORNL)  
<http://www.ornl.gov/>

## Authors and Collaborators

- Monika ten Bruggencate, Cray Inc.
- David Knaak, Cray Inc.
- Jens Manser, DoD
- Nick Park, DoD
- Lauren Smith, DoD
- James Dinan, Intel
- Jeff Hammond, Intel
- Jeff Kuehn, Los Alamos National Laboratory (LANL)
- Swaroop Pophale, Mellanox
- Eduardo D’Azevedo, ORNL
- Manjunath Gorentla Venkata, ORNL
- Oscar Hernandez, ORNL
- Gregory Koenig, ORNL
- Graham Lopez, ORNL
- Pavel Shamis, ORNL
- Steve Poole, OSSS
- Karl Feind, SGI
- Michael Raymond, SGI
- Barbara Chapman, University of Houston (UH)
- Tony Curtis, UH
- Sameer Shende, University of Oregon (UO)

## Acknowledgements

The OpenSHMEM specification belongs to Open Source Software Solutions, Inc. (OSSS), a non-profit organization, under an agreement with SGI. The development work of the specification is supported by the Oak Ridge National Laboratory Extreme Scale Systems Center and the Department of Defense.

We would also like to acknowledge the contribution of the members of the OpenSHMEM mailing list for their ideas, discussions, suggestions, and constructive criticism which has helped us improve this document.

# Contents

1	The OpenSHMEM Effort . . . . .	1	1
2	Programming Model Overview . . . . .	1	12
3	Memory Model . . . . .	3	13
4	Execution Model . . . . .	4	14
4.1	Progress of OpenSHMEM Operations . . . . .	4	15
4.2	Atomicity Guarantees . . . . .	4	16
5	Language Bindings and Conformance . . . . .	5	17
6	Library Constants . . . . .	5	18
7	Environment Variables . . . . .	6	19
8	OpenSHMEM Library API . . . . .	7	20
8.1	Library Setup, Exit, and Query Routines . . . . .	7	21
8.1.1	SHMEM_INIT . . . . .	7	22
8.1.2	SHMEM_MY_PE . . . . .	8	23
8.1.3	SHMEM_N_PES . . . . .	9	24
8.1.4	SHMEM_FINALIZE . . . . .	10	25
8.1.5	SHMEM_GLOBAL_EXIT . . . . .	11	26
8.1.6	SHMEM_PE_ACCESSIBLE . . . . .	12	27
8.1.7	SHMEM_ADDR_ACCESSIBLE . . . . .	13	28
8.1.8	SHMEM_PTR . . . . .	14	29
8.1.9	SHMEM_INFO_GET_VERSION . . . . .	15	30
8.1.10	SHMEM_INFO_GET_NAME . . . . .	16	31
8.1.11	START_PES . . . . .	17	32
8.2	Memory Management Routines . . . . .	18	33
8.2.1	SHMEM_MALLOC, SHMEM_FREE, SHMEM_REALLOC, SHMEM_ALIGN	18	34
8.2.2	SHPALLOC . . . . .	19	35
8.2.3	SHPCLMOVE . . . . .	20	36
8.2.4	SHPDEALLC . . . . .	21	37
8.3	Remote Memory Access Routines . . . . .	21	38
8.3.1	SHMEM_PUT . . . . .	22	39
8.3.2	SHMEM_P . . . . .	24	40
8.3.3	SHMEM_IPUT . . . . .	25	41
8.3.4	SHMEM_GET . . . . .	27	42
8.3.5	SHMEM_G . . . . .	29	43
8.3.6	SHMEM_IGET . . . . .	30	44
8.4	Atomic Memory Operations . . . . .	32	45
8.4.1	SHMEM_ADD . . . . .	32	46
8.4.2	SHMEM_CSWAP . . . . .	34	47
8.4.3	SHMEM_SWAP . . . . .	35	48
8.4.4	SHMEM_FINC . . . . .	36	49
8.4.5	SHMEM_INC . . . . .	38	50
8.4.6	SHMEM_FADD . . . . .	39	51
8.5	Collective Routines . . . . .	40	52
8.5.1	SHMEM_BARRIER_ALL . . . . .	41	53

---

8.5.2	<b>SHMEM_BARRIER</b>	42
8.5.3	<b>SHMEM_BROADCAST</b>	43
8.5.4	<b>SHMEM_COLLECT, SHMEM_FCOLLECT</b>	46
8.5.5	<b>SHMEM_REDUCTIONS</b>	48
8.6	Point-To-Point Synchronization Routines	56
8.6.1	<b>SHMEM_WAIT</b>	56
8.7	Memory Ordering Routines	58
8.7.1	<b>SHMEM_FENCE</b>	59
8.7.2	<b>SHMEM_QUIET</b>	60
8.7.3	Synchronization and Communication Ordering in OpenSHMEM	61
8.8	Distributed Locking Routines	64
8.8.1	<b>SHMEM_LOCK</b>	64
8.9	Cache Management	65
8.9.1	<b>SHMEM_CACHE</b>	65
<b>A</b>	<b>Writing OpenSHMEM Programs</b>	<b>67</b>
<b>B</b>	<b>Compiling and Running Programs</b>	<b>70</b>
1	Compilation	70
2	Running Programs	70
<b>C</b>	<b>Undefined Behavior in OpenSHMEM</b>	<b>72</b>
<b>D</b>	<b>Interoperability with other Programming Models</b>	<b>73</b>
1	<i>Message Passing Interface (MPI)</i> Interoperability	73
<b>E</b>	<b>History of OpenSHMEM</b>	<b>74</b>
<b>F</b>	<b>OpenSHMEM Specification and Deprecated API</b>	<b>75</b>
<b>G</b>	<b>Changes to this Document</b>	<b>76</b>
1	Version 1.2	76
2	Version 1.1	77

## Introduction

### 1 The OpenSHMEM Effort

OpenSHMEM is a *Partitioned Global Address Space* (PGAS) library interface specification. OpenSHMEM aims to provide a standard *Application Programming Interface* (API) for SHMEM libraries to aid portability and facilitate uniform predictable results of OpenSHMEM programs by explicitly stating the behavior and semantics of the OpenSHMEM library calls. Through the different versions, OpenSHMEM will continue to address the requirements of the PGAS community. As of this specification, existing vendors are moving towards OpenSHMEM compliant implementations and new vendors are developing OpenSHMEM library implementations to help the users write portable OpenSHMEM code. This ensures that programs can run on multiple platforms without having to deal with subtle vendor-specific implementation differences. For more details on the history of OpenSHMEM please refer to [The History of OpenSHMEM](#) section.

The OpenSHMEM<sup>1</sup> effort is driven by the Extreme Scale Systems Center (ESSC) at ORNL and the University of Houston with significant input from the OpenSHMEM community. Besides the specification, the effort also includes providing a reference OpenSHMEM implementation, validation and verification suites, tools, a mailing list and website infrastructure to support specification activities. For more information please refer to: <http://www.openshmem.org/>.

### 2 Programming Model Overview

OpenSHMEM implements PGAS by defining remotely accessible data objects as mechanisms to share information among OpenSHMEM processes or *Processing Elements* (PEs) and private data objects that are accessible by the PE itself. The API allows communication and synchronization operations on both private (local to the PE initiating the operation) and remotely accessible data objects. The key feature of OpenSHMEM is that data transfer operations are *one-sided* in nature. This means that a local PE executing a data transfer routine does not require the participation of the remote PE to complete the routine. This allows for overlap between communication and computation to hide data transfer latencies, which makes OpenSHMEM ideal for unstructured, small/medium size data communication patterns. The OpenSHMEM library routines have the potential to provide a low-latency, high-bandwidth communication API for use in highly parallelized scalable programs.

The OpenSHMEM interfaces can be used to implement *Single Program Multiple Data* (SPMD) style programs. It provides interfaces to start the OpenSHMEM PEs in parallel, and communication and synchronization interfaces to access remotely accessible data objects across PEs. These interfaces can be leveraged to divide a problem into multiple sub-problems that can be solved independently or with coordination using the communication and synchronization interfaces. The OpenSHMEM specification defines library calls, constants, variables, and language bindings for *C* and *Fortran*. The *C++* interface is currently the same as that for *C*. Unlike UPC, Fortran 2008, Titanium, X10 and Chapel, which are all PGAS languages, OpenSHMEM relies on the user to use the library calls to implement the correct semantics of its programming model.

An overview of the OpenSHMEM routines is described below:

#### 1. Library Setup and Query

- (a) *Initialization*: The OpenSHMEM library environment is initialized.
- (b) *Query*: The local PE may get number of PEs running the same program and its unique integer identifier.
- (c) *Accessibility*: The local PE can find out if a remote PE is executing the same binary, or if a particular symmetric data object can be accessed by a remote PE, or may obtain a pointer to a symmetric data object on the specified remote PE on shared memory systems.

#### 2. Symmetric Data Object Management

---

<sup>1</sup>The OpenSHMEM specification is owned by Open Source Software Solutions Inc., a non-profit organization, under an agreement with SGI.

- 1 (a) *Allocation*: All executing PEs must participate in the allocation of a symmetric data object with identical
- 2 arguments.
- 3 (b) *Deallocation*: All executing PEs must participate in the deallocation of the same symmetric data object
- 4 with identical arguments.
- 5 (c) *Reallocation*: All executing PEs must participate in the reallocation of the same symmetric data object with
- 6 identical arguments.

### 7 3. Remote Memory Access

- 8
- 9 (a) *Put*: The local PE specifies the *source* data object (private or symmetric) that is copied to the symmetric
- 10 data object on the remote PE.
- 11 (b) *Get*: The local PE specifies the symmetric data object on the remote PE that is copied to a data object
- 12 (private or symmetric) on the local PE.

### 13 4. Atomics

- 14
- 15 (a) *Swap*: The PE initiating the swap gets the old value of a symmetric data object from a remote PE and
- 16 copies a new value to that symmetric data object on the remote PE.
- 17 (b) *Increment*: The PE initiating the increment adds 1 to the symmetric data object on the remote PE.
- 18 (c) *Add*: The PE initiating the add specifies the value to be added to the symmetric data object on the remote
- 19 PE.
- 20 (d) *Compare and Swap*: The PE initiating the swap gets the old value of the symmetric data object based on a
- 21 value to be compared and copies a new value to the symmetric data object on the remote PE.
- 22 (e) *Fetch and Increment*: The PE initiating the increment adds 1 to the symmetric data object on the remote
- 23 PE and returns with the old value.
- 24 (f) *Fetch and Add*: The PE initiating the add specifies the value to be added to the symmetric data object on
- 25 the remote PE and returns with the old value.

### 26 5. Synchronization and Ordering

- 27
- 28 (a) *Fence*: The PE calling fence ensures ordering of remote access operations and stores to symmetric data
- 29 objects with respect to a specific destination PE.
- 30 (b) *Quiet*: The PE calling quiet ensures completion of remote access operations and stores to symmetric data
- 31 objects.
- 32 (c) *Barrier*: All or some PEs collectively synchronize and ensure completion of all remote and local updates
- 33 prior to any PE returning from the call.

### 34 6. Collective Communication

- 35
- 36 (a) *Broadcast*: The *root* PE specifies a symmetric data object to be copied to a symmetric data object on one
- 37 or more remote PEs (not including itself).
- 38 (b) *Collection*: All PEs participating in the routine get the result of concatenated symmetric objects contributed
- 39 by each of the PEs in another symmetric data object.
- 40 (c) *Reduction*: All PEs participating in the routine get the result of an associative binary routine over elements
- 41 of the specified symmetric data object on another symmetric data object.

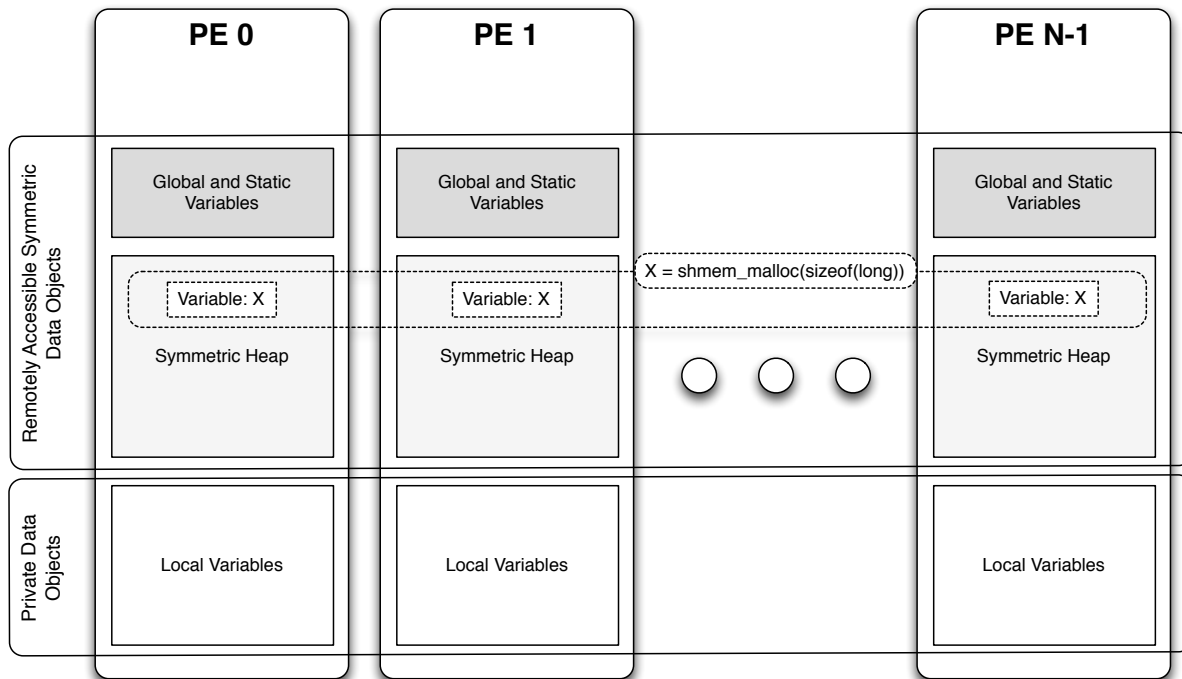
### 42 7. Mutual Exclusion

- 43
- 44 (a) *Set Lock*: The PE acquires exclusive access to the region bounded by the symmetric *lock* variable.
- 45 (b) *Test Lock*: The PE tests the symmetric *lock* variable for availability.
- 46 (c) *Clear Lock*: The PE which has previously acquired the *lock* releases it.

### 47 8. Data Cache Control (*deprecated on cache coherent systems* )

- 48 (a) Implementation of mechanisms to exploit the capabilities of hardware cache if available.

## 3 Memory Model

Figure 1: *OpenSHMEM* Memory Model

An OpenSHMEM program consists of data objects that are private to each PE and data objects that are remotely accessible by all PEs. Private data objects are stored in the local memory of each PE and can only be accessed by the PE itself; these data objects cannot be accessed by other PEs via OpenSHMEM routines. Private data objects follow the memory model of *C* or *Fortran*. Remotely accessible objects, however, can be accessed by remote PEs using OpenSHMEM routines. Remotely accessible data objects are called *Symmetric Objects*. Each symmetric data object has a corresponding object with the same name, type, size, and offset (from an arbitrary memory address) on all PEs where that object is accessible via the OpenSHMEM API. (For the definition of what is accessible, see the descriptions for *shmem\_pe\_accessible* and *shmem\_addr\_accessible* in sections 8.1.6 and 8.1.7.) Symmetric data objects accessed via typed OpenSHMEM interfaces are required to be natural aligned based on their type requirements and underlying architecture. In OpenSHMEM the following kinds of data objects are symmetric:

- *Fortran* data objects in common blocks or with the SAVE attribute. These data objects must not be defined in a dynamic shared object (DSO).
- Global and static *C* and *C++* variables. These data objects must not be defined in a DSO.
- *Fortran* arrays allocated with *shpalloc*
- *C* and *C++* data allocated by *shmem\_malloc*

OpenSHMEM dynamic memory allocation routines (*shpalloc* and *shmem\_malloc*) allow collective allocation of *Symmetric Data Objects* on a special memory region called the *Symmetric Heap*. The Symmetric Heap is created during the execution of a program at a memory location determined by the implementation. The Symmetric Heap may reside in different memory regions on different PEs. Figure 1 shows how OpenSHMEM implements a PGAS model using remotely accessible symmetric objects and private data objects when executing an OpenSHMEM program. Symmetric data objects are stored on the symmetric heap or in the global/static memory section of each PE.

## 4 Execution Model

An OpenSHMEM program consists of a set of OpenSHMEM processes called PEs that execute in a SPMD-like model where each PE can take a different execution path. A PE can be implemented using an OS process or an OS thread<sup>2</sup>. The PEs progress asynchronously, and can communicate/synchronize via the OpenSHMEM interfaces. All PEs in an OpenSHMEM program should start by calling the initialization routine `shmem_init`<sup>3</sup> before using any of the other OpenSHMEM library routines. An OpenSHMEM program finishes execution by returning from the main routine or when any PE calls `shmem_global_exit`. When returning from main, OpenSHMEM must complete all pending communication and release all the resources associated to the library using an implicit collective synchronization across PEs. The user has the option to call `shmem_finalize` (before returning from main) to complete all pending communication and release all the OpenSHMEM library resources without terminating the program. Calling any OpenSHMEM routine after `shmem_finalize` leads to undefined behavior.

The PEs of the OpenSHMEM program are identified by unique integers. The identifiers are integers assigned in a monotonically increasing manner from zero to the total number of PEs minus 1. PE identifiers are used for OpenSHMEM calls (e.g. to specify *Put* or *Get* routines on symmetric data objects, collective synchronization calls) or to dictate a control flow for PEs using constructs of *C* or *Fortran*. The identifiers are fixed for the life cycle of the OpenSHMEM program.

### 4.1 Progress of OpenSHMEM Operations

The OpenSHMEM model assumes that computation and communication are naturally overlapped. OpenSHMEM programs are expected to exhibit progression of communication both with and without OpenSHMEM calls. Consider a PE that is engaged in a computation with no OpenSHMEM calls. Other PEs should be able to communicate (*put*, *get*, *collective*, *atomic*, etc) and complete communication operations with that computationally-bound PE without that PE issuing any explicit OpenSHMEM calls. OpenSHMEM communication calls involving that PE should progress regardless of when that PE next engages in an OpenSHMEM call.

#### Note to implementors:

- An OpenSHMEM implementation for hardware that does not provide asynchronous communication capabilities may require a software progress thread in order to process remotely-issued communication requests without explicit program calls to the OpenSHMEM library.
- High performance implementations of OpenSHMEM are expected to leverage hardware offload capabilities and provide asynchronous one-sided communication without software assistance.
- Implementations should avoid deferring the execution of one-sided operations until a synchronization point where data is known to be available. High-quality implementations should attempt asynchronous delivery whenever possible, for performance reasons. Additionally, the OpenSHMEM community discourages releasing OpenSHMEM implementations that do not provide asynchronous one-sided operations, as these have very limited performance value for OpenSHMEM programs.

### 4.2 Atomicity Guarantees

OpenSHMEM contains a number of routines that operate on symmetric data atomically (Section 8.4). These routines guarantee that accesses by OpenSHMEM's atomic operations will be exclusive, but do not guarantee exclusivity in combination with other routines, either inside OpenSHMEM or outside.

For example: during the execution of an atomic remote integer increment operation on a symmetric variable *X*, no other OpenSHMEM atomic operation may access *X*. After the increment, *X* will have increased its value by 1 on the destination PE, at which point other atomic operations may then modify that *X*. However, access to the symmetric object *X* with non-atomic operations, such as one-sided *Put* or *Get* operations, will *invalidate* the atomicity guarantees.

<sup>2</sup>Implementing PEs using OS threads requires compiler techniques to implement the OpenSHMEM memory model.

<sup>3</sup>`start_pes` has been deprecated as of Specification 1.2



## 5 Language Bindings and Conformance

OpenSHMEM provides ISO *C* and *Fortran 90* language bindings. Any implementation that provides both *C* and *Fortran* bindings can claim conformance to the specification. An implementation that provides e.g. only a *C* interface may claim to conform to the OpenSHMEM specification with respect to the *C* language, but not to *Fortran*, and should make this clear in its documentation. The OpenSHMEM header files for *C* and *Fortran* must contain only the interfaces and constant names defined in this specification.

OpenSHMEM APIs can be implemented as either routines or macros. However, implementing the interfaces using macros is strongly discouraged as this could severely limit the use of external profiling tools and high-level compiler optimizations. An OpenSHMEM program should avoid defining routine names, variables, or identifiers with the prefix *SHMEM\_* (for *C* and *Fortran*), *\_SHMEM\_* (for *C*) or with OpenSHMEM API names.

All OpenSHMEM extension APIs that are not part of this specification must be defined in the *shmemx.h* include file. These extensions shall use the *shmemx\_* prefix for all routine, variable, and constant names.

## 6 Library Constants

The constants that start with *SHMEM\_\** are for *Fortran*. For backwards compatibility, the *C/C++* constants may start with either *\_SHMEM\_\** or *SHMEM\_\**.

Constant	Description
<i>C/C++:</i> _SHMEM_BCAST_SYNC_SIZE SHMEM_BCAST_SYNC_SIZE <i>Fortran:</i> SHMEM_BCAST_SYNC_SIZE	Length of the <i>pSync</i> arrays needed for broadcast routines. The value of this constant is implementation specific. Refer to the <a href="#">Broadcast Routines</a> section under <a href="#">Library Routines</a> for more information about the usage of this constant.
<i>C/C++:</i> _SHMEM_SYNC_VALUE SHMEM_SYNC_VALUE <i>Fortran:</i> SHMEM_SYNC_VALUE	Holds the value used to initialize the elements of <i>pSync</i> arrays. The value of this constant is implementation specific.
<i>C/C++:</i> _SHMEM_REDUCE_SYNC_SIZE SHMEM_REDUCE_SYNC_SIZE <i>Fortran:</i> SHMEM_REDUCE_SYNC_SIZE	Length of the work arrays needed for reduction routines. The value of this constant is implementation specific. Refer to the <a href="#">Reduction Routines</a> section under <a href="#">Library Routines</a> for more information about the usage of this constant.
<i>C/C++:</i> _SHMEM_BARRIER_SYNC_SIZE SHMEM_BARRIER_SYNC_SIZE <i>Fortran:</i> SHMEM_BARRIER_SYNC_SIZE	Length of the work array needed for barrier routines. The value of this constant is implementation specific. Refer to the <a href="#">Barrier Synchronization Routines</a> section under <a href="#">Library Routines</a> for more information about the usage of this constant.
<i>C/C++:</i> _SHMEM_COLLECT_SYNC_SIZE SHMEM_COLLECT_SYNC_SIZE <i>Fortran:</i> SHMEM_COLLECT_SYNC_SIZE	Length of the work array needed for collect routines. The value of this constant is implementation specific. Refer to the <a href="#">Collect Routines</a> section under <a href="#">Library Routines</a> for more information about the usage of this constant.

1 2 3 4 5 6	<p><i>C/C++:</i>            _SHMEM_REDUCE_MIN_WRKDATA_SIZE            SHMEM_REDUCE_MIN_WRKDATA_SIZE</p> <p><i>Fortran:</i>            SHMEM_REDUCE_MIN_WRKDATA_SIZE</p>	Minimum length of work arrays used in various collective routines.
7 8 9 10 11	<p><i>C/C++:</i>            _SHMEM_MAJOR_VERSION            SHMEM_MAJOR_VERSION</p> <p><i>Fortran:</i>            SHMEM_MAJOR_VERSION</p>	Integer representing the major version of OpenSHMEM standard in use.
12 13 14 15 16	<p><i>C/C++:</i>            _SHMEM_MINOR_VERSION            SHMEM_MINOR_VERSION</p> <p><i>Fortran:</i>            SHMEM_MINOR_VERSION</p>	Integer representing the minor version of OpenSHMEM standard in use.
17 18 19 20 21 22	<p><i>C/C++:</i>            _SHMEM_MAX_NAME_LEN            SHMEM_MAX_NAME_LEN</p> <p><i>Fortran:</i>            SHMEM_MAX_NAME_LEN</p>	Integer representing the length of vendor string.
23 24 25 26 27	<p><i>C/C++:</i>            _SHMEM_VENDOR_STRING            SHMEM_VENDOR_STRING</p> <p><i>Fortran:</i>            SHMEM_VENDOR_STRING</p>	String representing the string of length less than SHMEM_MAX_NAME_LEN .

## 7 Environment Variables

The OpenSHMEM specification provides a set of environment variables that allows users to configure the OpenSHMEM implementation, and receive information about the implementation. The implementations of the specification are free to define additional variables. Currently, the specification defines four environment variables.

Variable	Value	Routine
SMA_VERSION	any	print the library version at start-up
SMA_INFO	any	print helpful text about all these environment variables
SMA_SYMMETRIC_SIZE	non-negative integer	number of bytes to allocate for symmetric heap
SMA_DEBUG	any	enable debugging messages

## 8 OpenSHMEM Library API

### 8.1 Library Setup, Exit, and Query Routines

The library setup and query interfaces that initialize and monitor the parallel environment of the PEs.

#### 8.1.1 SHMEM\_INIT

A collective operation that allocates and initializes the resources used by the OpenSHMEM library.

#### SYNOPSIS

##### C/C++:

```
void shmem_init(void);
```

##### FORTRAN:

```
CALL SHMEM_INIT()
```

#### DESCRIPTION

##### Arguments

None.

##### API description

*shmem\_init* allocates and initializes resources used by the OpenSHMEM library. It is a collective operation that all PEs must call before any other OpenSHMEM routine may be called. At the end of the OpenSHMEM program which it initialized, the call to *shmem\_init* must be matched with a call to *shmem\_finalize*. After a single call to *shmem\_init*, a subsequent call to *shmem\_init* in the same program results in undefined behavior.

##### Return Values

None.

##### Notes

As of OpenSHMEM Specification 1.2 the use of *start\_pes* has been deprecated and is replaced with *shmem\_init*. While support for *start\_pes* is still required in OpenSHMEM libraries, users are encouraged to use *shmem\_init*. Replacing *start\_pes* with *shmem\_init* in OpenSHMEM programs with no further changes is possible; there is an implicit *shmem\_finalize* at the end of main. However, *shmem\_init* differs slightly from *start\_pes*: multiple calls to *shmem\_init* within a program results in undefined behavior, while in the case of *start\_pes*, any subsequent calls to *start\_pes* after the first one resulted in a no-op.

#### EXAMPLES

This is a simple program that calls *shmem\_init*:

```
PROGRAM PUT
  INTEGER TARG, SRC, RECEIVER, BAR
  COMMON /T/ TARG
  PARAMETER (RECEIVER=1)
  CALL SHMEM_INIT()

  IF (SHMEM_MY_PE() .EQ. 0) THEN
    SRC = 33
    CALL SHMEM_INTEGER_PUT(TARG, SRC, 1, RECEIVER)
```

```

1      ENDIF
2
3      CALL SHMEM_BARRIER_ALL           ! SYNCHRONIZES SENDER AND RECEIVER
4
5      IF (SHMEM_MY_PE() .EQ. RECEIVER) THEN
6          PRINT*, 'PE ', SHMEM_MY_PE(), ' TARG=', TARG, ' (expect 33)'
7      ENDIF
8
9      CALL SHMEM_FINALIZE()
10
11     END

```

## 8.1.2 SHMEM\_MY\_PE

Returns the number of the calling PE.

### SYNOPSIS

#### C/C++:

```
int shmem_my_pe(void);
```

#### FORTRAN:

```
INTEGER SHMEM_MY_PE, ME
ME = SHMEM_MY_PE()
```

### DESCRIPTION

#### Arguments

None

#### API description

This routine returns the PE number of the calling PE. It accepts no arguments. The result is an integer between 0 and  $npes - 1$ , where  $npes$  is the total number of PEs executing the current program.

#### Return Values

Integer - Between 0 and  $npes - 1$

#### Notes

Each PE has a unique number or identifier. As of OpenSHMEM Specification 1.2 the use of name `_my_pe` has been deprecated. Although OpenSHMEM libraries are required to support the call, users are encouraged to use `shmem_my_pe` instead. The behavior and signature of the routine `shmem_my_pe` remains unchanged from the deprecated `_my_pe` version.

### EXAMPLES

The following `shmem_my_pe` example is for C/C++ programs:

```

41  #include <stdio.h>
42  #include <shmem.h>
43
44  int main(void)
45  {
46      int me;
47
48      shmem_init();
49      me = shmem_my_pe();
50      printf("My PE id is: %d\n", me);

```

```
    return 0;
}
```

### 8.1.3 SHMEM\_N\_PES

Returns the number of PEs running in a program.

#### SYNOPSIS

##### C/C++:

```
int shmem_n_pes(void);
```

##### FORTRAN:

```
INTEGER SHMEM_N_PES, N_PES
N_PES = SHMEM_N_PES()
```

#### DESCRIPTION

##### Arguments

None

##### API description

The routine returns the number of PEs running the program.

##### Return Values

Integer - Number of PEs running the OpenSHMEM program.

##### Notes

As of OpenSHMEM Specification 1.2 the use of `_num_pes` has been deprecated. Although OpenSHMEM libraries are required to support the call, users are encouraged to use `shmem_n_pes` instead. The behavior and signature of the routine `shmem_n_pes` remains unchanged from the deprecated `_num_pes` version.

#### EXAMPLES

The following `shmem_n_pes` example is for C/C++ programs:

```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
    int npes;

    shmem_init();

    npes = shmem_n_pes();

    if (shmem_my_pe() == 0) {
        printf("Number of PEs executing this program is: %d\n", npes);
    }

    return 0;
}
```

### 8.1.4 SHMEM\_FINALIZE

A collective operation that releases resources used by the OpenSHMEM library. This only terminates the OpenSHMEM portion of a program, not the entire program.

#### SYNOPSIS

##### C/C++:

```
void shmem_finalize(void);
```

##### FORTRAN:

```
CALL SHMEM_FINALIZE
```

#### DESCRIPTION

##### Arguments

None.

##### API description

*shmem\_finalize* is a collective operation that ends the OpenSHMEM portion of a program previously initialized by *shmem\_init* and releases resources used by the OpenSHMEM library. This collective operation requires all PEs to participate in the call. There is an implicit global barrier in *shmem\_finalize* so that pending communication is completed, and no resources can be released until all PEs have entered *shmem\_finalize*. *shmem\_finalize* must be the last OpenSHMEM library call encountered in the OpenSHMEM portion of a program. A call to *shmem\_finalize* will release any resources initialized by a corresponding call to *shmem\_init*. All processes and threads that represent the PEs will still exist after the call to *shmem\_finalize* returns, but they will no longer have access to any resources that have been released.

##### Return Values

None.

##### Notes

*shmem\_finalize* releases all resources used by the OpenSHMEM library including the symmetric memory heap and pointers initiated by *shmem\_ptr*. This collective operation requires all PEs to participate in the call, not just a subset of the PEs. The non-OpenSHMEM portion of a program may continue after a call to *shmem\_finalize* by all PEs. There is an implicit *shmem\_finalize* at the end of main, so that having an explicit call to *shmem\_finalize* is optional. However, an explicit *shmem\_finalize* may be required as an entry point for wrappers used by profiling or other tools that need to perform their own finalization.

#### EXAMPLES

The following finalize example is for C/C++ programs:

```
#include <stdio.h>
#include <shmem.h>

long x = 10101;

int main(void) {
    int me, npes;
    long y = -1;

    shmem_init();

    me = shmem_my_pe();
    npes = shmem_n_pes();
```

```

    if (me == 0)
        y = shmem_long_g(&x, npes-1);

    printf("%d: y = %ld\n", me, y);

    shmem_finalize();
    return 0;
}

```

### 8.1.5 SHMEM\_GLOBAL\_EXIT

A routine that allows any PE to force termination of an entire program.

#### SYNOPSIS

##### C/C++:

```
void shmem_global_exit(int status);
```

##### FORTRAN:

```
INTEGER STATUS
```

```
CALL SHMEM_GLOBAL_EXIT(status)
```

#### DESCRIPTION

##### Arguments

<b>IN</b>	<i>status</i>	The exit status from the main program.
-----------	---------------	--

##### API description

*shmem\_global\_exit* is a non-collective routine that allows any one PE to force termination of an OpenSHMEM program for all PEs, passing an exit status to the execution environment. This routine terminates the entire program, not just the OpenSHMEM portion. When any PE calls *shmem\_global\_exit*, it results in the immediate notification to all PEs to terminate. *shmem\_global\_exit* flushes I/O and releases resources in accordance with C/C++/Fortran language requirements for normal program termination. If more than one PE calls *shmem\_global\_exit*, then the exit status returned to the environment shall be one of the values passed to *shmem\_global\_exit* as the status argument. There is no return to the caller of *shmem\_global\_exit*; control is returned from the OpenSHMEM program to the execution environment for all PEs.

##### Return Values

None.

##### Notes

*shmem\_global\_exit* may be used in situations where one or more PEs have determined that the program has completed and/or should terminate early. Accordingly, the integer status argument can be used to pass any information about the nature of the exit, e.g. an encountered error or a found solution. Since *shmem\_global\_exit* is a non-collective routine, there is no implied synchronization, and all PEs must terminate regardless of their current execution state. While I/O must be flushed for standard language I/O calls from C/C++/Fortran, it is implementation dependent as to how I/O done by other means (e.g. third party I/O libraries) is handled. Similarly, resources are released according to C/C++/Fortran standard language requirements, but this may not include all resources allocated for the OpenSHMEM program. However, a quality implementation will make a best effort to flush all I/O and clean up all resources.

#### EXAMPLES

```

1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <shmem.h>
4
5      int
6      main(void)
7      {
8          int me, npes;
9
10         shmem_init();
11
12         me = shmem_my_pe();
13         npes = shmem_n_pes();
14
15         if (me == 0) {
16             FILE *fp = fopen("input.txt", "r");
17
18             if (fp == NULL) { /* Input file required by program is not available */
19                 shmem_global_exit(EXIT_FAILURE);
20             }
21
22             /* do something with the file */
23
24             fclose(fp);
25         }
26
27         return 0;
28     }

```

### 8.1.6 SHMEM\_PE\_ACCESSIBLE

Determines whether a PE is accessible via OpenSHMEM's data transfer routines.

#### SYNOPSIS

##### C/C++:

```
int shmem_pe_accessible(int pe);
```

##### FORTRAN:

```
LOGICAL LOG, SHMEM_PE_ACCESSIBLE
```

```
INTEGER pe
```

```
LOG = SHMEM_PE_ACCESSIBLE(pe)
```

#### DESCRIPTION

##### Arguments

<b>IN</b>	<i>pe</i>	Specific PE to be checked for accessibility from the local PE.
-----------	-----------	--

##### API description

*shmem\_pe\_accessible* is a query routine that indicates whether a specified PE is accessible via OpenSHMEM from the local PE. The *shmem\_pe\_accessible* routine returns *TRUE* only if the remote PE is a process running from the same executable file as the local PE, indicating that full OpenSHMEM support for symmetric data objects (that reside in the static memory and symmetric heap) is available, otherwise it returns *FALSE*. This routine may be particularly useful for hybrid programming with other communication libraries (such as a MPI) or parallel languages. For example, on SGI Altix series systems, OpenSHMEM is supported across multiple partitioned hosts and InfiniBand connected hosts. When running multiple executable MPI programs using OpenSHMEM on an Altix, full OpenSHMEM support is available between processes running from the same executable file. However, OpenSHMEM support between processes of



different executable files is supported only for data objects on the symmetric heap, since static data objects are not symmetric between different executable files.

### Return Values

*C*: The return value is 1 if the specified PE is a valid remote PE for OpenSHMEM routines; otherwise, it is 0.

*Fortran*: The return value is *.TRUE.* if the specified PE is a valid remote PE for OpenSHMEM routines; otherwise, it is *.FALSE.*

### Notes

None.

## 8.1.7 SHMEM\_ADDR\_ACCESSIBLE

Determines whether an address is accessible via OpenSHMEM data transfer routines from the specified remote PE.

### SYNOPSIS

#### C/C++:

```
int shmem_addr_accessible(void *addr, int pe);
```

#### FORTTRAN:

```
LOGICAL LOG, SHMEM_ADDR_ACCESSIBLE
```

```
INTEGER pe
```

```
LOG = SHMEM_ADDR_ACCESSIBLE(addr, pe)
```

### DESCRIPTION

#### Arguments

IN	<i>addr</i>	Data object on the local PE.
IN	<i>pe</i>	Integer id of a remote PE.

#### API description

*shmem\_addr\_accessible* is a query routine that indicates whether a local address is accessible via OpenSHMEM routines from the specified remote PE.

This routine verifies that the data object is symmetric and accessible with respect to a remote PE via OpenSHMEM data transfer routines. The specified address *addr* is a data object on the local PE.

This routine may be particularly useful for hybrid programming with other communication libraries (such as MPI) or parallel languages. For example, in SGI Altix series systems, for multiple executable MPI programs that use OpenSHMEM routines, it is important to note that static memory, such as a *Fortran* common block or *C* global variable, is symmetric between processes running from the same executable file, but is not symmetric between processes running from different executable files. Data allocated from the symmetric heap (*shmem\_malloc* or *shpalloc*) is symmetric across the same or different executable files.

### Return Values

*C/C++*: The return value is 1 if *addr* is a symmetric data object and accessible via OpenSHMEM routines from the specified remote PE; otherwise, it is 0.

*Fortran*: The return value is *.TRUE.* if *addr* is a symmetric data object and accessible via OpenSHMEM routines from the specified remote PE; otherwise, it is *.FALSE.*

**Notes**

None.

**8.1.8 SHMEM\_PTR**

Returns a pointer to a data object on a specified PE.

**SYNOPSIS****C/C++:**

```
void *shmem_ptr(void *dest, int pe);
```

**FORTRAN:**

```
POINTER (PTR, POINTEE)
```

```
INTEGER pe
```

```
PTR = SHMEM_PTR(dest, pe)
```

**DESCRIPTION****Arguments**

<b>IN</b>	<i>dest</i>	The symmetric data object to be referenced.
<b>IN</b>	<i>pe</i>	An integer that indicates the PE number on which <i>dest</i> is to be accessed. If you are using <i>Fortran</i> , it must be a default integer value.

**API description**

*shmem\_ptr* returns an address that may be used to directly reference *dest* on the specified PE. This address can be assigned to a pointer. After that, ordinary loads and stores to this remote address may be performed.

When a sequence of loads (gets) and stores (puts) to a data object on a remote PE does not match the access pattern provided in an OpenSHMEM data transfer routine like *shmem\_put32* or *shmem\_real\_iget*, the *shmem\_ptr* routine can provide an efficient means to accomplish the communication.

**Return Values**

The return value is a non-NULL address of the *dest* data object when it is accessible using memory loads and stores in addition to OpenSHMEM operations. Otherwise, a NULL address is returned.

**Notes**

When calling *shmem\_ptr*, *dest* is the address of the referenced symmetric data object on the calling PE.

**EXAMPLES**

This *Fortran* program calls *shmem\_ptr* and then PE 0 writes to the *BIGD* array on PE 1:

```
PROGRAM REMOTEWRITE
  INCLUDE 'shmem.fh'

  INTEGER BIGD(100)
  SAVE BIGD

  INTEGER POINTEE(*)
  POINTER (PTR,POINTEE)

  CALL SHMEM_INIT()
```

```

IF (SHMEM_MY_PE() .EQ. 0) THEN
  ! initialize PE 1's BIGD array
  PTR = SHMEM_PTR(BIGD, 1)      ! get address of PE 1's BIGD
                                ! array
  DO I=1,100
    POINTEE(I) = I
  ENDDO
ENDIF

CALL SHMEM_BARRIER_ALL

IF (SHMEM_MY_PE() .EQ. 1) THEN
  PRINT*, 'BIGD on PE 1 is: '
  PRINT*, BIGD
ENDIF
END

```

This is the equivalent program written in C:

```

#include <stdio.h>
#include <shmem.h>

int main(void)
{
  static int bigd[100];
  int *ptr;
  int i;

  shmem_init();

  if (shmem_my_pe() == 0) {
    /* initialize PE 1's bigd array */
    ptr = shmem_ptr(bigd, 1);
    if (ptr == NULL)
      printf("can't use pointer to directly access PE 1's array\n");
    else
      for (i=0; i<100; i++)
        *ptr++ = i+1;
  }

  shmem_barrier_all();

  if (shmem_my_pe() == 1) {
    printf("bigd on PE 1 is:\n");
    for (i=0; i<100; i++)
      printf(" %d\n", bigd[i]);
    printf("\n");
  }
  return 1;
}

```

### 8.1.9 SHMEM\_INFO\_GET\_VERSION

Returns the major and minor version of the library implementation.

#### SYNOPSIS

##### C/C++:

```
void shmem_info_get_version(int *major, int *minor);
```

##### FORTRAN:

```
SHMEM_INFO_GET_VERSION(MAJOR, MINOR)
```

```
INTEGER MAJOR, MINOR
```

**DESCRIPTION****Arguments**

<b>OUT</b>	<i>major</i>	The major version of the OpenSHMEM standard in use.
<b>OUT</b>	<i>minor</i>	The minor version of the OpenSHMEM standard in use.

**API description**

This routine returns the major and minor version of the OpenSHMEM standard in use. For a given library implementation, the major and minor version returned by these calls is consistent with the compile-time constants, SHMEM\_MAJOR\_VERSION and SHMEM\_MINOR\_VERSION, defined in its shmem.h. The valid major version value is 1, and the valid minor version value is 2.

**Return Values**

None.

**Notes**

None.

**8.1.10 SHMEM\_INFO\_GET\_NAME**

This routine returns the vendor defined character string.

**SYNOPSIS****C/C++:**

```
void shmem_info_get_name(char *name);
```

**FORTRAN:**

```
SHMEM_INFO_GET_NAME (NAME)
```

```
CHARACTER * (*) NAME
```

**DESCRIPTION****Arguments**

<b>OUT</b>	<i>name</i>	The vendor defined string.
------------	-------------	----------------------------

**API description**

This routine returns the vendor defined character string of size defined by the constant SHMEM\_MAX\_NAME\_LEN. The program calling this function prepares the memory of size SHMEM\_MAX\_NAME\_LEN, and the implementation copies the string of size at most SHMEM\_MAX\_NAME\_LEN. In C, the string is terminated by a null character. In Fortran, the string of size less than SHMEM\_MAX\_NAME\_LEN is padded with blank characters up to size SHMEM\_MAX\_NAME\_LEN. The implementation copying a string of size greater than SHMEM\_MAX\_NAME\_LEN results in an undefined behavior. Multiple invocations of the routine in an OpenSHMEM program always return the same string. For a given library implementation, the major and minor version returned by these calls is consistent with the compile-time constants defined in its shmem.h.

**Return Values**

None.

**Notes**

None.

### 8.1.11 START\_PES

Called at the beginning of an OpenSHMEM program to initialize the execution environment. (This routine is deprecated and is provided for backwards compatibility. Implementations must include it, and the routines should function properly while notifying the user about deprecation of its use.)

#### SYNOPSIS

**C/C++:**

```
void start_pes(int npes);
```

**FORTTRAN:**

```
CALL START_PES(npes)
```

#### DESCRIPTION

**Arguments**

<b>npes</b>	<i>Unused</i>	Should be set to 0.
-------------	---------------	---------------------

**API description**

The *start\_pes* routine initializes the OpenSHMEM execution environment. An OpenSHMEM program must call *start\_pes* before calling any other OpenSHMEM routine.

**Return Values**

None.

**Notes**

If any other OpenSHMEM call occurs before *start\_pes*, the behavior is undefined. Although it is recommended to set *npes* to 0 for *start\_pes*, this is not mandated. The value is ignored. Calling *start\_pes* more than once has no subsequent effect.

**Notes**

As of OpenSHMEM Specification 1.2 the use of *start\_pes* has been deprecated. Although OpenSHMEM libraries are required to support the call, program users are encouraged to use *shmem\_init* instead.

#### EXAMPLES

This is a simple program that calls *start\_pes*:

```
PROGRAM PUT
INTEGER TARG, SRC, RECEIVER, BAR
COMMON /T/ TARG
PARAMETER (RECEIVER=1)
CALL START_PES(0)

IF (SHMEM_MY_PE() .EQ. 0) THEN
  SRC = 33
  CALL SHMEM_INTEGER_PUT(TARG, SRC, 1, RECEIVER)
ENDIF

CALL SHMEM_BARRIER_ALL          ! SYNCHRONIZES SENDER AND RECEIVER

IF (SHMEM_MY_PE() .EQ. RECEIVER) THEN
  PRINT*, 'PE ', SHMEM_MY_PE(), ' TARG=', TARG, ' (expect 33)'
ENDIF
END
```

## 8.2 Memory Management Routines

OpenSHMEM provides a set of APIs for managing the symmetric heap. The APIs allow one to dynamically allocate, deallocate, reallocate and align symmetric data objects in the symmetric heap, in *C* and *Fortran*.

### 8.2.1 SHMEM\_MALLOC, SHMEM\_FREE, SHMEM\_REALLOC, SHMEM\_ALIGN

Symmetric heap memory management routines.

#### SYNOPSIS

##### C/C++:

```
void *shmem_malloc(size_t size);
void shmem_free(void *ptr);
void *shmem_realloc(void *ptr, size_t size);
void *shmem_align(size_t alignment, size_t size);
```

#### DESCRIPTION

##### Arguments

<b>IN</b>	<i>size</i>	The size, in bytes, of a block to be allocated from the symmetric heap. This argument is of type <i>size_t</i>
<b>IN</b>	<i>ptr</i>	Points to a block within the symmetric heap.
<b>IN</b>	<i>alignment</i>	Byte alignment of the block allocated from the symmetric heap.

##### API description

The *shmem\_malloc* routine returns a pointer to a block of at least *size* bytes suitably aligned for any use. This space is allocated from the symmetric heap (in contrast to *malloc*, which allocates from the private heap).

The *shmem\_align* routine allocates a block in the symmetric heap that has a byte alignment specified by the alignment argument.

The *shmem\_free* routine causes the block to which *ptr* points to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs.

The *shmem\_realloc* routine changes the size of the block to which *ptr* points to the size (in bytes) specified by *size*. The contents of the block are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the block is indeterminate. If *ptr* is a *NULL* pointer, the *shmem\_realloc* routine behaves like the *shmem\_malloc* routine for the specified size. If *size* is 0 and *ptr* is not a *NULL* pointer, the block to which it points is freed. If the space cannot be allocated, the block to which *ptr* points is unchanged.

The *shmem\_malloc*, *shmem\_free*, and *shmem\_realloc* routines are provided so that multiple PEs in a program can allocate symmetric, remotely accessible memory blocks. These memory blocks can then be used with OpenSHMEM communication routines. Each of these routines call the *shmem\_barrier\_all* routine before returning; this ensures that all PEs participate in the memory allocation, and that the memory on other PEs can be used as soon as the local PE returns. The user is responsible for calling these routines with identical argument(s) on all PEs; if differing *size* arguments are used, the behavior of the call and any subsequent OpenSHMEM calls becomes undefined.

##### Return Values

The *shmem\_malloc* routine returns a pointer to the allocated space; otherwise, it returns a *NULL* pointer.

The *shmem\_free* routine returns no value.

The *shmem\_realloc* routine returns a pointer to the allocated space (which may have moved); otherwise, it returns a null pointer.

The *shmem\_align* routine returns an aligned pointer to the allocated space; otherwise, it returns a *NULL* pointer.

### Notes

As of Specification 1.2 the use of *shmalloc*, *shmemalign*, *shfree*, and *shrealloc* has been deprecated. Although OpenSHMEM libraries are required to support the calls, program users are encouraged to use *shmem\_malloc*, *shmem\_align*, *shmem\_free*, and *shmem\_realloc* instead. The behavior and signature of the routines remains unchanged from the deprecated versions.

The total size of the symmetric heap is determined at job startup. One can adjust the size of the heap using the *SMA\_SYMMETRIC\_SIZE* environment variable (where available).

The *shmem\_malloc*, *shmem\_free*, and *shmem\_realloc* routines differ from the private heap allocation routines in that all PEs in a program must call them (a barrier is used to ensure this).

### Note to implementors

The symmetric heap allocation functions always return a pointer to corresponding symmetric objects across all PEs. The OpenSHMEM specification does not require that the virtual addresses are equal across all PEs. Nevertheless, the implementation must avoid costly address translation operations in the communication path, including order *N* (where *N* is the number of PEs) memory translation tables. In order to avoid address translations, the implementation may re-map the allocated block of memory based on agreed virtual address. Additionally, some operating systems provide an option to disable virtual address randomization, which enables predictable allocation of virtual memory addresses.

## 8.2.2 SHPALLOC

Allocates a block of memory from the symmetric heap.

### SYNOPSIS

#### FORTRAN:

```
POINTER (addr, A(1))
INTEGER length, errcode, abort
CALL SHPALLOC(addr, length, errcode, abort)
```

### DESCRIPTION

#### Arguments

<b>OUT</b>	<i>addr</i>	First word address of the allocated block.
<b>IN</b>	<i>length</i>	Number of words of memory requested. One word is 32 bits.
<b>OUT</b>	<i>errcode</i>	Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error.
<b>IN</b>	<i>abort</i>	Abort code; nonzero requests abort on error; 0 requests an error code.

#### API description

*SHPALLOC* allocates a block of memory from the program's symmetric heap that is greater than or equal to the size requested. To maintain symmetric heap consistency, all PEs in an program must call *SHPALLOC* with the same value of *length*; if any PEs are missing, the program will hang.

By using the *Fortran POINTER* mechanism in the following manner, you can use array *A* to refer to the block allocated by *SHPALLOC*: *POINTER* (*addr*, *A*())

#### Return Values

Error Code	Condition
-1	Length is not an integer greater than 0
-2	No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).

**Notes**

The total size of the symmetric heap is determined at job startup. One may adjust the size of the heap using the `SMA_SYMMETRIC_SIZE` environment variable (if available).

**Note to implementors**

The symmetric heap allocation functions always return a pointer to corresponding symmetric objects across all PEs. The OpenSHMEM specification does not require that the virtual addresses are equal across all PEs. Nevertheless, the implementation must avoid costly address translation operations in the communication path, including order  $N$  (where  $N$  is the number of PEs) memory translation tables. In order to avoid address translations, the implementation may re-map the allocated block of memory based on agreed virtual address. Additionally, some operating systems provide an option to disable virtual address randomization, which enables predictable allocation of virtual memory addresses.

**8.2.3 SHPCLMOVE**

Extends a symmetric heap block or copies the contents of the block into a larger block.

**SYNOPSIS****FORTRAN:**

```

POINTER (addr, A(1))
INTEGER length, status, abort
CALL SHPCLMOVE (addr, length, status, abort)

```

**DESCRIPTION****Arguments**

<b>INOUT</b>	<i>addr</i>	On entry, first word address of the block to change; on exit, the new address of the block if it was moved.
<b>IN</b>	<i>length</i>	Requested new total length in words. One word is 32 bits.
<b>OUT</b>	<i>status</i>	Status is 0 if the block was extended in place, 1 if it was moved, and a negative integer for the type of error detected.
<b>IN</b>	<i>abort</i>	Abort code. Nonzero requests abort on error; 0 requests an error code.

**API description**

The `SHPCLMOVE` routine either extends a symmetric heap block if the block is followed by a large enough free block or copies the contents of the existing block to a larger block and returns a status code indicating that the block was moved. This routine also can reduce the size of a block if the new length is less than the old length. All PEs in a program must call `SHPCLMOVE` with the same value of *addr* to maintain symmetric heap consistency; if any PEs are missing, the program hangs.

**Return Values**

Error Code	Condition
-1	Length is not an integer greater than 0
-2	No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).



-3	Address is outside the bounds of the symmetric heap.
-4	Block is already free.
-5	Address is not at the beginning of a block.

**Notes**

None.

**8.2.4 SHPDEALLC**

Returns a memory block to the symmetric heap.

**SYNOPSIS****FORTRAN:**

```

POINTER (addr, A(1))
INTEGER errcode, abort
CALL SHPDEALLC(addr, errcode, abort)

```

**DESCRIPTION****Arguments**

<b>IN</b>	<i>addr</i>	First word address of the block to deallocate.
<b>OUT</b>	<i>errcode</i>	Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error.
<b>IN</b>	<i>abort</i>	Abort code. Nonzero requests abort on error; 0 requests an error code.

**API description**

SHPDEALLC returns a block of memory (allocated using *SHPALLOC*) to the list of available space in the symmetric heap. To maintain symmetric heap consistency, all PEs in a program must call *SHPDEALLC* with the same value of *addr*; if any PEs are missing, the program hangs.

**Return Values**

Error Code	Condition
-1	Length is not an integer greater than 0
-2	No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).
-3	Address is outside the bounds of the symmetric heap.
-4	Block is already free.
-5	Address is not at the beginning of a block.

**Notes**

None.

**8.3 Remote Memory Access Routines**

The *Remote Memory Access* (RMA) routines described in this section are one-sided communication mechanisms of the OpenSHMEM API. While using these mechanisms, the user is required to provide parameters only on the calling side.

A characteristic of one-sided communication is that it decouples communication from the synchronization. One-sided communication mechanisms transfer the data but do not synchronize the sender of the data with the receiver of the data.

OpenSHMEM RMA routines are all performed on the symmetric objects. The initiator PE of the call is designated as *source*, and the PE in which memory is accessed is designated as *dest*. In the case of the remote update routine, *Put*, the origin is the *source* PE and the destination PE is the *dest* PE. In the case of the remote read routine, *Get*, the origin is the *dest* PE and the destination is the *source* PE.

OpenSHMEM provides three different types of one-sided communication interfaces. *shmem\_put<bits>* interface transfers data in chunks of bits. *shmem\_put32*, for example, copies data to a *dest* PE in chunks of 32 bits. *shmem\_<datatype>\_put* interface copies elements of type *datatype* from a *source* PE to a *dest* PE. For example, *shmem\_integer\_put*, copies elements of type integer from a *source* PE to a *dest* PE. *shmem\_<datatype>\_p* interface is similar to *shmem\_<datatype>\_put* except that it only transfers one element of type *datatype*.

OpenSHMEM provides interfaces for transferring both contiguous and non-contiguous data. The non-contiguous data transfer interfaces are prefixed with “*r*”. *shmem\_<datatype>\_iput* interface, for example, copies strided data elements from the *source* PE to a *dest* PE.

### 8.3.1 SHMEM\_PUT

The put routines provide a method for copying data from a contiguous local data object to a data object on a specified PE.

#### SYNOPSIS

##### C/C++:

```
void shmem_double_put(double *dest, const double *source, size_t nelems, int pe);
void shmem_float_put(float *dest, const float *source, size_t nelems, int pe);
void shmem_int_put(int *dest, const int *source, size_t nelems, int pe);
void shmem_long_put(long *dest, const long *source, size_t nelems, int pe);
void shmem_longdouble_put(long double *dest, const long double *source, size_t nelems, int pe);
void shmem_longlong_put(long long *dest, const long long *source, size_t nelems, int pe);
void shmem_put32(void *dest, const void *source, size_t nelems, int pe);
void shmem_put64(void *dest, const void *source, size_t nelems, int pe);
void shmem_put128(void *dest, const void *source, size_t nelems, int pe);
void shmem_putmem(void *dest, const void *source, size_t nelems, int pe);
void shmem_short_put(short *dest, const short *source, size_t nelems, int pe);
```

##### FORTTRAN:

```
CALL SHMEM_CHARACTER_PUT(dest, source, nelems, pe)
CALL SHMEM_COMPLEX_PUT(dest, source, nelems, pe)
CALL SHMEM_DOUBLE_PUT(dest, source, nelems, pe)
CALL SHMEM_INTEGER_PUT(dest, source, nelems, pe)
CALL SHMEM_LOGICAL_PUT(dest, source, nelems, pe)
CALL SHMEM_PUT4(dest, source, nelems, pe)
CALL SHMEM_PUT8(dest, source, nelems, pe)
CALL SHMEM_PUT32(dest, source, nelems, pe)
CALL SHMEM_PUT64(dest, source, nelems, pe)
CALL SHMEM_PUT128(dest, source, nelems, pe)
CALL SHMEM_PUTMEM(dest, source, nelems, pe)
CALL SHMEM_REAL_PUT(dest, source, nelems, pe)
```

#### DESCRIPTION

**Arguments**

<b>IN</b>	<i>dest</i>	Data object to be updated on the remote PE. This data object must be remotely accessible.
<b>OUT</b>	<i>source</i>	Data object containing the data to be copied.
<b>IN</b>	<i>nelems</i>	Number of elements in the <i>dest</i> and <i>source</i> arrays. <i>nelems</i> must be of type <i>size_t</i> for <i>C</i> . If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.
<b>IN</b>	<i>pe</i>	PE number of the remote PE. <i>pe</i> must be of type integer. If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.

**API description**

The routines return after the data has been copied out of the *source* array on the local PE. The delivery of data words into the data object on the destination PE may occur in any order. Furthermore, two successive put routines may deliver data out of order unless a call to *shmem\_fence* is introduced between the two calls.

The *dest* and *source* data objects must conform to certain typing constraints, which are as follows:

Routine	Data Type of dest and source
<i>shmem_putmem</i>	<i>Fortran</i> : Any noncharacter type. <i>C</i> : Any data type. <i>nelems</i> is scaled in bytes.
<i>shmem_put4</i> , <i>shmem_put32</i>	Any noncharacter type that has a storage size equal to 32 bits.
<i>shmem_put8</i> , <i>shmem_put64</i>	Any noncharacter type that has a storage size equal to 64 bits.
<i>shmem_put128</i>	Any noncharacter type that has a storage size equal to 128 bits.
<i>shmem_double_put</i>	Elements of type double.
<i>shmem_longdouble_put</i>	Elements of type long double.
<i>SHMEM_CHARACTER_PUT</i>	Elements of type character. <i>nelems</i> is the number of characters to transfer. The actual character lengths of the <i>source</i> and <i>dest</i> variables are ignored.
<i>SHMEM_COMPLEX_PUT</i>	Elements of type complex of default size.
<i>SHMEM_DOUBLE_PUT</i>	Elements of type double precision.
<i>SHMEM_INTEGER_PUT</i>	Elements of type integer.
<i>SHMEM_LOGICAL_PUT</i>	Elements of type logical.
<i>SHMEM_REAL_PUT</i>	Elements of type real.

**Return Values**

None.

**Notes**

If you are using *Fortran*, data types must be of default size. For example, a real variable must be declared as *REAL*, *REAL\*4*, or *REAL(KIND=KIND(1.0))*. The Fortran API routine *SHMEM\_PUT* has been deprecated, and either *SHMEM\_PUT8* or *SHMEM\_PUT64* should be used in its place.

**EXAMPLES**

The following *shmem\_put* example is for *C/C++* programs:

```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
    long source[10] = { 1, 2, 3, 4, 5,
```

```

1         6, 7, 8, 9, 10 };
2     static long dest[10];
3     shmem_init();
4     if (shmem_my_pe() == 0) {
5         /* put 10 words into dest on PE 1 */
6         shmem_long_put(dest, source, 10, 1);
7     }
8     shmem_barrier_all(); /* sync sender and receiver */
9     printf("dest[0] on PE %d is %ld\n", shmem_my_pe(), dest[0]);
10    return 1;
11 }

```

### 8.3.2 SHMEM\_P

Copies one data item to a remote PE.

#### SYNOPSIS

##### C/C++:

```

17 void shmем_char_p(char *addr, char value, int pe);
18 void shmем_short_p(short *addr, short value, int pe);
19 void shmем_int_p(int *addr, int value, int pe);
20 void shmем_long_p(long *addr, long value, int pe);
21 void shmем_longlong_p(long long *addr, long long value, int pe);
22 void shmем_float_p(float *addr, float value, int pe);
23 void shmем_double_p(double *addr, double value, int pe);
24 void shmем_longdouble_p(long double *addr, long double value, int pe);

```

#### DESCRIPTION

##### Arguments

IN	<i>addr</i>	The remotely accessible array element or scalar data object which will receive the data on the remote PE.
IN	<i>value</i>	The value to be transferred to <i>addr</i> on the remote PE.
IN	<i>pe</i>	The number of the remote PE.

##### API description

These routines provide a very low latency put capability for single elements of most basic types.

As with *shmем\_put*, these routines start the remote transfer and may return before the data is delivered to the remote PE. Use *shmем\_quiet* to force completion of all remote *Put* transfers.

##### Return Values

None.

##### Notes

None.

#### EXAMPLES

The following example uses *shmем\_double\_p* in a C program.

```

47 #include <stdio.h>
48 #include <math.h>
49 #include <shmем.h>

```

```

static const double e = 2.71828182;
static const double epsilon = 0.00000001;

int main(void)
{
    double *f;
    int me;

    shmem_init();
    me = shmem_my_pe();
    f = (double *) shmem_malloc(sizeof (*f));

    *f = 3.1415927;
    shmem_barrier_all();

    if (me == 0)
        shmem_double_p(f, e, 1);

    shmem_barrier_all();
    if (me == 1)
        printf("%s\n", (fabs (*f - e) < epsilon) ? "OK" : "FAIL");

    return 0;
}

```

### 8.3.3 SHMEM\_IPUT

Copies strided data to a specified PE.

#### SYNOPSIS

##### C/C++:

```

void shmem_double_iput(double *dest, const double *source, ptrdiff_t dst, ptrdiff_t sst,
    size_t nelems, int pe);
void shmem_float_iput(float *dest, const float *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int pe);
void shmem_int_iput(int *dest, const int *source, ptrdiff_t dst, ptrdiff_t sst, size_t nelems
    , int pe);
void shmem_iput32(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t nelems
    , int pe);
void shmem_iput64(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t nelems
    , int pe);
void shmem_iput128(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int pe);
void shmem_long_iput(long *dest, const long *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int pe);
void shmem_longdouble_iput(long double *dest, const long double *source, ptrdiff_t dst,
    ptrdiff_t sst, size_t nelems, int pe);
void shmem_longlong_iput(long long *dest, const long long *source, ptrdiff_t dst, ptrdiff_t
    sst, size_t nelems, int pe);
void shmem_short_iput(short *dest, const short *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int pe);

```

##### FORTRAN:

```

INTEGER dst, sst, nelems, pe
CALL SHMEM_COMPLEX_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_DOUBLE_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_INTEGER_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT4(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT8(dest, source, dst, sst, nelems, pe)

```

```

1  CALL SHMEM_IPUT32(dest, source, dst, sst, nelems, pe)
2  CALL SHMEM_IPUT64(dest, source, dst, sst, nelems, pe)
3  CALL SHMEM_IPUT128(dest, source, dst, sst, nelems, pe)
4  CALL SHMEM_LOGICAL_IPUT(dest, source, dst, sst, nelems, pe)
5  CALL SHMEM_REAL_IPUT(dest, source, dst, sst, nelems, pe)

```

## DESCRIPTION

### Arguments

<b>OUT</b>	<i>dest</i>	Array to be updated on the remote PE. This data object must be remotely accessible.
<b>IN</b>	<i>source</i>	Array containing the data to be copied.
<b>IN</b>	<i>dst</i>	The stride between consecutive elements of the <i>dest</i> array. The stride is scaled by the element size of the <i>dest</i> array. A value of <i>1</i> indicates contiguous data. <i>dst</i> must be of type <i>ptrdiff_t</i> . If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>sst</i>	The stride between consecutive elements of the <i>source</i> array. The stride is scaled by the element size of the <i>source</i> array. A value of <i>1</i> indicates contiguous data. <i>sst</i> must be of type <i>ptrdiff_t</i> . If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>nelems</i>	Number of elements in the <i>dest</i> and <i>source</i> arrays. <i>nelems</i> must be of type <i>size_t</i> for <i>C</i> . If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.
<b>IN</b>	<i>pe</i>	PE number of the remote PE. <i>pe</i> must be of type integer. If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.

### API description

The *iput* routines provide a method for copying strided data elements (specified by *sst*) of an array from a *source* array on the local PE to locations specified by stride *dst* on a *dest* array on specified remote PE. Both strides, *dst* and *sst* must be greater than or equal to *1*. The routines return when the data has been copied out of the *source* array on the local PE but not necessarily before the data has been delivered to the remote data object.

The *dest* and *source* data objects must conform to typing constraints, which are as follows:

Routine	Data Type of dest and source
<code>shmem_iput32</code> , <code>shmem_iput4</code>	Any noncharacter type that has a storage size equal to 32 bits.
<code>shmem_iput64</code> , <code>shmem_iput8</code>	Any noncharacter type that has a storage size equal to 64 bits.
<code>shmem_iput128</code>	Any noncharacter type that has a storage size equal to 128 bits.
<code>shmem_short_iput</code>	Elements of type short.
<code>shmem_int_iput</code>	Elements of type int.
<code>shmem_long_iput</code>	Elements of type long.
<code>shmem_longlong_iput</code>	Elements of type long long.
<code>shmem_float_iput</code>	Elements of type float.
<code>shmem_double_iput</code>	Elements of type double.
<code>shmem_longdouble_iput</code>	Elements of type long double.
<code>SHMEM_COMPLEX_IPUT</code>	Elements of type complex of default size.
<code>SHMEM_DOUBLE_IPUT</code>	Elements of type double precision.
<code>SHMEM_INTEGER_IPUT</code>	Elements of type integer.
<code>SHMEM_LOGICAL_IPUT</code>	Elements of type logical.
<code>SHMEM_REAL_IPUT</code>	Elements of type real.

**Return Values**

None.

**Notes**

If you are using *Fortran*, data types must be of default size. For example, a real variable must be declared as *REAL*, *REAL\*4* or *REAL(KIND=KIND(1.0))*. See Introduction for a definition of the term remotely accessible.

**EXAMPLES**

Consider the following *shmem\_short\_iput* example for *C/C++* programs.

```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
    short source[10] = { 1, 2, 3, 4, 5,
                       6, 7, 8, 9, 10 };
    static short dest[10];
    shmem_init();
    if (shmem_my_pe() == 0) {
        /* put 5 words into dest on PE 1 */
        shmem_short_iput(dest, source, 1, 2, 5, 1);
    }
    shmem_barrier_all(); /* sync sender and receiver */
    if (shmem_my_pe() == 1) {
        printf("dest on PE %d is %d %d %d %d %d\n", shmem_my_pe(),
              (int)dest[0], (int)dest[1], (int)dest[2],
              (int)dest[3], (int)dest[4] );
    }
    shmem_barrier_all(); /* sync before exiting */
    return 1;
}
```

**8.3.4 SHMEM\_GET**

Copies data from a specified PE.

**SYNOPSIS****C/C++:**

```
void shmem_double_get(double *dest, const double *source, size_t nelems, int pe);
void shmem_float_get(float *dest, const float *source, size_t nelems, int pe);
void shmem_get32(void *dest, const void *source, size_t nelems, int pe);
void shmem_get64(void *dest, const void *source, size_t nelems, int pe);
void shmem_get128(void *dest, const void *source, size_t nelems, int pe);
void shmem_getmem(void *dest, const void *source, size_t nelems, int pe);
void shmem_int_get(int *dest, const int *source, size_t nelems, int pe);
void shmem_long_get(long *dest, const long *source, size_t nelems, int pe);
void shmem_longdouble_get(long double *dest, const long double *source, size_t nelems, int pe);
);
void shmem_longlong_get(long long *dest, const long long *source, size_t nelems, int pe);
void shmem_short_get(short *dest, const short *source, size_t nelems, int pe);
```

**FORTTRAN:**

```
INTEGER nelems, pe
CALL SHMEM_CHARACTER_GET(dest, source, nelems, pe)
```

```

1  CALL SHMEM_COMPLEX_GET(dest, source, nelems, pe)
2  CALL SHMEM_DOUBLE_GET(dest, source, nelems, pe)
3  CALL SHMEM_GET4(dest, source, nelems, pe)
4  CALL SHMEM_GET8(dest, source, nelems, pe)
5  CALL SHMEM_GET32(dest, source, nelems, pe)
6  CALL SHMEM_GET128(dest, source, nelems, pe)
7  CALL SHMEM_GETMEM(dest, source, nelems, pe)
8  CALL SHMEM_INTEGER_GET(dest, source, nelems, pe)
9  CALL SHMEM_LOGICAL_GET(dest, source, nelems, pe)
10 CALL SHMEM_REAL_GET(dest, source, nelems, pe)

```

## DESCRIPTION

### Arguments

<b>OUT</b>	<i>dest</i>	Local data object to be updated.
<b>IN</b>	<i>source</i>	Data object on the PE identified by <i>pe</i> that contains the data to be copied. This data object must be remotely accessible.
<b>IN</b>	<i>nelems</i>	Number of elements in the <i>dest</i> and <i>source</i> arrays. <i>nelems</i> must be of type <i>size_t</i> for <i>C</i> . If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.
<b>IN</b>	<i>pe</i>	PE number of the remote PE. <i>pe</i> must be of type integer. If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.

### API description

The get routines provide a method for copying a contiguous symmetric data object from a different PE to a contiguous data object on the local PE. The routines return after the data has been delivered to the *dest* array on the local PE.

The *dest* and *source* data objects must conform to typing constraints, which are as follows:

Routine	Data Type of dest and source
<code>shmem_getmem</code>	<i>Fortran</i> : Any noncharacter type. <i>C</i> : Any data type. <i>nelems</i> is scaled in bytes.
<code>shmem_get4</code> , <code>shmem_get32</code>	Any noncharacter type that has a storage size equal to 32 bits.
<code>shmem_get8</code> , <code>shmem_get64</code>	Any noncharacter type that has a storage size equal to 64 bits.
<code>shmem_get128</code>	Any noncharacter type that has a storage size equal to 128 bits.
<code>shmem_short_get</code>	Elements of type short.
<code>shmem_int_get</code>	Elements of type int.
<code>shmem_long_get</code>	Elements of type long.
<code>shmem_longlong_get</code>	Elements of type long long.
<code>shmem_float_get</code>	Elements of type float.
<code>shmem_double_get</code>	Elements of type double.
<code>shmem_longdouble_get</code>	Elements of type long double.
<code>SHMEM_CHARACTER_GET</code>	Elements of type character. <i>nelems</i> is the number of characters to transfer. The actual character lengths of the <i>source</i> and <i>dest</i> variables are ignored.
<code>SHMEM_COMPLEX_GET</code>	Elements of type complex of default size.
<code>SHMEM_DOUBLE_GET</code>	<i>Fortran</i> : Elements of type double precision.
<code>SHMEM_INTEGER_GET</code>	Elements of type integer.
<code>SHMEM_LOGICAL_GET</code>	Elements of type logical.
<code>SHMEM_REAL_GET</code>	Elements of type real.



**Return Values**

None.

**Notes**

See Introduction for a definition of the term remotely accessible. If you are using *Fortran*, data types must be of default size. For example, a real variable must be declared as *REAL*, *REAL\*4*, or *REAL(KIND=KIND(1.0))*.

**EXAMPLES**

Consider this example for *Fortran*.

```

PROGRAM REDUCTION
REAL VALUES, SUM
COMMON /C/ VALUES
REAL WORK
CALL SHMEM_INIT()           ! ALLOW ANY NUMBER OF PES
VALUES = SHMEM_MY_PE()      ! INITIALIZE IT TO SOMETHING
CALL SHMEM_BARRIER_ALL
SUM = 0.0
DO I = 0, SHMEM_N_PES()-1
  CALL SHMEM_REAL_GET(WORK, VALUES, (SHMEM_N_PES()-1), I)
  SUM = SUM + WORK
ENDDO
PRINT*, 'PE ', SHMEM_MY_PE(), ' COMPUTED SUM=', SUM
CALL SHMEM_BARRIER_ALL
END

```

**8.3.5 SHMEM\_G**

Transfers one data item from a remote PE

**SYNOPSIS****C/C++:**

```

char shmem_char_g(char *addr, int pe);
short shmem_short_g(short *addr, int pe);
int shmem_int_g(int *addr, int pe);
long shmem_long_g(long *addr, int pe);
long long shmem_longlong_g(long long *addr, int pe);
float shmem_float_g(float *addr, int pe);
double shmem_double_g(double *addr, int pe);
long double shmem_longdouble_g(long double *addr, int pe);

```

**DESCRIPTION****Arguments**

<b>IN</b>	<i>addr</i>	The remotely accessible array element or scalar data object.
<b>IN</b>	<i>pe</i>	The number of the remote PE on which <i>addr</i> resides.

**API description**

These routines provide a very low latency get capability for single elements of most basic types.

**Return Values**

Returns a single element of type specified in the synopsis.

**Notes**

None.

**EXAMPLES**

The following *shmem\_long\_g* example is for C/C++ programs:

```

7  #include <stdio.h>
8  #include <shmem.h>
9
10 long x = 10101;
11
12 int main(void)
13 {
14     int me, npes;
15     long y = -1;
16
17     shmem_init();
18     me = shmem_my_pe();
19     npes = shmem_n_pes();
20
21     if (me == 0)
22         y = shmem_long_g(&x, npes-1);
23
24     printf("%d: y = %ld\n", me, y);
25
26     return 0;
27 }

```

**8.3.6 SHMEM\_IGET**

Copies strided data from a specified PE.

**SYNOPSIS****C/C++:**

```

30 void shmem_double_iget(double *dest, const double *source, ptrdiff_t dst, ptrdiff_t sst,
31     size_t nelems, int pe);
32 void shmem_float_iget(float *dest, const float *source, ptrdiff_t dst, ptrdiff_t sst, size_t
33     nelems, int pe);
34 void shmem_iget32(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
35     , int pe);
36 void shmem_iget64(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
37     , int pe);
38 void shmem_iget128(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
39     nelems, int pe);
40 void shmem_int_iget(int *dest, const int *source, ptrdiff_t dst, ptrdiff_t sst, size_t
41     , int pe);
42 void shmem_long_iget(long *dest, const long *source, ptrdiff_t dst, ptrdiff_t sst, size_t
43     nelems, int pe);
44 void shmem_longdouble_iget(long double *dest, const long double *source, ptrdiff_t dst,
45     ptrdiff_t sst, size_t nelems, int pe);
46 void shmem_longlong_iget(long long *dest, const long long *source, ptrdiff_t dst, ptrdiff_t
47     sst, size_t nelems, int pe);
48 void shmem_short_iget(short *dest, const short *source, ptrdiff_t dst, ptrdiff_t sst, size_t
49     nelems, int pe);

```

**FORTTRAN:**

```

INTEGER dst, sst, nelems, pe

```

```

CALL SHMEM_COMPLEX_IGET(dest, source, dst, sst, nelems, pe)
CALL SHMEM_DOUBLE_IGET(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IGET4(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IGET8(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IGET32(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IGET64(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IGET128(dest, source, dst, sst, nelems, pe)
CALL SHMEM_INTEGER_IGET(dest, source, dst, sst, nelems, pe)
CALL SHMEM_LOGICAL_IGET(dest, source, dst, sst, nelems, pe)
CALL SHMEM_REAL_IGET(dest, source, dst, sst, nelems, pe)

```

## DESCRIPTION

### Arguments

<b>OUT</b>	<i>dest</i>	Array to be updated on the local PE.
<b>IN</b>	<i>source</i>	Array containing the data to be copied on the remote PE.
<b>IN</b>	<i>dst</i>	The stride between consecutive elements of the <i>dest</i> array. The stride is scaled by the element size of the <i>dest</i> array. A value of <i>1</i> indicates contiguous data. <i>dst</i> must be of type <i>ptrdiff_t</i> . If you are calling from <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>sst</i>	The stride between consecutive elements of the <i>source</i> array. The stride is scaled by the element size of the <i>source</i> array. A value of <i>1</i> indicates contiguous data. <i>sst</i> must be of type <i>ptrdiff_t</i> . If you are calling from <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>nelems</i>	Number of elements in the <i>dest</i> and <i>source</i> arrays. <i>nelems</i> must be of type <i>size_t</i> for <i>C</i> . If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.
<b>IN</b>	<i>pe</i>	PE number of the remote PE. <i>pe</i> must be of type integer. If you are using <i>Fortran</i> , it must be a constant, variable, or array element of default integer type.

### API description

The *iget* routines provide a method for copying strided data elements from a symmetric array from a specified remote PE to strided locations on a local array. The routines return when the data has been copied into the local *dest* array.

The *dest* and *source* data objects must conform to typing constraints, which are as follows:

Routine	Data Type of dest and source
shmem_iget32, shmem_iget4	Any noncharacter type that has a storage size equal to 32 bits.
shmem_iget64, shmem_iget8	Any noncharacter type that has a storage size equal to 64 bits.
shmem_iget128	Any noncharacter type that has a storage size equal to 128 bits.
shmem_short_iget	Elements of type short.
shmem_int_iget	Elements of type int.
shmem_long_iget	Elements of type long.
shmem_longlong_iget	Elements of type long long.
shmem_float_iget	Elements of type float.
shmem_double_iget	Elements of type double.
shmem_longdouble_iget	Elements of type long double.
SHMEM_COMPLEX_IGET	Elements of type complex of default size.
SHMEM_DOUBLE_IGET	<i>Fortran</i> : Elements of type double precision.
SHMEM_INTEGER_IGET	Elements of type integer.

SHMEM_LOGICAL_IGET	Elements of type logical.
SHMEM_REAL_IGET	Elements of type real.

**Return Values**

None.

**Notes**

If you are using *Fortran*, data types must be of default size. For example, a real variable must be declared as *REAL*, *REAL\*4*, or *REAL(KIND=KIND(1.0))*.

**EXAMPLES**

The following example uses *shmem\_logical\_iget* in a *Fortran* program.

```

PROGRAM STRIDELOGICAL
LOGICAL SOURCE(10), DEST(5)
SAVE SOURCE ! SAVE MAKES IT REMOTELY ACCESSIBLE
DATA SOURCE /.T.,.F.,.T.,.F.,.T.,.F.,.T.,.F.,.T.,.F./
DATA DEST / 5*.F. /
CALL SHMEM_INIT()
IF (SHMEM_MY_PE() .EQ. 0) THEN
CALL SHMEM_LOGICAL_IGET(DEST, SOURCE, 1, 2, 5, 1)
PRINT*, 'DEST AFTER SHMEM_LOGICAL_IGET:', DEST
ENDIF
CALL SHMEM_BARRIER_ALL

```

**8.4 Atomic Memory Operations**

*Atomic Memory Operation* (AMO) is a one-sided communication mechanism that combines memory update operations with atomicity guarantees described in Section 4.2. Similar to the RMA routines, described in Section 8.3, the AMOs are performed only on symmetric objects. OpenSHMEM defines the two types of AMO routines:

- The *fetch-and-operate* routines combine memory update and fetch operations in a single atomic operation. The routines return after the data has been fetched and delivered to the local PE.

The *fetch-and-operate* operations include: *SHMEM\_CSWAP*, *SHMEM\_SWAP*, *SHMEM\_FINC*, and *SHMEM\_FADD*.

- The *non-fetch* atomic routines update the remote memory in a single atomic operation. A *non-fetch* atomic routine starts the atomic operation and may return before the operation execution on the remote PE. To force completion for these *non-fetch* atomic routines, *shmem\_quiet*, *shmem\_barrier*, or *shmem\_barrier\_all* can be used by an OpenSHMEM program.

The *non-fetch* operations include: *SHMEM\_INC* and *SHMEM\_ADD*.

**8.4.1 SHMEM\_ADD**

Performs an atomic add operation on a remote symmetric data object.

**SYNOPSIS****C/C++:**

```

void shmem_int_add(int *dest, int value, int pe);
void shmem_long_add(long *dest, long value, int pe);
void shmem_longlong_add(long long *dest, long long value, int pe);

```

**FORTRAN:**

```

INTEGER pe
INTEGER*4 value_i4
CALL SHMEM_INT4_ADD(dest, value_i4, pe)
INTEGER*8 value_i8
CALL SHMEM_INT8_ADD(dest, value_i8, pe)

```

## DESCRIPTION

### Arguments

<b>OUT</b>	<i>dest</i>	The remotely accessible integer data object to be updated on the remote PE. If you are using <i>C/C++</i> , the type of <i>dest</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>value</i>	The value to be atomically added to <i>dest</i> . If you are using <i>C/C++</i> , the type of <i>value</i> should match that implied in the SYNOPSIS section. If you are using <i>Fortran</i> , it must be of type integer with an element size of <i>dest</i> .
<b>IN</b>	<i>pe</i>	An integer that indicates the PE number upon which <i>dest</i> is to be updated. If you are using <i>Fortran</i> , it must be a default integer value.

### API description

The *shmem\_add* routine performs an atomic add operation. It adds *value* to *dest* on PE *pe* and atomically updates the *dest* without returning the value.

If you are using *Fortran*, *dest* must be of the following type:

Routine	Data Type of dest and source
SHMEM_INT4_ADD	4-byte integer
SHMEM_INT8_ADD	8-byte integer

### Return Values

None.

### Notes

The term remotely accessible is defined in the Introduction.

## EXAMPLES

```

#include <stdio.h>
#include <shmem.h>

int main(void)
{
    int me, old;
    static int dst;

    shmem_init();
    me = shmem_my_pe();

    old = -1;
    dst = 22;
    shmem_barrier_all();

    if (me == 1){

```

```

1     old = shmem_int_fadd(&dst, 44, 0);
2     }
3     shmem_barrier_all();
4     printf("%d: old = %d, dst = %d\n", me, old, dst);
5     return 0;
6 }

```

## 8.4.2 SHMEM\_CSWAP

Performs an atomic conditional swap on a remote data object.

### SYNOPSIS

#### C/C++:

```

13 int shmem_int_cswap(int *dest, int cond, int value, int pe);
14 long shmem_long_cswap(long *dest, long cond, long value, int pe);
15 long shmem_longlong_cswap(long long *dest, long long cond, long long value, int pe);

```

#### FORTRAN:

```

16 INTEGER pe
17
18 INTEGER*4 SHMEM_INT4_CSWAP, cond_i4, value_i4, ires_i4
19 ires_i4 = SHMEM_INT4_CSWAP(dest, cond_i4, value_i4, pe)
20 INTEGER*8 SHMEM_INT8_CSWAP, cond_i8, value_i8, ires_i8
21 ires_i8 = SHMEM_INT8_CSWAP(dest, cond_i8, value_i8, pe)

```

### DESCRIPTION

#### Arguments

<b>OUT</b>	<i>dest</i>	The remotely accessible integer data object to be updated on the remote PE.
<b>IN</b>	<i>cond</i>	<i>cond</i> is compared to the remote <i>dest</i> value. If <i>cond</i> and the remote <i>dest</i> are equal, then <i>value</i> is swapped into the remote <i>dest</i> . Otherwise, the remote <i>dest</i> is unchanged. In either case, the old value of the remote <i>dest</i> is returned as the routine return value. <i>cond</i> must be of the same data type as <i>dest</i> .
<b>IN</b>	<i>value</i>	The value to be atomically written to the remote PE. <i>value</i> must be the same data type as <i>dest</i> .
<b>IN</b>	<i>pe</i>	An integer that indicates the PE number upon which <i>dest</i> is to be updated. If you are using <i>Fortran</i> , it must be a default integer value.

#### API description

The conditional swap routines conditionally update a *dest* data object on the specified PE and return the prior contents of the data object in one atomic operation.

The *dest* and source data objects must conform to certain typing constraints, which are as follows:

Routine	Data Type of dest and source
SHMEM_INT4_CSWAP	4-byte integer.
SHMEM_INT8_CSWAP	8-byte integer.

#### Return Values

The contents that had been in the *dest* data object on the remote PE prior to the conditional swap. Data type is the same as the *dest* data type.

**Notes**

None.

**EXAMPLES**

The following call ensures that the first PE to execute the conditional swap will successfully write its PE number to *race\_winner* on PE 0.

```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
    static int race_winner = -1;
    int oldval;
    shmem_init();
    oldval = shmem_int_cswap(&race_winner, -1, shmem_my_pe(), 0);
    if(oldval == -1) printf("pe %d was first\n", shmem_my_pe());
    return 1;
}
```

**8.4.3 SHMEM\_SWAP**

Performs an atomic swap to a remote data object.

**SYNOPSIS****C/C++:**

```
double shmem_double_swap(double *dest, double value, int pe);
float shmem_float_swap(float *dest, float value, int pe);
int shmem_int_swap(int *dest, int value, int pe);
long shmem_long_swap(long *dest, long value, int pe);
long long shmem_longlong_swap(long long *dest, long long value, int pe);
long shmem_swap(long *dest, long value, int pe);
```

**FORTRAN:**

```
INTEGER SHMEM_SWAP, value, pe
ires = SHMEM_SWAP(dest, value, pe)
INTEGER*4 SHMEM_INT4_SWAP, value_i4, ires_i4
ires_i4 = SHMEM_INT4_SWAP(dest, value_i4, pe)
INTEGER*8 SHMEM_INT8_SWAP, value_i8, ires_i8
ires_i8 = SHMEM_INT8_SWAP(dest, value_i8, pe)
REAL*4 SHMEM_REAL4_SWAP, value_r4, res_r4
res_r4 = SHMEM_REAL4_SWAP(dest, value_r4, pe)
REAL*8 SHMEM_REAL8_SWAP, value_r8, res_r8
res_r8 = SHMEM_REAL8_SWAP(dest, value_r8, pe)
```

**DESCRIPTION****Arguments**

<b>OUT</b>	<i>dest</i>	The remotely accessible integer data object to be updated on the remote PE. If you are using C/C++, the type of <i>dest</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>value</i>	The value to be atomically written to the remote PE. <i>value</i> is the same type as <i>dest</i> .

**IN** *pe* An integer that indicates the PE number on which *dest* is to be updated. If you are using *Fortran*, it must be a default integer value.

### API description

*shmem\_swap* performs an atomic swap operation. It writes *value* into *dest* on PE and returns the previous contents of *dest* as an atomic operation.

If you are using *Fortran*, *dest* must be of the following type:

Routine	Data Type of dest and source
SHMEM_SWAP	Integer of default kind
SHMEM_INT4_SWAP	4-byte integer
SHMEM_INT8_SWAP	8-byte integer
SHMEM_REAL4_SWAP	4-byte real
SHMEM_REAL8_SWAP	8-byte real

### Return Values

The content that had been at the *dest* address on the remote PE prior to the swap is returned.

### Notes

None.

## EXAMPLES

The example below swap values between odd numbered PEs and their right (modulo) neighbor and outputs the result of swap.

```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
    long *dest;
    int me, npes;
    long swapped_val, new_val;

    shmem_init();
    me = shmem_my_pe();
    npes = shmem_n_pes();
    dest = (long *) shmem_malloc(sizeof(*dest));
    *dest = me;
    shmem_barrier_all();
    new_val = me;
    if (me & 1){
        swapped_val = shmem_long_swap(dest, new_val, (me + 1) % npes);
        printf("%d: dest = %ld, swapped = %ld\n", me, *dest, swapped_val);
    }
    shmem_free(dest);
    return 0;
}
```

### 8.4.4 SHMEM\_FINC

Performs an atomic fetch-and-increment operation on a remote data object.

### SYNOPSIS

**C/C++:**



```

int shmem_int_finc(int *dest, int pe);
long shmem_long_finc(long *dest, int pe);
long long shmem_longlong_finc(long long *dest, int pe);

```

**FORTTRAN:**

```

INTEGER pe
INTEGER*4 SHMEM_INT4_FINC, ires_i4
ires_i4 = SHMEM_INT4_FINC(dest, pe)
INTEGER*8 SHMEM_INT8_FINC, ires_i8
ires_i8 = SHMEM_INT8_FINC(dest, pe)

```

**DESCRIPTION****Arguments**

<b>IN</b>	<i>dest</i>	The remotely accessible integer data object to be updated on the remote PE. The type of <i>dest</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>pe</i>	An integer that indicates the PE number on which <i>dest</i> is to be updated. If you are using <i>Fortran</i> , it must be a default integer value.

**API description**

These routines perform a fetch-and-increment operation. The *dest* on PE *pe* is increased by one and the routine returns the previous contents of *dest* as an atomic operation.

If you are using *Fortran*, *dest* must be of the following type:

Routine	Data Type of dest and source
SHMEM_INT4_FINC	4-byte integer
SHMEM_INT8_FINC	8-byte integer

**Return Values**

The contents that had been at the *dest* address on the remote PE prior to the increment. The data type of the return value is the same as the *dest*.

**Notes**

None.

**EXAMPLES**

The following *shmem\_finc* example is for C/C++ programs:

```

#include <stdio.h>
#include <shmem.h>

int dst;

int main(void)
{
    int me;
    int old;

    shmem_init();
    me = shmem_my_pe();

    old = -1;
    dst = 22;

```

```

1     shmem_barrier_all();
2
3     if (me == 0)
4         old = shmem_int_finc(&dst, 1);
5
6     shmem_barrier_all();
7     printf("%d: old = %d, dst = %d\n", me, old, dst);
8     return 0;
9 }

```

### 8.4.5 SHMEM\_INC

Performs an atomic increment operation on a remote data object.

#### SYNOPSIS

##### C/C++:

```

void shmem_int_inc(int *dest, int pe);
void shmem_long_inc(long *dest, int pe);
void shmem_longlong_inc(long long *dest, int pe);

```

##### FORTRAN:

```

INTEGER pe
CALL SHMEM_INT4_INC(dest, pe)
CALL SHMEM_INT8_INC(dest, pe)

```

#### DESCRIPTION

##### Arguments

<b>IN</b>	<i>dest</i>	The remotely accessible integer data object to be updated on the remote PE. The type of <i>dest</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>pe</i>	An integer that indicates the PE number on which <i>dest</i> is to be updated. If you are using <i>Fortran</i> , it must be a default integer value.

##### API description

These routines perform an atomic increment operation on the *dest* data object on PE.

If you are using *Fortran*, *dest* must be of the following type:

Routine	Data Type of dest and source
SHMEM_INT4_INC	4-byte integer
SHMEM_INT8_INC	8-byte integer

##### Return Values

None.

##### Notes

The term remotely accessible is defined in the Introduction.

#### EXAMPLES

The following *shmem\_int\_inc* example is for C/C++ programs:

```

#include <stdio.h>
#include <shmem.h>

int dst;

int main(void)
{
    int me;

    shmem_init();
    me = shmem_my_pe();

    dst = 74;
    shmem_barrier_all();

    if (me == 0)
        shmem_int_inc(&dst, 1);
    shmem_barrier_all();

    printf("%d: dst = %d\n", me, dst);
    return 0;
}

```

#### 8.4.6 SHMEM\_FADD

Performs an atomic fetch-and-add operation on a remote data object.

#### SYNOPSIS

##### C/C++:

```

int shmem_int_fadd(int *dest, int value, int pe);
long shmem_long_fadd(long *dest, long value, int pe);
long long shmem_longlong_fadd(long long *dest, long long value, int pe);

```

##### FORTRAN:

```

INTEGER pe
INTEGER*4 SHMEM_INT4_FADD, ires_i4, value_i4
ires_i4 = SHMEM_INT4_FADD(dest, value_i4, pe)
INTEGER*8 SHMEM_INT8_FADD, ires_i8, value_i8
ires_i8 = SHMEM_INT8_FADD(dest, value_i8, pe)

```

#### DESCRIPTION

##### Arguments

<b>OUT</b>	<i>dest</i>	The remotely accessible integer data object to be updated on the remote PE. The type of <i>dest</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>value</i>	The value to be atomically added to <i>dest</i> . The type of <i>value</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>pe</i>	An integer that indicates the PE number on which <i>dest</i> is to be updated. If you are using <i>Fortran</i> , it must be a default integer value.

##### API description

*shmem\_fadd* routines perform an atomic fetch-and-add operation. An atomic fetch-and-add operation fetches the old *dest* and adds *value* to *dest* without the possibility of another atomic operation on the *dest* between the time of the fetch and the update. These routines add *value* to *dest* on *pe* and return the previous contents of *dest* as an atomic operation.

If you are using *Fortran*, *dest* must be of the following type:

Routine	Data Type of dest and source
SHMEM_INT4_FADD	4-byte integer
SHMEM_INT8_FADD	8-byte integer

### Return Values

The contents that had been at the *dest* address on the remote PE prior to the atomic addition operation. The data type of the return value is the same as the *dest*.

### Notes

None.

## EXAMPLES

The following *shmem\_fadd* example is for *C/C++* programs:

```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
    int me, old;
    static int dst;

    shmem_init();
    me = shmem_my_pe();

    old = -1;
    dst = 22;
    shmem_barrier_all();

    if (me == 1) {
        old = shmem_int_fadd(&dst, 44, 0);
    }
    shmem_barrier_all();
    printf("%d: old = %d, dst = %d\n", me, old, dst);
    return 0;
}
```

## 8.5 Collective Routines

*Collective routines* are defined as communication or synchronization operations on a group of PEs called an *Active set*. The collective routines require all PEs in the *Active set* to simultaneously call the routine. A PE that is not part of the *Active set* calling the collective routines results in an undefined behavior. All collective routines have an *Active set* as an input parameter except *SHMEM\_BARRIER\_ALL*. The *SHMEM\_BARRIER\_ALL* is called by all PEs of the OpenSHMEM program.

The *Active set* is defined by the arguments *PE\_start*, *logPE\_stride*, and *PE\_size*. *PE\_start* is the starting PE number, a log (base 2) of *logPE\_stride* is the stride between PEs, and *PE\_size* is the number of PEs participating in the *Active set*. All PEs participating in the collective routines provide the same values for these arguments.

Another argument important to collective routines is *pSync*, which is a symmetric work array. All PEs participating in a collective must pass the same *pSync* array. On completion of a collective call, the *pSync* is restored to its original contents. The user is permitted to reuse a *pSync* array if all previous collective routines using the *pSync* array have been completed by all participating PEs. One can use a synchronization collective routine such as *SHMEM\_BARRIER*

to ensure completion of previous collective routines. The *shmem\_barrier* routine allows the same *pSync* array to be used on consecutive calls as long as the PE *Active set* does not change.

All collective routines defined in the specification are blocking. The collective routines return on completion. The collective routines defined in the OpenSHMEM specification are:

*SHMEM\_BROADCAST*

*SHMEM\_BARRIER*

*SHMEM\_BARRIER\_ALL*

*SHMEM\_COLLECT*

*Reduction Operations*

### 8.5.1 SHMEM\_BARRIER\_ALL

Registers the arrival of a PE at a barrier and suspends PE execution until all other PEs arrive at the barrier and all local and remote memory updates are completed.

#### SYNOPSIS

**C/C++:**

```
void shmem_barrier_all(void);
```

**FORTRAN:**

```
CALL SHMEM_BARRIER_ALL
```

#### DESCRIPTION

##### Arguments

None.

##### API description

The *shmem\_barrier\_all* routine registers the arrival of a PE at a barrier. Barriers are a fast mechanism for synchronizing all PEs at once. This routine causes a PE to suspend execution until all PEs have called *shmem\_barrier\_all*. This routine must be used with PEs started by *shmem\_init*.

Prior to synchronizing with other PEs, *shmem\_barrier\_all* ensures completion of all previously issued memory stores and remote memory updates issued via OpenSHMEM AMOs and RMA routine calls such as *shmem\_int\_add* and *shmem\_put32*.

##### Return Values

None.

##### Notes

None.

#### EXAMPLES

The following *shmem\_barrier\_all* example is for C/C++ programs:

```

1  #include <stdio.h>
2  #include <shmem.h>
3
4  int x=1010;
5
6  int main(void)
7  {
8      int me, npes;
9
10     shmem_init();
11     me = shmem_my_pe();
12     npes = shmem_n_pes();
13
14     /*put to next PE in a circular fashion*/
15     shmem_int_p(&x, 4, (me+1)%npes);
16     /*synchronize all PEs*/
17     shmem_barrier_all();
18
19     printf("%d: x = %d\n", me, x);
20     return 0;
21 }

```

## 8.5.2 SHMEM\_BARRIER

Performs all operations described in the *shmem\_barrier\_all* interface but with respect to a subset of PEs defined by the *Active set*.

### SYNOPSIS

#### C/C++:

```
void shmem_barrier(int PE_start, int logPE_stride, int PE_size, long *pSync);
```

#### FORTRAN:

```
INTEGER PE_start, logPE_stride, PE_size
```

```
INTEGER pSync (SHMEM_BARRIER_SYNC_SIZE)
```

```
CALL SHMEM_BARRIER(PE_start, logPE_stride, PE_size, pSync)
```

### DESCRIPTION

#### Arguments

<b>IN</b>	<i>PE_start</i>	The lowest PE number of the <i>Active set</i> of PEs. <i>PE_start</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>logPE_stride</i>	The log (base 2) of the stride between consecutive PE numbers in the <i>Active set</i> . <i>logPE_stride</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>PE_size</i>	The number of PEs in the <i>Active set</i> . <i>PE_size</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>pSync</i>	A symmetric work array. In <i>C/C++</i> , <i>pSync</i> must be of type long and size <i>_SHMEM_BARRIER_SYNC_SIZE</i> . In <i>Fortran</i> , <i>pSync</i> must be of type integer and size <i>SHMEM_BARRIER_SYNC_SIZE</i> . If you are using <i>Fortran</i> , it must be a default integer type. Every element of this array must be initialized to <i>SHMEM_SYNC_VALUE</i> before any of the PEs in the <i>Active set</i> enter <i>shmem_barrier</i> the first time.

#### API description

*shmem\_barrier* is a collective synchronization routine over an *Active set*. Control returns from *shmem\_barrier* after all PEs in the *Active set* (specified by *PE\_start*, *logPE\_stride*, and *PE\_size*) have called *shmem\_barrier*.

As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the *Active set* call the routine. If a PE not in the *Active set* calls an OpenSHMEM collective routine, undefined behavior results.

The values of arguments *PE\_start*, *logPE\_stride*, and *PE\_size* must be equal on all PEs in the *Active set*. The same work array must be passed in *pSync* to all PEs in the *Active set*.

*shmem\_barrier* ensures that all previously issued stores and remote memory updates, including AMOs and RMA operations, done by any of the PEs in the *Active set* are complete before returning.

The same *pSync* array may be reused on consecutive calls to *shmem\_barrier* if the same active PE set is used.

### Return Values

None.

### Notes

If the *pSync* array is initialized at run time, be sure to use some type of synchronization, for example, a call to *shmem\_barrier\_all*, before calling *shmem\_barrier* for the first time.

If the *Active set* does not change, *shmem\_barrier* can be called repeatedly with the same *pSync* array. No additional synchronization beyond that implied by *shmem\_barrier* itself is necessary in this case.

## EXAMPLES

The following barrier example is for C/C++ programs:

```
#include <stdio.h>
#include <shmem.h>

long pSync[_SHMEM_BARRIER_SYNC_SIZE];
int x = 10101;

int main(void)
{
    int i, me, npes;

    for (i = 0; i < _SHMEM_BARRIER_SYNC_SIZE; i += 1){
        pSync[i] = _SHMEM_SYNC_VALUE;
    }

    shmem_init();
    me = shmem_my_pe();
    npes = shmem_n_pes();

    if(me % 2 == 0){
        x = 1000 + me;
        /*put to next even PE in a circular fashion*/
        shmem_int_p(&x, 4, (me+2)%npes);
        /*synchronize all even pes*/
        shmem_barrier(0, 1, (npes/2 + npes%2), pSync);
    }
    printf("%d: x = %d\n", me, x);
    return 0;
}
```

### 8.5.3 SHMEM\_BROADCAST

Broadcasts a block of data from one PE to one or more destination PEs.

#### SYNOPSIS

C/C++:

```

1 void shmem_broadcast32(void *dest, const void *source, size_t nelems, int PE_root, int
2   PE_start, int logPE_stride, int PE_size, long *pSync);
3 void shmem_broadcast64(void *dest, const void *source, size_t nelems, int PE_root, int
4   PE_start, int logPE_stride, int PE_size, long *pSync);

```

**FORTRAN:**

```

5
6 INTEGER nelems, PE_root, PE_start, logPE_stride, PE_size
7 INTEGER pSync(SHMEM_BCAST_SYNC_SIZE)
8 CALL SHMEM_BROADCAST4(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
9 CALL SHMEM_BROADCAST8(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
10 CALL SHMEM_BROADCAST32(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
11 CALL SHMEM_BROADCAST64(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)

```

**DESCRIPTION****Arguments**

<b>OUT</b>	<i>dest</i>	A symmetric data object.
<b>IN</b>	<i>source</i>	A symmetric data object that can be of any data type that is permissible for the <i>dest</i> argument.
<b>IN</b>	<i>nelems</i>	The number of elements in <i>source</i> . For <i>shmem_broadcast32</i> and <i>shmem_broadcast4</i> , this is the number of 32-bit halfwords. <i>nelems</i> must be of type <i>size_t</i> in <i>C</i> . If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>PE_root</i>	Zero-based ordinal of the PE, with respect to the <i>Active set</i> , from which the data is copied. Must be greater than or equal to 0 and less than <i>PE_size</i> . <i>PE_root</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>PE_start</i>	The lowest PE number of the <i>Active set</i> of PEs. <i>PE_start</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>logPE_stride</i>	The log (base 2) of the stride between consecutive PE numbers in the <i>Active set</i> . <i>logPE_stride</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>PE_size</i>	The number of PEs in the <i>Active set</i> . <i>PE_size</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>pSync</i>	A symmetric work array. In <i>C/C++</i> , <i>pSync</i> must be of type long and size <i>_SHMEM_BCAST_SYNC_SIZE</i> . In <i>Fortran</i> , <i>pSync</i> must be of type integer and size <i>SHMEM_BCAST_SYNC_SIZE</i> . Every element of this array must be initialized with the value <i>_SHMEM_SYNC_VALUE</i> (in <i>C/C++</i> ) or <i>SHMEM_SYNC_VALUE</i> (in <i>Fortran</i> ) before any of the PEs in the <i>Active set</i> enter <i>shmem_broadcast</i> .

**API description**

OpenSHMEM broadcast routines are collective routines. They copy data object *source* on the processor specified by *PE\_root* and store the values at *dest* on the other PEs specified by the triplet *PE\_start*, *logPE\_stride*, *PE\_size*. The data is not copied to the *dest* area on the root PE.

As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the *Active set* call the routine. If a PE not in the *Active set* calls an OpenSHMEM collective routine, undefined behavior results.

The values of arguments *PE\_root*, *PE\_start*, *logPE\_stride*, and *PE\_size* must be equal on all PEs in the *Active set*. The same *dest* and *source* data objects and the same *pSync* work array must be passed to all PEs in the *Active set*.

Before any PE calls a broadcast routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this): The *pSync* array on all PEs in



the *Active set* is not still in use from a prior call to a broadcast routine. The *dest* array on all PEs in the *Active set* is ready to accept the broadcast data.

Upon return from a broadcast routine, the following are true for the local PE: If the current PE is not the root PE, the *dest* data object is updated. The values in the *pSync* array are restored to the original values.

The *dest* and *source* data objects must conform to certain typing constraints, which are as follows:

Routine	Data Type of <i>dest</i> and <i>source</i>
<code>shmem_broadcast8</code> , <code>shmem_broadcast64</code> <code>shmem_broadcast32</code>	Any noncharacter type that has an element size of 64 bits. No <i>Fortran</i> derived types or <i>C/C++</i> structures are allowed.
<code>shmem_broadcast32</code>	Any noncharacter type that has an element size of 32 bits. No <i>Fortran</i> derived types or <i>C/C++</i> structures are allowed.
<code>shmem_broadcast4</code>	Any noncharacter type that has an element size of 32 bits.

### Return Values

None.

### Notes

All OpenSHMEM broadcast routines restore *pSync* to its original contents. Multiple calls to OpenSHMEM routines that use the same *pSync* array do not require that *pSync* be reinitialized after the first call.

You must ensure that the *pSync* array is not being updated by any PE in the *Active set* while any of the PEs participates in processing of an OpenSHMEM broadcast routine. Be careful to avoid these situations: If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the *Active set* have initialized *pSync* before any of them enter an OpenSHMEM routine called with the *pSync* synchronization array. A *pSync* array may be reused on a subsequent OpenSHMEM broadcast routine only if none of the PEs in the *Active set* are still processing a prior OpenSHMEM broadcast routine call that used the same *pSync* array. In general, this can be ensured only by doing some type of synchronization.

### EXAMPLES

In the following examples, the call to `shmem_broadcast64` copies *source* on PE 4 to *dest* on PEs 5, 6, and 7.

*C/C++* example:

```
#include <stdio.h>
#include <stdlib.h>
#include <shmem.h>

#define NUM_ELEMS 4
long pSync[_SHMEM_BCAST_SYNC_SIZE];
long source[NUM_ELEMS], dest[NUM_ELEMS];

int main(void)
{
    int i, me, npes;

    shmem_init();
    me = shmem_my_pe();
    npes = shmem_n_pes();

    if (me == 0)
        for (i = 0; i < NUM_ELEMS; i++)
            source[i] = i;
    for (i=0; i < _SHMEM_BCAST_SYNC_SIZE; i++) {
        pSync[i] = _SHMEM_SYNC_VALUE;
```

```

1      }
2      shmem_barrier_all(); /* Wait for all PEs to initialize pSync */
3
4      shmem_broadcast64(dest, source, NUM_ELEMS, 4, 5, 0, 3, pSync);
5      printf("%d: %ld", me, dest[0]);
6      for (i = 1; i < NUM_ELEMS; i++)
7          printf(", %ld", dest[i]);
8      printf("\n");
9      return 0;
10     }

```

*Fortran example:*

```

11     INTEGER PSYNC(SHMEM_BCAST_SYNC_SIZE)
12     INTEGER DEST, SOURCE, NLONG, PE_ROOT, PE_START,
13     & LOGPE_STRIDE, PE_SIZE, PSYNC
14     COMMON /COM/ DEST, SOURCE
15
16     DATA PSYNC /SHMEM_BCAST_SYNC_SIZE*SHMEM_SYNC_VALUE/
17
18     CALL SHMEM_BROADCAST64(DEST, SOURCE, NLONG, 0, 4, 0, 4, PSYNC)

```

#### 8.5.4 SHMEM\_COLLECT, SHMEM\_FCOLLECT

Concatenates blocks of data from multiple PEs to an array in every PE.

#### SYNOPSIS

##### **C/C++:**

```

24 void shmem_collect32(void *dest, const void *source, size_t nelems, int PE_start, int
25     logPE_stride, int PE_size, long *pSync);
26 void shmem_collect64(void *dest, const void *source, size_t nelems, int PE_start, int
27     logPE_stride, int PE_size, long *pSync);
28 void shmem_fcollect32(void *dest, const void *source, size_t nelems, int PE_start, int
29     logPE_stride, int PE_size, long *pSync);
30 void shmem_fcollect64(void *dest, const void *source, size_t nelems, int PE_start, int
31     logPE_stride, int PE_size, long *pSync);

```

##### **FORTRAN:**

```

32 INTEGER nelems
33 INTEGER PE_start, logPE_stride, PE_size
34 INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)
35 CALL SHMEM_COLLECT4(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
36 CALL SHMEM_COLLECT8(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
37 CALL SHMEM_COLLECT32(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
38 CALL SHMEM_COLLECT64(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
39 CALL SHMEM_FCOLLECT4(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
40 CALL SHMEM_FCOLLECT8(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
41 CALL SHMEM_FCOLLECT32(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
42 CALL SHMEM_FCOLLECT64(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)

```

#### DESCRIPTION

##### **Arguments**

48

<b>OUT</b>	<i>dest</i>	A symmetric array. The <i>dest</i> argument must be large enough to accept the concatenation of the <i>source</i> arrays on all PEs. The data types are as follows: For <i>shmem_collect8</i> , <i>shmem_collect64</i> , <i>shmem_fcollect8</i> , and <i>shmem_fcollect64</i> , any data type with an element size of 64 bits. <i>Fortran</i> derived types, <i>Fortran</i> character type, and <i>C/C++</i> structures are not permitted. For <i>shmem_collect4</i> , <i>shmem_collect32</i> , <i>shmem_fcollect4</i> , and <i>shmem_fcollect32</i> , any data type with an element size of 32 bits. <i>Fortran</i> derived types, <i>Fortran</i> character type, and <i>C/C++</i> structures are not permitted.	1 2 3 4 5 6 7 8
<b>IN</b>	<i>source</i>	A symmetric data object that can be of any type permissible for the <i>dest</i> argument.	9 10
<b>IN</b>	<i>nelems</i>	The number of elements in the <i>source</i> array. <i>nelems</i> must be of type <i>size_t</i> for <i>C</i> . If you are using <i>Fortran</i> , it must be a default integer value.	11 12
<b>IN</b>	<i>PE_start</i>	The lowest PE number of the <i>Active set</i> of PEs. <i>PE_start</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.	13 14
<b>IN</b>	<i>logPE_stride</i>	The log (base 2) of the stride between consecutive PE numbers in the <i>Active set</i> . <i>logPE_stride</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.	15 16
<b>IN</b>	<i>PE_size</i>	The number of PEs in the <i>Active set</i> . <i>PE_size</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.	17 18
<b>IN</b>	<i>pSync</i>	A symmetric work array. In <i>C/C++</i> , <i>pSync</i> must be of type long and size <i>_SHMEM_COLLECT_SYNC_SIZE</i> . In <i>Fortran</i> , <i>pSync</i> must be of type integer and size <i>SHMEM_COLLECT_SYNC_SIZE</i> . If you are using <i>Fortran</i> , it must be a default integer value. Every element of this array must be initialized with the value <i>_SHMEM_SYNC_VALUE</i> in <i>C/C++</i> or <i>SHMEM_SYNC_VALUE</i> in <i>Fortran</i> before any of the PEs in the <i>Active set</i> enter <i>shmem_collect</i> or <i>shmem_fcollect</i> .	19 20 21 22 23 24 25

### API description

*OpenSHMEM collect* and *fcollect* routines concatenate *nelems* 64-bit or 32-bit data items from the *source* array into the *dest* array, over the set of PEs defined by *PE\_start*, *log2PE\_stride*, and *PE\_size*, in processor number order. The resultant *dest* array contains the contribution from PE *PE\_start* first, then the contribution from PE *PE\_start + PE\_stride* second, and so on. The collected result is written to the *dest* array for all PEs in the *Active set*.

The *fcollect* routines require that *nelems* be the same value in all participating PEs, while the *collect* routines allow *nelems* to vary from PE to PE.

As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the *Active set* call the routine. If a PE not in the *Active set* and calls this collective routine, the behavior is undefined.

The values of arguments *PE\_start*, *logPE\_stride*, and *PE\_size* must be equal on all PEs in the *Active set*. The same *dest* and *source* arrays and the same *pSync* work array must be passed to all PEs in the *Active set*.

Upon return from a collective routine, the following are true for the local PE: The *dest* array is updated. The values in the *pSync* array are restored to the original values.

### Return Values

None.

### Notes

All OpenSHMEM collective routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the *Active set* while any of the PEs participate in processing of an OpenSHMEM collective routine. Be careful to avoid these situations: If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter an OpenSHMEM routine called with the *pSync*

synchronization array. A *pSync* array can be reused on a subsequent OpenSHMEM collective routine only if none of the PEs in the *Active set* are still processing a prior OpenSHMEM collective routine call that used the same *pSync* array. In general, this may be ensured only by doing some type of synchronization.

The collective routines operate on active PE sets that have a non-power-of-two *PE\_size* with some performance degradation. They operate with no performance degradation when *nelems* is a non-power-of-two value.

## EXAMPLES

The following *shmem\_collect* example is for C/C++ programs:

```

11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <shmem.h>
14
15 long pSync[_SHMEM_COLLECT_SYNC_SIZE];
16 int source[2];
17
18 int main(void)
19 {
20     int i, me, npes;
21     int *dest;
22
23     shmem_init();
24     me = shmem_my_pe();
25     npes = shmem_n_pes();
26
27     source[0] = me * 2;
28     source[1] = me * 2 + 1;
29     dest = (int *)shmem_malloc(sizeof(int) * npes * 2);
30     for (i=0; i < _SHMEM_COLLECT_SYNC_SIZE; i++) {
31         pSync[i] = _SHMEM_SYNC_VALUE;
32     }
33     shmem_barrier_all(); /* Wait for all PEs to initialize pSync */
34
35     shmem_collect32(dest, source, 2, 0, 0, npes, pSync);
36     printf("%d: %d", me, dest[0]);
37     for (i = 1; i < npes * 2; i++)
38         printf(" %d", dest[i]);
39     printf("\n");
40     return 0;
41 }

```

The following *SHMEM\_COLLECT* example is for Fortran programs:

```

34 INTEGER PSYNC(SHMEM_COLLECT_SYNC_SIZE)
35 DATA PSYNC /SHMEM_COLLECT_SYNC_SIZE*SHMEM_SYNC_VALUE/
36
37 CALL SHMEM_COLLECT4(DEST, SOURCE, 64, PE_START, LOGPE_STRIDE,
38 & PE_SIZE, PSYNC)

```

### 8.5.5 SHMEM\_REDUCTIONS

Performs arithmetic and logical operations across a set of PEs.

#### SYNOPSIS

##### AND

Performs a bitwise AND function across a set of processing elements (PEs).

**C/C++:**

```

void shmem_int_and_to_all(int *dest, int *source, int nreduce, int PE_start, int logPE_stride
, int PE_size, int *pWrk, long *pSync);
void shmem_long_and_to_all(long *dest, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlong_and_to_all(long long *dest, long long *source, int nreduce, int PE_start,
int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_and_to_all(short *dest, short *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_int_and_to_all(int *dest, int *source, int nreduce, int PE_start, int logPE_stride
, int PE_size, int *pWrk, long *pSync);

```

**FORTRAN:**

```

CALL SHMEM_INT4_AND_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
pSync)
CALL SHMEM_INT8_AND_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
pSync)

```

**MAX**

Performs a maximum function reduction across a set of processing elements (PEs).

**C/C++:**

```

void shmem_double_max_to_all(double *dest, double *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_max_to_all(float *dest, float *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_max_to_all(int *dest, int *source, int nreduce, int PE_start, int logPE_stride
, int PE_size, int *pWrk, long *pSync);
void shmem_long_max_to_all(long *dest, long *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_max_to_all(long double *dest, long double *source, int nreduce, int
PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_max_to_all(long long *dest, long long *source, int nreduce, int PE_start,
int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_max_to_all(short *dest, short *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, short *pWrk, long *pSync);

```

**FORTRAN:**

```

CALL SHMEM_INT4_MAX_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
pSync)
CALL SHMEM_INT8_MAX_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
pSync)
CALL SHMEM_REAL4_MAX_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
pSync)
CALL SHMEM_REAL8_MAX_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
pSync)
CALL SHMEM_REAL16_MAX_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
pSync)

```

**MIN**

Performs a minimum function reduction across a set of processing elements (PEs).

**C/C++:**

```

void shmem_double_min_to_all(double *dest, double *source, int nreduce, int PE_start, int
logPE_stride, int PE_size, double *pWrk, long *pSync);

```

```

1 void shmem_float_min_to_all(float *dest, float *source, int nreduce, int PE_start, int
2   logPE_stride, int PE_size, float *pWrk, long *pSync);
3 void shmem_int_min_to_all(int *dest, int *source, int nreduce, int PE_start, int logPE_stride
4   , int PE_size, int *pWrk, long *pSync);
5 void shmem_long_min_to_all(long *dest, long *source, int nreduce, int PE_start, int
6   logPE_stride, int PE_size, long *pWrk, long *pSync);
7 void shmem_longdouble_min_to_all(long double *dest, long double *source, int nreduce, int
8   PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
9 void shmem_longlong_min_to_all(long long *dest, long long *source, int nreduce, int PE_start,
10  int logPE_stride, int PE_size, long long *pWrk, long *pSync);
11 void shmem_short_min_to_all(short *dest, short *source, int nreduce, int PE_start, int
12  logPE_stride, int PE_size, short *pWrk, long *pSync);

```

**FORTRAN:**

```

13 CALL SHMEM_INT4_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
14   pSync)
15 CALL SHMEM_INT8_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
16   pSync)
17 CALL SHMEM_REAL4_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
18   pSync)
19 CALL SHMEM_REAL8_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
20   pSync)
21 CALL SHMEM_REAL16_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
22   pSync)

```

**SUM**

Performs a sum reduction across a set of processing elements (PEs).

**C/C++:**

```

27 void shmem_complexd_sum_to_all(double complex *dest, double complex *source, int nreduce, int
28   PE_start, int logPE_stride, int PE_size, double complex *pWrk, long *pSync);
29 void shmem_complexf_sum_to_all(float complex *dest, float complex *source, int nreduce, int
30   PE_start, int logPE_stride, int PE_size, float complex *pWrk, long *pSync);
31 void shmem_double_sum_to_all(double *dest, double *source, int nreduce, int PE_start, int
32   logPE_stride, int PE_size, double *pWrk, long *pSync);
33 void shmem_float_sum_to_all(float *dest, float *source, int nreduce, int PE_start, int
34   logPE_stride, int PE_size, float *pWrk, long *pSync);
35 void shmem_int_sum_to_all(int *dest, int *source, int nreduce, int PE_start, int logPE_stride
36   , int PE_size, int *pWrk, long *pSync);
37 void shmem_long_sum_to_all(long *dest, long *source, int nreduce, int PE_start, int
38   logPE_stride, int PE_size, long *pWrk, long *pSync);
39 void shmem_longdouble_sum_to_all(long double *dest, long double *source, int nreduce, int
40   PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
41 void shmem_longlong_sum_to_all(long long *dest, long long *source, int nreduce, int PE_start,
42   int logPE_stride, int PE_size, long long *pWrk, long *pSync);

```

**FORTRAN:**

```

44 CALL SHMEM_COMP4_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
45   pSync)
46 CALL SHMEM_COMP8_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
47   pSync)
48 CALL SHMEM_INT4_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,

```

```

    pSync)
CALL SHMEM_INT8_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL4_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL8_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL16_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)

```

**PROD**

Performs a product reduction across a set of processing elements (PEs).

**C/C++:**

```

void shmem_complexd_prod_to_all(double complex *dest, double complex *source, int nreduce,
    int PE_start, int logPE_stride, int PE_size, double complex *pWrk, long *pSync);
void shmem_complexf_prod_to_all(float complex *dest, float complex *source, int nreduce, int
    PE_start, int logPE_stride, int PE_size, float complex *pWrk, long *pSync);
void shmem_double_prod_to_all(double *dest, double *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_prod_to_all(float *dest, float *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_prod_to_all(int *dest, int *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_prod_to_all(long *dest, long *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_prod_to_all(long double *dest, long double *source, int nreduce, int
    PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_prod_to_all(long long *dest, long long *source, int nreduce, int PE_start
    , int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_prod_to_all(short *dest, short *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, short *pWrk, long *pSync);

```

**FORTRAN:**

```

CALL SHMEM_COMP4_PROD_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_COMP8_PROD_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_INT4_PROD_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_INT8_PROD_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL4_PROD_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL8_PROD_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL16_PROD_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)

```

**OR**

Performs a bitwise OR function reduction across a set of processing elements (PEs).

**C/C++:**

```

1 void shmem_int_or_to_all(int *dest, int *source, int nreduce, int PE_start, int logPE_stride,
2   int PE_size, int *pWrk, long *pSync);
3 void shmem_long_or_to_all(long *dest, long *source, int nreduce, int PE_start, int
4   logPE_stride, int PE_size, long *pWrk, long *pSync);
5 void shmem_longlong_or_to_all(long long *dest, long long *source, int nreduce, int PE_start,
6   int logPE_stride, int PE_size, long long *pWrk, long *pSync);
7 void shmem_short_or_to_all(short *dest, short *source, int nreduce, int PE_start, int
8   logPE_stride, int PE_size, short *pWrk, long *pSync);

```

**FORTRAN:**

```

9 CALL SHMEM_INT4_OR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync
10 )
11 CALL SHMEM_INT8_OR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync
12 )
13
14

```

**XOR**

Performs a bitwise EXCLUSIVE OR reduction across a set of processing elements (PEs).

**C/C++:**

```

15 void shmem_int_xor_to_all(int *dest, int *source, int nreduce, int PE_start, int logPE_stride
16   , int PE_size, int *pWrk, long *pSync);
17 void shmem_long_xor_to_all(long *dest, long *source, int nreduce, int PE_start, int
18   logPE_stride, int PE_size, long *pWrk, long *pSync);
19 void shmem_longlong_xor_to_all(long long *dest, long long *source, int nreduce, int PE_start,
20   int logPE_stride, int PE_size, long long *pWrk, long *pSync);
21 void shmem_short_xor_to_all(short *dest, short *source, int nreduce, int PE_start, int
22   logPE_stride, int PE_size, short *pWrk, long *pSync);
23
24

```

**FORTRAN:**

```

25 CALL SHMEM_INT4_XOR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
26   pSync)
27 CALL SHMEM_INT8_XOR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
28   pSync)
29
30

```

**DESCRIPTION****Arguments**

<b>IN</b>	<i>dest</i>	A symmetric array, of length <i>nreduce</i> elements, to receive the result of the reduction routines. The data type of <i>dest</i> varies with the version of the reduction routine being called. When calling from C/C++, refer to the SYNOPSIS section for data type information.
<b>IN</b>	<i>source</i>	A symmetric array, of length <i>nreduce</i> elements, that contains one element for each separate reduction routine. The <i>source</i> argument must have the same data type as <i>dest</i> .
<b>IN</b>	<i>nreduce</i>	The number of elements in the <i>dest</i> and <i>source</i> arrays. <i>nreduce</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>PE_start</i>	The lowest PE number of the <i>Active set</i> of PEs. <i>PE_start</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>logPE_stride</i>	The log (base 2) of the stride between consecutive PE numbers in the <i>Active set</i> . <i>logPE_stride</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.
<b>IN</b>	<i>PE_size</i>	The number of PEs in the <i>Active set</i> . <i>PE_size</i> must be of type integer. If you are using <i>Fortran</i> , it must be a default integer value.



<b>IN</b>	<i>pWrk</i>	A symmetric work array. The <i>pWrk</i> argument must have the same data type as <i>dest</i> . In <i>C/C++</i> , this contains $\max(nreduce/2 + 1, \_SHMEM\_REDUCE\_MIN\_WRKDATA\_SIZE)$ elements. In <i>Fortran</i> , this contains $\max(nreduce/2 + 1, SHMEM\_REDUCE\_MIN\_WRKDATA\_SIZE)$ elements.
<b>IN</b>	<i>pSync</i>	A symmetric work array. In <i>C/C++</i> , <i>pSync</i> must be of type long and size <i>_SHMEM_REDUCE_SYNC_SIZE</i> . In <i>Fortran</i> , <i>pSync</i> must be of type integer and size <i>SHMEM_REDUCE_SYNC_SIZE</i> . If you are using <i>Fortran</i> , it must be a default integer value. Every element of this array must be initialized with the value <i>_SHMEM_SYNC_VALUE</i> (in <i>C/C++</i> ) or <i>SHMEM_SYNC_VALUE</i> (in <i>Fortran</i> ) before any of the PEs in the <i>Active set</i> enter the reduction routine.

### API description

OpenSHMEM reduction routines compute one or more reductions across symmetric arrays on multiple PEs. A reduction performs an associative binary routine across a set of values.

The *nreduce* argument determines the number of separate reductions to perform. The *source* array on all PEs in the *Active set* provides one element for each reduction. The results of the reductions are placed in the *dest* array on all PEs in the *Active set*. The *Active set* is defined by the *PE\_start*, *logPE\_stride*, *PE\_size* triplet.

The *source* and *dest* arrays may be the same array, but they may not be overlapping arrays.

As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the *Active set* call the routine. If a PE not in the *Active set* calls an OpenSHMEM collective routine, undefined behavior results.

The values of arguments *nreduce*, *PE\_start*, *logPE\_stride*, and *PE\_size* must be equal on all PEs in the *Active set*. The same *dest* and *source* arrays, and the same *pWrk* and *pSync* work arrays, must be passed to all PEs in the *Active set*.

Before any PE calls a reduction routine, you must ensure that the following conditions exist (synchronization via a *barrier* or some other method is often needed to ensure this): The *pWrk* and *pSync* arrays on all PEs in the *Active set* are not still in use from a prior call to a collective OpenSHMEM routine. The *dest* array on all PEs in the *Active set* is ready to accept the results of the *reduction*.

Upon return from a reduction routine, the following are true for the local PE: The *dest* array is updated. The values in the *pSync* array are restored to the original values.

When calling from *Fortran*, the *dest* data types are as follows:

Routine	Data Type
<i>shmem_int8_and_to_all</i>	Integer, with an element size of 8 bytes.
<i>shmem_int4_and_to_all</i>	Integer, with an element size of 4 bytes.
<i>shmem_comp8_max_to_all</i>	Complex, with an element size equal to two 8-byte real values.
<i>shmem_int4_max_to_all</i>	Integer, with an element size of 4 bytes.
<i>shmem_int8_max_to_all</i>	Integer, with an element size of 8 bytes.
<i>shmem_real4_max_to_all</i>	Real, with an element size of 4 bytes.
<i>shmem_real16_max_to_all</i>	Real, with an element size of 16 bytes.
<i>shmem_int4_min_to_all</i>	Integer, with an element size of 4 bytes.
<i>shmem_int8_min_to_all</i>	Integer, with an element size of 8 bytes.
<i>shmem_real4_min_to_all</i>	Real, with an element size of 4 bytes.
<i>shmem_real8_min_to_all</i>	Real, with an element size of 8 bytes.
<i>shmem_real16_min_to_all</i>	Real, with an element size of 16 bytes.
<i>shmem_comp4_sum_to_all</i>	Complex, with an element size equal to two 4-byte real values.
<i>shmem_comp8_sum_to_all</i>	Complex, with an element size equal to two 8-byte real values.
<i>shmem_int4_sum_to_all</i>	Integer, with an element size of 4 bytes.
<i>shmem_int8_sum_to_all</i>	Integer, with an element size of 8 bytes.

1	shmem_real4_sum_to_all	Real, with an element size of 4 bytes.
2	shmem_real8_sum_to_all	Real, with an element size of 8 bytes.
3	shmem_real16_sum_to_all	Real, with an element size of 16 bytes.
4	shmem_comp4_prod_to_all	Complex, with an element size equal to two 4-byte real values.
5	shmem_comp8_prod_to_all	Complex, with an element size equal to two 8-byte real values.
6	shmem_int4_prod_to_all	Integer, with an element size of 4 bytes.
7	shmem_int8_prod_to_all	Integer, with an element size of 8 bytes.
8	shmem_real4_prod_to_all	Real, with an element size of 4 bytes.
9	shmem_real8_prod_to_all	Real, with an element size of 8 bytes.
10	shmem_real16_prod_to_all	Real, with an element size of 16 bytes.
11	shmem_int8_or_to_all	Integer, with an element size of 8 bytes.
12	shmem_int4_or_to_all	Integer, with an element size of 4 bytes.
13	shmem_int8_xor_to_all	Integer, with an element size of 8 bytes.
14	shmem_int4_xor_to_all	Integer, with an element size of 4 bytes.

### Return Values

None.

### Notes

All OpenSHMEM reduction routines reset the values in *pSync* before they return, so a particular *pSync* buffer need only be initialized the first time it is used.

You must ensure that the *pSync* array is not being updated on any PE in the *Active set* while any of the PEs participate in processing of an OpenSHMEM reduction routine. Be careful to avoid the following situations: If the *pSync* array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized *pSync* before any of them enter an OpenSHMEM routine called with the *pSync* synchronization array. A *pSync* or *pWrk* array can be reused in a subsequent reduction routine call only if none of the PEs in the *Active set* are still processing a prior reduction routine call that used the same *pSync* or *pWrk* arrays. In general, this can be assured only by doing some type of synchronization.

### EXAMPLES

This *Fortran* reduction example statically initializes the *pSync* array and finds the logical *AND* of the integer variable *FOO* across all even PEs.

```

31 INCLUDE "shmem.fh"
32
33 INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
34 DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
35 PARAMETER (NR=1)
36 INTEGER*4 PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
37 INTEGER FOO, FOOAND
38 SAVE FOO, FOOAND, PWRK
39 INTRINSIC SHMEM_MY_PE()
40
41 FOO = SHMEM_MY_PE()
42 IF (MOD(SHMEM_MY_PE(),2) .EQ. 0) THEN
43   IF (MOD(SHMEM_N_PES(),2) .EQ. 0) THEN
44     CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2, &
45     PWRK, PSYNC)
46   ELSE
47     CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2+1, &
48     PWRK, PSYNC)
49   ENDIF
50 PRINT*, 'Result on PE ', SHMEM_MY_PE(), ' is ', FOOAND
51 ENDIF

```

This *Fortran* example statically initializes the *pSync* array and finds the *maximum* value of real variable *FOO* across all even PEs.

```

INCLUDE "shmem.fh"
INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOMAX, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOMAX, PWRK
INTRINSIC SHMEM_MY_PE()

IF ( MOD(SHMEM_MY_PE() .EQ. 0) THEN
    CALL SHMEM_REAL8_MAX_TO_ALL(FOOMAX, FOO, NR, 0, 1, N$PES/2,
    & PWRK, PSYNC)
    PRINT*, 'Result on PE ', SHMEM_MY_PE(), ' is ', FOOMAX
ENDIF

```

This *Fortran* example statically initializes the *pSync* array and finds the *minimum* value of real variable *FOO* across all the even PEs.

```

INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOMIN, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOMIN, PWRK
INTRINSIC SHMEM_MY_PE()

IF ( MOD(SHMEM_MY_PE() .EQ. 0) THEN
    CALL SHMEM_REAL8_MIN_TO_ALL(FOOMIN, FOO, NR, 0, 1, N$PES/2,
    & PWRK, PSYNC)
    PRINT*, 'Result on PE ', SHMEM_MY_PE(), ' is ', FOOMIN
ENDIF

```

This *Fortran* example statically initializes the *pSync* array and finds the *sum* of the real variable *FOO* across all even PEs.

```

INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOSUM, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOSUM, PWRK
INTRINSIC SHMEM_MY_PE()

IF ( MOD(SHMEM_MY_PE() .EQ. 0) THEN
    CALL SHMEM_INT4_SUM_TO_ALL(FOOSUM, FOO, NR, 0, 1, N$PES/2,
    & PWRK, PSYNC)
    PRINT*, 'Result on PE ', SHMEM_MY_PE(), ' is ', FOOSUM
ENDIF

```

This *Fortran* example statically initializes the *pSync* array and finds the *product* of the real variable *FOO* across all the even PEs.

```

INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
REAL FOO, FOOPROD, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOPROD, PWRK
INTRINSIC SHMEM_MY_PE()

IF ( MOD(SHMEM_MY_PE() .EQ. 0) THEN
    CALL SHMEM_COMP8_PROD_TO_ALL(FOOPROD, FOO, NR, 0, 1, N$PES/2,
    & PWRK, PSYNC)
    PRINT*, 'Result on PE ', SHMEM_MY_PE(), ' is ', FOOPROD
ENDIF

```

This *Fortran* example statically initializes the *pSync* array and finds the logical *OR* of the integer variable *FOO* across all even PEs.

```

1  INCLUDE "mpp/shmem.fh"
2
3  INTEGER PSYNC (SHMEM_REDUCE_SYNC_SIZE)
4  DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
5  PARAMETER (NR=1)
6  REAL PWRK (MAX (NR/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
7  INTEGER FOO, FOOOR
8  COMMON /COM/ FOO, FOOOR, PWRK
9  INTRINSIC SHMEM_MY_PE ()
10
11 IF ( MOD (SHMEM_MY_PE () .EQ. 0) THEN
12     CALL SHMEM_INT8_OR_TO_ALL (FOOOR, FOO, NR, 0, 1, N$PES/2,
13     & PWRK, PSYNC)
14     PRINT*, 'Result on PE ', SHMEM_MY_PE (), ' is ', FOOOR
15 ENDIF

```

This *Fortran* example statically initializes the *pSync* array and computes the exclusive *XOR* of variable *FOO* across all even PEs.

```

16 INCLUDE "mpp/shmem.fh"
17
18 INTEGER PSYNC (SHMEM_REDUCE_SYNC_SIZE)
19 DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
20 PARAMETER (NR=1)
21 REAL FOO, FOOOR, PWRK (MAX (NR/2+1, SHMEM_REDUCE_MIN_WRKDATA_SIZE))
22 COMMON /COM/ FOO, FOOOR, PWRK
23 INTRINSIC SHMEM_MY_PE ()
24
25 IF ( MOD (SHMEM_MY_PE () .EQ. 0) THEN
26     CALL SHMEM_REAL8_XOR_TO_ALL (FOOOR, FOO, NR, 0, 1, N$PES/2,
27     & PWRK, PSYNC)
28     PRINT*, 'Result on PE ', SHMEM_MY_PE (), ' is ', FOOOR
29 ENDIF

```

## 8.6 Point-To-Point Synchronization Routines

The following section discusses OpenSHMEM APIs that provides a mechanism for synchronization between two PEs based on the value of a symmetric data object.

### 8.6.1 SHMEM\_WAIT

Wait for a variable on the local PE to change.

#### SYNOPSIS

##### C/C++:

```

38 void shmem_int_wait (int *ivar, int cmp_value);
39 void shmem_int_wait_until (int *ivar, int cmp, int cmp_value);
40 void shmem_long_wait (long *ivar, long cmp_value);
41 void shmem_long_wait_until (long *ivar, int cmp, long cmp_value);
42 void shmem_longlong_wait (long long *ivar, long long cmp_value);
43 void shmem_longlong_wait_until (long long *ivar, int cmp, long long cmp_value);
44 void shmem_short_wait (short *ivar, short cmp_value);
45 void shmem_short_wait_until (short *ivar, int cmp, short cmp_value);
46 void shmem_wait (long *ivar, long cmp_value);
47 void shmem_wait_until (long *ivar, int cmp, long cmp_value);

```

##### FORTTRAN:

```

CALL SHMEM_INT4_WAIT(ivar, cmp_value)
CALL SHMEM_INT4_WAIT_UNTIL(ivar, cmp, cmp_value)
CALL SHMEM_INT8_WAIT(ivar, cmp_value)
CALL SHMEM_INT8_WAIT_UNTIL(ivar, cmp, cmp_value)
CALL SHMEM_WAIT(ivar, cmp_value)
CALL SHMEM_WAIT_UNTIL(ivar, cmp, cmp_value)

```

## DESCRIPTION

### Arguments

<b>OUT</b>	<i>ivar</i>	A remotely accessible integer variable that is being updated by another PE. If you are using <i>C/C++</i> , the type of <i>ivar</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>cmp</i>	The compare operator that compares <i>ivar</i> with <i>cmp_value</i> . <i>cmp</i> must be of type integer. If you are using <i>Fortran</i> , it must be of default kind. If you are using <i>C/C++</i> , the type of <i>cmp</i> should match that implied in the SYNOPSIS section.
<b>IN</b>	<i>cmp_value</i>	<i>cmp_value</i> must be of type integer. If you are using <i>C/C++</i> , the type of <i>cmp_value</i> should match that implied in the SYNOPSIS section. If you are using <i>Fortran</i> , <i>cmp_value</i> must be an integer of the same size and kind as <i>ivar</i> .

### API description

*shmem\_wait* and *shmem\_wait\_until* wait for *ivar* to be changed by a remote write or an atomic operation issued by a different PE. These routines can be used for point-to-point direct synchronization. A call to *shmem\_wait* does not return until some other PE writes a value, not equal to *cmp\_value*, into *ivar* on the waiting PE. A call to *shmem\_wait\_until* does not return until some other PE changes *ivar* to satisfy the condition implied by *cmp* and *cmp\_value*. This mechanism is useful when a PE needs to tell another PE that it has completed some action. The *shmem\_wait* routines return when *ivar* is no longer equal to *cmp\_value*. The *shmem\_wait\_until* routines return when the compare condition is true. The compare condition is defined by the *ivar* argument compared with the *cmp\_value* using the comparison operator, *cmp*.

If you are using *Fortran*, *ivar* must be a specific sized integer type according to the routine being called, as follows:

Routine	Type of <i>ivar</i>
<i>shmem_wait</i> , <i>shmem_wait_until</i>	default INTEGER
<i>shmem_int4_wait</i> , <i>shmem_int4_wait_until</i>	INTEGER*4
<i>shmem_int8_wait</i> , <i>shmem_int8_wait_until</i>	INTEGER*8

The following *cmp* values are supported:

CMP Value	Comparison
<i>_SHMEM_CMP_EQ</i>	Equal
<i>_SHMEM_CMP_NE</i>	Not equal
<i>_SHMEM_CMP_GT</i>	Greater than
<i>_SHMEM_CMP_LE</i>	Less than or equal to
<i>_SHMEM_CMP_LT</i>	Less than
<i>_SHMEM_CMP_GE</i>	Greater than or equal to

*C/C++*:

*Fortran:*

SHMEM_CMP_EQ	Equal
SHMEM_CMP_NE	Not equal
SHMEM_CMP_GT	Greater than
SHMEM_CMP_LE	Less than or equal to
SHMEM_CMP_LT	Less than
SHMEM_CMP_GE	Greater than or equal to

### Return Values

None.

### Note to implementors

Implementations must ensure that *shmem\_wait* and *shmem\_wait\_until* do not return before the update of the memory indicated by *ivar* is fully complete. Partial updates to the memory must not cause *shmem\_wait* or *shmem\_wait\_until* to return.

## EXAMPLES

The following call returns when variable *ivar* is not equal to 100:

```
INTEGER*8 IVAR
CALL SHMEM_INT8_WAIT(IVAR, INTEGER*8(100))
```

The following call to *SHMEM\_INT8\_WAIT\_UNTIL* is equivalent to the call to *SHMEM\_INT8\_WAIT* in example 1:

```
INTEGER*8 IVAR
CALL SHMEM_INT8_WAIT_UNTIL(IVAR, SHMEM_CMP_NE, INTEGER*8(100))
```

The following *C/C++* call waits until the sign bit in *ivar* is set by a transfer from a remote PE:

```
#include <stdio.h>
#include <shmem.h>

int ivar;
shmem_int_wait_until(&ivar, SHMEM_CMP_LT, 0);
```

The following *Fortran* example is in the context of a subroutine:

```
SUBROUTINE EXAMPLE()
INTEGER FLAG_VAR
COMMON/FLAG/FLAG_VAR
. . .
FLAG_VAR = FLAG_VALUE      ! initialize the event variable
. . .
IF (FLAG_VAR .EQ. FLAG_VALUE) THEN
    CALL SHMEM_WAIT(FLAG_VAR, FLAG_VALUE)
ENDIF
FLAG_VAR = FLAG_VALUE      ! reset the event variable for next time
. . .
END
```

## 8.7 Memory Ordering Routines

The following section discusses OpenSHMEM APIs that provide mechanisms to ensure ordering and/or delivery of *Put*, *AMO*, and memory store routines to symmetric data objects.

### 8.7.1 SHMEM\_FENCE

Assures ordering of delivery of *Put*, AMOs, and memory store routines to symmetric data objects.

#### SYNOPSIS

##### C/C++:

```
void shmem_fence(void);
```

##### FORTRAN:

```
CALL SHMEM_FENCE
```

#### DESCRIPTION

##### Arguments

None.

##### API description

This routine assures ordering of delivery of *Put*, AMOs, and memory store routines to symmetric data objects. All *Put*, AMOs, and memory store routines to symmetric data objects issued to a particular remote PE prior to the call to *shmem\_fence* are guaranteed to be delivered before any subsequent *Put*, AMOs, and memory store routines to symmetric data objects to the same PE. *shmem\_fence* guarantees order of delivery, not completion.

##### Return Values

None.

##### Notes

*shmem\_fence* only provides per-PE ordering guarantees and does not guarantee completion of delivery. There is a subtle difference between *shmem\_fence* and *shmem\_quiet*, in that, *shmem\_quiet* guarantees completion of *Put*, AMOs, and memory store routines to symmetric data objects which makes the updates visible to all other PEs.

The *shmem\_quiet* routine should be called if completion of PUT, AMOs, and memory store routines to symmetric data objects is desired when multiple remote PEs are involved.

#### EXAMPLES

The following *shmem\_fence* example is for C/C++ programs:

```
#include <stdio.h>
#include <shmem.h>

long dest[10] = {0};
int targ = 0;

int main(void)
{
    long source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int src = 99;
    shmem_init();
    if (shmem_my_pe() == 0) {
        shmem_long_put(dest, source, 10, 1); /*put1*/
        shmem_long_put(dest, source, 10, 2); /*put2*/
        shmem_fence();
        shmem_int_put(&targ, &src, 1, 1); /*put3*/
        shmem_int_put(&targ, &src, 1, 2); /*put4*/
    }
}
```

```

1     shmem_barrier_all(); /* sync sender and receiver */
2     printf("dest[0] on PE %d is %ld\n", shmem_my_pe(), dest[0]);
3     return 1;
4     }

```

*Put1* will be ordered to be delivered before *put3* and *put2* will be ordered to be delivered before *put4*.

## 8.7.2 SHMEM\_QUIET

Waits for completion of all outstanding *Put*, AMOs and memory store routines to symmetric data objects issued by a PE.

### SYNOPSIS

#### C/C++:

```
void shmem_quiet(void);
```

#### FORTTRAN:

```
CALL SHMEM_QUIET
```

### DESCRIPTION

#### Arguments

None.

#### API description

The *shmem\_quiet* routine ensures completion of *Put*, AMOs, and memory store routines on symmetric data objects issued by the calling PE. All *Put*, AMOs, memory store routines to symmetric data objects are guaranteed to be completed and visible to all PEs when *shmem\_quiet* returns.

#### Return Values

None.

#### Notes

*shmem\_quiet* is most useful as a way of ensuring completion of several *Put*, AMOs, and memory store routines to symmetric data objects initiated by the calling PE. For example, you might use *shmem\_quiet* to await delivery of a block of data before issuing another *Put*, which sets a completion flag on another PE.

*shmem\_quiet* is not usually needed if *shmem\_barrier\_all* or *shmem\_barrier* are called. The barrier routines wait for the completion of outstanding writes (*Put*, AMO, memory stores) to symmetric data objects on all PEs.

### EXAMPLES

The following example uses *shmem\_quiet* in a C/C++ program:

```

42 #include <stdio.h>
43 #include <shmem.h>
44
45 long dest[3] = {0};
46 int targ = 0;
47 long source[3] = {1, 2, 3};
48 int src = 90;
49
50 int main(void)
51 {

```



```

long x[3] = {0};
int y = 0;

shmem_init();
if (shmem_my_pe() == 0) {
    shmem_long_put(dest, source, 3, 1); /*put1*/
    shmem_int_put(&targ, &src, 1, 2); /*put2*/

    shmem_quiet();

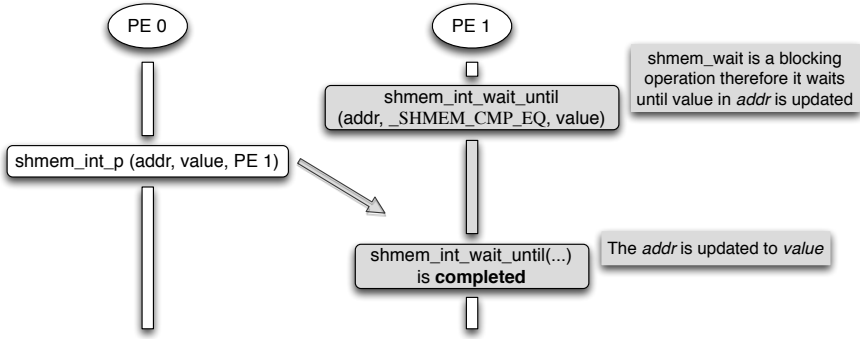
    shmem_long_get(x, dest, 3, 1); /*gets updated value from dest on PE 1 to local array x
    */
    shmem_int_get(&y, &targ, 1, 2); /*gets updated value from targ on PE 2 to local
    variable y*/
    printf("x: {%ld,%ld,%ld}\n",x[0],x[1],x[2]); /*x: {1,2,3}*/
    printf("y: %d\n", y); /*y: 90*/

    shmem_int_put(&targ, &src, 1, 1); /*put3*/
    shmem_int_put(&targ, &src, 1, 2); /*put4*/
}
shmem_barrier_all(); /* sync sender and receiver */
return 0;
}
    
```

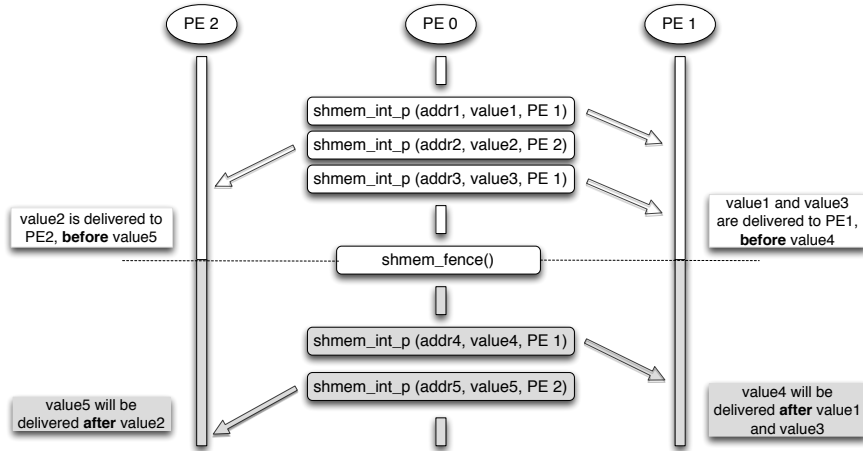
Put1 and put2 will be completed and visible before put3 and put4.

### 8.7.3 Synchronization and Communication Ordering in OpenSHMEM

When using the OpenSHMEM API, synchronization, ordering, and completion of communication become critical. The updates via *Put* routines, AMOs and store routines on symmetric data cannot be guaranteed until some form of synchronization or ordering is introduced by the program user. The table below gives the different synchronization and ordering choices, and the situations where they may be useful.

OpenSHMEM API	Working of OpenSHMEM API
Point-to-point synchronization <i>shmem_wait</i> , <i>shmem_wait_until</i>	 <p>The diagram illustrates the interaction between two Process Elements (PE 0 and PE 1). PE 0 sends a message to PE 1 using the <code>shmem_int_p(addr, value, PE 1)</code> routine. PE 1 then calls <code>shmem_int_wait_until(addr, _SHMEM_CMP_EQ, value)</code>, which is a blocking operation. A callout box explains: "shmem_wait is a blocking operation therefore it waits until value in addr is updated". Once the data is received, a second callout box states: "The addr is updated to value" and "shmem_int_wait_until(...) is completed".</p> <p>Waits for a symmetric variable to be updated by a remote PE. Should be used when computation on the local PE cannot proceed without the value that the remote PE is to update.</p>

Ordering puts issued by a local PE  
*shmem\_fence*

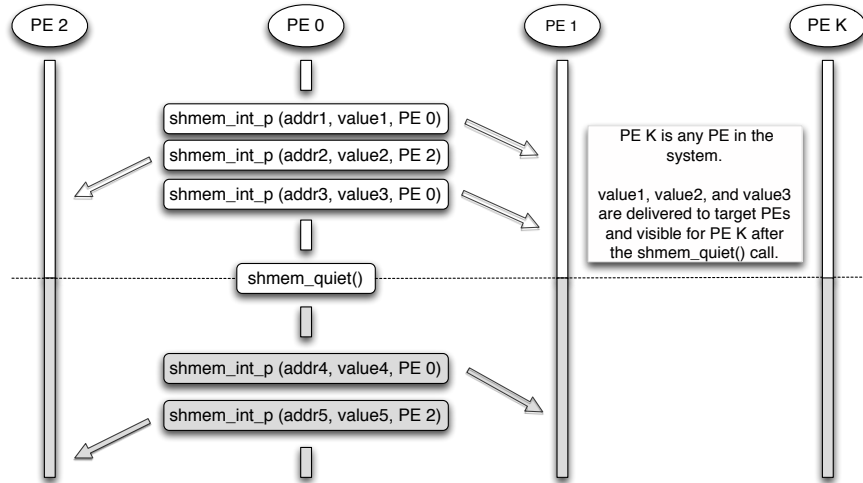


All *Put* routines, AMOs and store routines on symmetric data issued to same PE are guaranteed to be delivered before Puts (to the same PE) issued after the *fence* call.

**OpenSHMEM API**

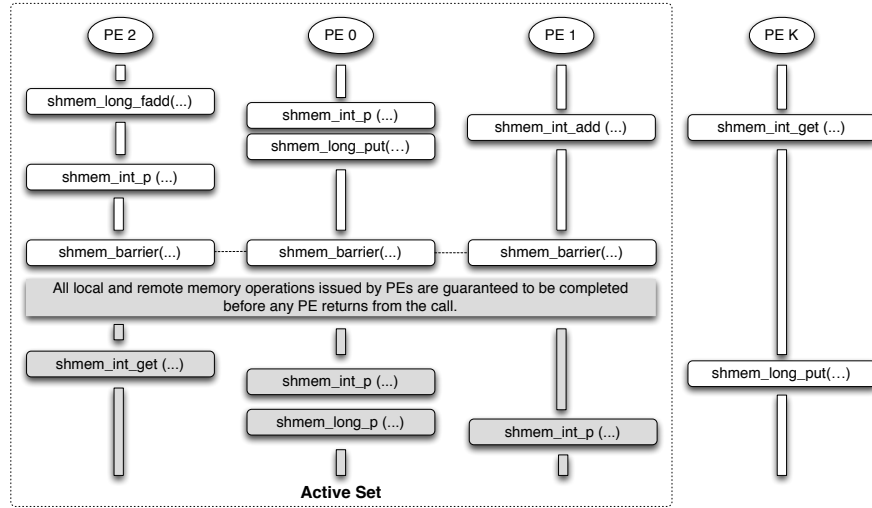
**Working of OpenSHMEM API**

Ordering puts issued by all PE  
*shmem\_quiet*



All *Put* routines, AMOs and store routines on symmetric data issued by a local PE to all remote PEs are guaranteed to be completed and visible once quiet returns. This routine should be used when all remote writes issued by a local PE need to be visible to all other PEs before the local PE proceeds.

Collective synchronization over an *Active set*  
*shmem\_barrier*

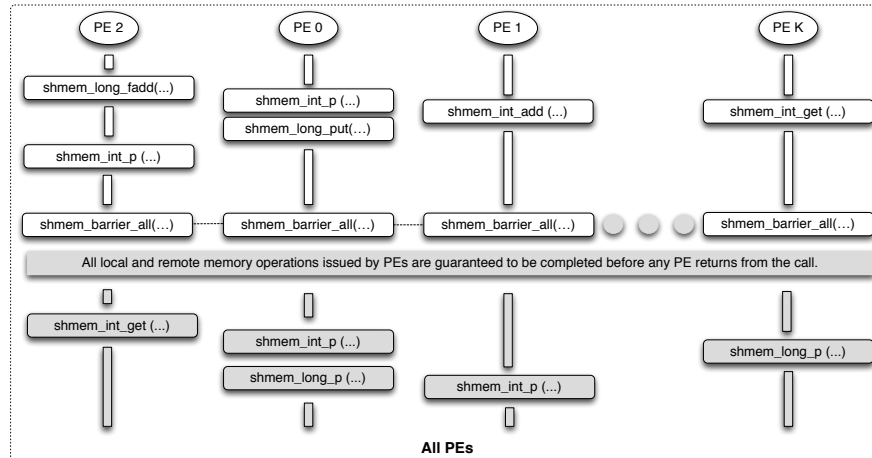


All local and remote memory operations issued by all PEs within the *Active set* are guaranteed to be completed before any PE in the *Active set* returns from the call. Additionally, no PE may return from the barrier until all PEs in the *Active set* have entered the same barrier call. This routine should be used when synchronization as well as completion of all stores and remote memory updates via OpenSHMEM is required over a sub set of the executing PEs.

**OpenSHMEM API**

**Working of OpenSHMEM API**

Collective synchronization over all PEs  
*shmem\_barrier\_all*



All local and remote memory operations issued by all PEs are guaranteed to be completed before any PE returns from the call. Additionally no PE shall return from the barrier until all PEs have entered the same *shmem\_barrier\_all* call. This routine should be used when synchronization as well as completion of all stores and remote memory updates via OpenSHMEM is required over all PEs.

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48

## 8.8 Distributed Locking Routines

The following section discusses OpenSHMEM locks as a mechanism to provide mutual exclusion. Three routines are available for distributed locking, *set*, *test* and *clear*.

### 8.8.1 SHMEM\_LOCK

Releases, locks, and tests a mutual exclusion memory lock.

#### SYNOPSIS

##### C/C++:

```
void shmem_clear_lock(long *lock);
void shmem_set_lock(long *lock);
int shmem_test_lock(long *lock);
```

##### FORTRAN:

```
INTEGER lock, SHMEM_TEST_LOCK
CALL SHMEM_CLEAR_LOCK(lock)
CALL SHMEM_SET_LOCK(lock)
I = SHMEM_TEST_LOCK(lock)
```

#### DESCRIPTION

##### Arguments

<b>IN</b>	<i>lock</i>	A symmetric data object that is a scalar variable or an array of length <i>I</i> . This data object must be set to 0 on all PEs prior to the first use. <i>lock</i> must be of type <i>long</i> . If you are using <i>Fortran</i> , it must be of default kind.
-----------	-------------	---

##### API description

The *shmem\_set\_lock* routine sets a mutual exclusion lock after waiting for the lock to be freed by any other PE currently holding the lock. Waiting PEs are assured of getting the lock in a first-come, first-served manner. The *shmem\_clear\_lock* routine releases a lock previously set by *shmem\_set\_lock* after ensuring that all local and remote stores initiated in the critical region are complete. The *shmem\_test\_lock* routine sets a mutual exclusion lock only if it is currently cleared. By using this routine, a PE can avoid blocking on a set lock. If the lock is currently set, the routine returns without waiting. These routines are appropriate for protecting a critical region from simultaneous update by multiple PEs.

##### Return Values

The *shmem\_test\_lock* routine returns 0 if the lock was originally cleared and this call was able to set the lock. A value of 1 is returned if the lock had been set and the call returned without waiting to set the lock.

##### Notes

The term symmetric data object is defined in Introduction. The lock variable should always be initialized to zero and accessed only by the OpenSHMEM locking API. Changing the value of the lock variable by other means without using the OpenSHMEM API, can lead to undefined behavior.

#### EXAMPLES

The following example uses *shmem\_lock* in a C program.

```
#include <stdio.h>
#include <unistd.h>
```

```

#include <shmem.h>
long L = 0;

int main(int argc, char **argv)
{
    int me, slp;
    shmem_init();
    me = shmem_my_pe();
    slp = 1;
    shmem_barrier_all();
    if (me == 1)
        sleep (3);
    shmem_set_lock (&L);
    printf("%d: sleeping %d second%s...\n", me, slp, slp == 1 ? "" : "s");
    sleep(slp);
    printf("%d: sleeping...done\n", me);
    shmem_clear_lock (&L);
    shmem_barrier_all();
    return 0;
}

```

## 8.9 Cache Management

All of these routines are deprecated and are provided for backwards compatibility. Implementations must include all items in this section and the routines should function properly, while notifying the user about deprecation of the functionality.

### 8.9.1 SHMEM\_CACHE

Controls data cache utilities.

#### SYNOPSIS

##### C/C++:

```

void shmem_clear_cache_inv(void);
void shmem_set_cache_inv(void);
void shmem_clear_cache_line_inv(void *dest);
void shmem_set_cache_line_inv(void *dest);
void shmem_udcflush(void);
void shmem_udcflush_line(void *dest);

```

##### FORTRAN:

```

CALL SHMEM_CLEAR_CACHE_INV
CALL SHMEM_SET_CACHE_INV
CALL SHMEM_SET_CACHE_LINE_INV(dest)
CALL SHMEM_UDCFLUSH
CALL SHMEM_UDCFLUSH_LINE(dest)

```

#### DESCRIPTION

##### Arguments

<b>IN</b>	<i>dest</i>	A data object that is local to the PE. <i>dest</i> can be of any noncharacter type. If you are using <i>Fortran</i> , it can be of any kind.
-----------	-------------	--

##### API description

*shmem\_set\_cache\_inv* enables automatic cache coherency mode.

*shmem\_set\_cache\_line\_inv* enables automatic cache coherency mode for the cache line associated with the address of *dest* only.

1 *shmem\_clear\_cache\_inv* disables automatic cache coherency mode previously enabled by *shmem\_set\_cache*  
2 *\_inv* or *shmem\_set\_cache\_line\_inv*.

3 *shmem\_udcflush* makes the entire user data cache coherent.

4 *shmem\_udcflush\_line* makes coherent the cache line that corresponds with the address specified by *dest*.

#### 6 **Return Values**

7 None.

#### 9 **Notes**

10 These routines have been retained for improved backward compatibility with legacy architectures. They  
11 are not required to be supported by implementing them as *no-ops* and where used, they may have no effect  
12 on cache line states.

#### 14 **EXAMPLES**

15 None.

# Annex A

## Writing OpenSHMEM Programs

### Incorporating OpenSHMEM into Programs

In this section, we describe how to write a “Hello World” OpenSHMEM program. To write a “Hello World” OpenSHMEM program we need to:

- Add the include file `shmem.h` (for *C*) or `shmem.fh` (for *Fortran*).
- Add the initialization call `shmem_init`, (line 9).
- Use OpenSHMEM calls to query the the total number of PEs (line 10) and PE id (line 11).
- There is no explicit finalize call; either a return from `main()` (line 13) or an explicit `exit()` acts as an implicit OpenSHMEM finalization.
- In OpenSHMEM the order in which lines appear in the output is not fixed as PEs execute asynchronously in parallel.

```
1 #include <stdio.h>
2 #include <shmem.h>          /* The shmem header file */
3
4 int
5 main (int argc, char *argv[])
6 {
7     int nprocs, me;
8
9     shmem_init ();
10    nprocs = shmem_n_pes ();
11    me = shmem_my_pe ();
12    printf ("Hello from %d of %d\n", me, nprocs);
13    return 0;
14 }
```

Listing A.1: Expected Output (4 processors)

```
1 Hello from 0 of 4
2 Hello from 2 of 4
3 Hello from 3 of 4
4 Hello from 1 of 4
```

OpenSHMEM also has a *Fortran* API, so for completeness we will now give the same program written in *Fortran*, in listing A:

```

1
2 1 program hello
3 2
4 3   include 'shmem.fh'
5 4   integer :: shmem_my_pe, shmem_n_pes
6 5
7 6   integer :: npes, me
8 7
9 8   call shmem_init ()
10 9   npes = shmem_n_pes ()
11 10  me = shmem_my_pe ()
12 11
13 12  write (*, 1000) me, npes
14 13
15 14 1000 format ('Hello from', 1X, I4, 1X, 'of', 1X, I4)
16 15
17 16 end program hello

```

Listing A.2: Expected Output (4 processors)

```

16 1 Hello from 0 of 4
17 2 Hello from 2 of 4
18 3 Hello from 3 of 4
19 4 Hello from 1 of 4

```

The following example shows a more complex OpenSHMEM program that illustrates the use of symmetric data objects. Note the declaration of the *static short dest* array and its use as the remote destination in OpenSHMEM short *Put*. The use of the *static* keyword results in the *dest* array being symmetric on PE 0 and PE 1. Each PE is able to transfer data to the *dest* array by simply specifying the local address of the symmetric data object which is to receive the data. This aids programmability, as the address of the *dest* need not be exchanged with the active side (PE 0) prior to the RMA (Remote Memory Access) routine. Conversely, the declaration of the *short source* array is asymmetric. Because the *Put* handles the references to the *source* array only on the active (local) side, the asymmetric *source* object is handled correctly.



```

1 #include <stdio.h>
2 #include <shmem.h>
3 #define SIZE 16
4 int
5 main(int argc, char* argv[])
6 {
7     short source[SIZE];
8     static short dest[SIZE];
9     int i, npes;
10    shmem_init();
11    npes = shmem_n_pes();
12    if (shmem_my_pe() == 0) {
13        /* initialize array */
14        for(i = 0; i < SIZE; i++)
15            source[i] = i;
16        /* local, not symmetric */
17        /* static makes it symmetric */
18        /* put "size" words into dest on each PE */
19        for(i = 1; i < npes; i++)
20            shmem_short_put(dest, source, SIZE, i);
21    }
22    shmem_barrier_all(); /* sync sender and receiver */
23    if (shmem_my_pe() != 0) {
24        printf("dest on PE %d is \t", shmem_my_pe());
25        for(i = 0; i < SIZE; i++)
26            printf("%hd \t", dest[i]);
27        printf("\n");
28    }
29    shmem_finalize();
30    return 0;
31 }

```

Listing A.3: Expected Output (4 processors)

```

1 dest on PE 1 is 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 dest on PE 2 is 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
3 dest on PE 3 is 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```

## Annex B

# Compiling and Running Programs

As of this writing, the OpenSHMEM specification is silent regarding how OpenSHMEM programs are compiled, linked and run. This section shows some examples of how wrapper programs are utilized in the OpenSHMEM Reference Implementation to compile and launch programs.

## 1 Compilation

### Programs written in *C*

The OpenSHMEM Reference Implementation provides a wrapper program named **oshcc**, to aid in the compilation of *C* programs, the wrapper could be called as follows:

```
oshcc <compiler options> -o myprogram myprogram.c
```

Where the *<compiler options>* are options understood by the underlying *C* compiler.

### Programs written in *C++*

The OpenSHMEM Reference Implementation provides a wrapper program named **oshCC**, to aid in the compilation of *C++* programs, the wrapper could be called as follows:

```
oshCC <compiler options> -o myprogram myprogram.cpp
```

Where the *<compiler options>* are options understood by the underlying *C++* compiler called by **oshCC**.

### Programs written in *Fortran*

The OpenSHMEM Reference Implementation provides a wrapper program named **oshfort**, to aid in the compilation of *Fortran* programs, the wrapper could be called as follows:

```
oshfort <compiler options> -o myprogram myprogram.f
```

Where the *<compiler options>* are options understood by the underlying *Fortran* compiler called by **oshfort**.

## 2 Running Programs

The OpenSHMEM Reference Implementation provides a wrapper program named **oshrun**, to launch OpenSHMEM programs, the wrapper could be called as follows:

```
oshrun <additional options> -np <#> <program> <program arguments>
```

The program arguments for **oshrun** are:

⟨additional options⟩	Options passed to the underlying launcher.
-np ⟨#⟩	The number of PEs to be used in the execution.
⟨program⟩	The program executable to be launched.
⟨program arguments⟩	Flags and other parameters to pass to the program.

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48

## Annex C

# Undefined Behavior in OpenSHMEM

The specification provides guidelines to the expected behavior of various library routines. In cases where routines are improperly used or the input is not in accordance with the specification, undefined behavior may be observed. Depending on the implementation there are many interpretations of undefined behavior.

Inappropriate Usage	Undefined Behavior
Uninitialized library	If OpenSHMEM is not initialized through a call to <i>shmem_init</i> , subsequent accesses to OpenSHMEM routines have undefined results. An implementation may choose, for example, to try to continue or abort immediately upon the first call to an uninitialized routine.
Accessing non-existent PEs	If a communications routine accesses a non-existent PE, then the OpenSHMEM library can choose to handle this situation in an implementation-defined way. For example, the library may issue an error message saying that the PE accessed is outside the range of accessible PEs, or may exit without a warning.
Use of non-symmetric variables	Some routines require remotely accessible variables to perform their function. A <i>Put</i> to a non-symmetric variable can be trapped where possible and the library can abort the program. Another implementation may choose to continue either with a warning or silently.
Non-symmetric variables	The symmetric memory management routines are collectives, which means that all PEs in the program must issue the same <i>shmem_malloc</i> call with the same size request. Program behavior after a mismatched <i>shmem_malloc</i> call is undefined.
Use of NULL pointers with non-zero <i>len</i> specified	In any OpenSHMEM routine that takes a pointer and <i>len</i> describing the number of elements in that pointer, NULL may not be specified for the pointer unless the corresponding <i>len</i> is also specified as zero. Otherwise, the resulting behavior is undefined. The following cases summarize this behavior: <ul style="list-style-type: none"><li>• <i>len</i> is 0, pointer is NULL: supported.</li><li>• <i>len</i> is not 0, pointer is NULL: undefined behavior.</li><li>• <i>len</i> is 0, pointer is not NULL: supported.</li><li>• <i>len</i> is not 0, pointer is not NULL: supported.</li></ul>
Multiple calls to <i>shmem_init</i>	In an OpenSHMEM program where <i>shmem_init</i> has already be called, any subsequent calls to <i>shmem_init</i> result in undefined behavior.

## Annex D

# Interoperability with other Programming Models

### 1 MPI Interoperability

OpenSHMEM routines can be used in conjunction with MPI routines in the same program. For example, on SGI systems, programs that use both MPI and OpenSHMEM routines call *MPI\_Init* and *MPI\_Finalize* but omit the call to the *shmem\_init* routine. OpenSHMEM PE numbers are equal to the MPI rank within the *MPI\_COMM\_WORLD* environment variable. Note that this precludes use of OpenSHMEM routines between processes in different *MPI\_COMM\_WORLD*s. MPI processes started using the *MPI\_Comm\_spawn* routine, for example, cannot use OpenSHMEM routines to communicate with their parent MPI processes.

On SGI systems where MPI jobs use TCP/sockets for inter-host communication, OpenSHMEM routines can be used to communicate with processes running on the same host. The *shmem\_pe\_accessible* routine can be used to determine if a remote PE is accessible via OpenSHMEM communication from the local PE. When running an MPI program involving multiple executable files, OpenSHMEM routines can be used to communicate with processes running from the same or different executable files, provided that the communication is limited to symmetric data objects. On these systems, static memory such as a *Fortran* common block or *C* global variable, is symmetric between processes running from the same executable file, but is not symmetric between processes running from different executable files. Data allocated from the symmetric heap (*shmem\_malloc* or *shpalloc*) is symmetric across the same or different executable files. The routine *shmem\_addr\_accessible* can be used to determine if a local address is accessible via OpenSHMEM communication from a remote PE.

Another important feature of these systems is that the *shmem\_pe\_accessible* routine returns *TRUE* only if the remote PE is a process running from the same executable file as the local PE, indicating that full OpenSHMEM support (static memory and symmetric heap) is available. When using OpenSHMEM routines within an MPI program, the use of MPI memory placement environment variables is required when using non-default memory placement options.

## Annex E

# History of OpenSHMEM

SHMEM has a long history as a parallel programming model, having been used extensively on a number of products since 1993, including Cray T3D, Cray X1E, the Cray XT3/4, SGI Origin, SGI Altix, clusters based on the Quadrics interconnect, and to a very limited extent, Infiniband based clusters.

- A SHMEM Timeline

- Cray SHMEM

- \* SHMEM first introduced by Cray Research Inc. in 1993 for Cray T3D
    - \* Cray is acquired by SGI in 1996
    - \* Cray is acquired by Tera in 2000 (MTA)
    - \* Platforms: Cray T3D, T3E, C90, J90, SV1, SV2, X1, X2, XE, XMT, XT

- SGI SHMEM

- \* SGI purchases Cray Research Inc. and SHMEM was integrated into SGI's Message Passing Toolkit (MPT)
    - \* SGI currently owns the rights to SHMEM and OpenSHMEM
    - \* Platforms: Origin, Altix 4700, Altix XE, ICE, UV
    - \* SGI was purchased by Rackable Systems in 2009
    - \* SGI and Open Source Software Solutions, Inc. (OSSS) signed a SHMEM trademark licensing agreement, in 2010

- Other Implementations

- \* Quadrics (Vega UK, Ltd.)
    - \* Hewlett Packard
    - \* GPSHMEM
    - \* IBM
    - \* QLogic
    - \* Mellanox
    - \* University of Florida

- OpenSHMEM Implementations

- SGI OpenSHMEM
  - University of Houston - OpenSHMEM Reference Implementation
  - Mellanox ScalableSHMEM
  - Portals-SHMEM
  - IBM OpenSHMEM

## Annex F

# OpenSHMEM Specification and Deprecated API

For the OpenSHMEM Specification(s), deprecation is the process of identifying API that is supported but no longer recommended for use by program users. For OpenSHMEM library users, said API **must** be supported until clearly indicated as otherwise by the Specification. In this chapter we will record the API that has been deprecated, the OpenSHMEM Specification that effected the deprecation, and if the feature is supported in the current version of the specification.

Deprecated API	Deprecated Since	Currently Supported(?)	Replaced By
<i>_my_pe</i>	1.2	Yes	<i>shmem_my_pe</i>
<i>_num_pes</i>	1.2	Yes	<i>shmem_n_pes</i>
<i>shm_malloc</i>	1.2	Yes	<i>shmem_malloc</i>
<i>shfree</i>	1.2	Yes	<i>shmem_free</i>
<i>shrealloc</i>	1.2	Yes	<i>shmem_realloc</i>
<i>shmalign</i>	1.2	Yes	<i>shmem_align</i>
<i>start_pes</i>	1.2	Yes	<i>shmem_init</i>
<i>SHMEM_PUT</i>	1.2	Yes	<i>SHMEM_PUT8</i> or <i>SHMEM_PUT64</i>

# Annex G

## Changes to this Document

### 1 Version 1.2

This section summarizes the changes from the OpenSHMEM specification Version 1.1 to Version 1.2. A major change in this version is that it improves upon the execution model described in 1.1 by introducing an explicit *shmem\_finalize* library call. This provides a collective mechanism of exiting an OpenSHMEM program and releasing resources used by the library.

The following list describes the specific changes in 1.2:

- Added specification of *pSync* initialization for all routines that use it.
- Replaced all placeholder variable names *target* with *dest* to avoid confusion with Fortran ‘target’ keyword.
- New Execution Model for exiting/finishing OpenSHMEM programs.  
See Section 4.
- New library constants to support API that query version and name information.  
See Section 6.
- New API *shmem\_init* to provide mechanism to start an OpenSHMEM program and replace deprecated *start\_pes*.  
See Section 8.1.
- Deprecation of *\_my\_pe* and *\_num\_pes* routines.  
See Sections 8.1.2 and 8.1.3.
- New API *shmem\_finalize* to provide collective mechanism to cleanly exit an OpenSHMEM program and release resources.  
See Section 8.1.4.
- New API *shmem\_global\_exit* to provide mechanism to exit an OpenSHMEM program.  
See Section 8.1.5.
- Clarification related to the address of the referenced object in *shmem\_ptr*.  
See Section 8.1.8.
- New API to query the version and name information.  
See Section 8.1.9 and 8.1.10.
- OpenSHMEM library API normalization. All C symmetric memory management API begins with *shmem\_*.  
See Section 8.2.1.
- Notes and clarifications added to *shmem\_malloc*.  
See Section 8.2.1.



- Deprecation of Fortran API routine *SHMEM\_PUT*.  
See Section 8.3.
- Clarification related to *shmem\_wait*.  
See Section 8.6.1.
- Undefined behavior for null pointers without zero counts added.  
See Section C
- Addition of new Annex for clearly specifying deprecated API and its support in the existing specification version.  
See Section F.

## 2 Version 1.1

This section summarizes the changes from the OpenSHMEM specification Version 1.0 to the Version 1.1. A major change in this version is that it provides an accurate description of OpenSHMEM interfaces so that they are in agreement with the SGI specification. This version also explains OpenSHMEM 's programming, memory, and execution model. The document was thoroughly changed to improve the readability of specification and usability of interfaces. The code examples were added to demonstrate the usability of API. Additionally, diagrams were added to help understand the subtle semantic differences of various operations.

The following list describes the specific changes in 1.1:

- Clarifications of the completion semantics of memory synchronization interfaces.  
See Section 8.7.
- Clarification of the completion semantics of memory load and store operations in context of *shmem\_barrier\_all* and *shmem\_barrier* routines.  
See Section 8.5 and 8.5.1.
- Clarification of the completion and ordering semantics of *shmem\_quiet* and *shmem\_fence*.  
See Section 8.7.1 and 8.7.
- Clarifications of the completion semantics of RMA and AMO routines.  
See Sections 8.3 and 8.4
- Clarifications of the memory model and the memory alignment requirements for symmetric data objects.  
See Section 3.
- Clarification of the execution model and the definition of a PE.  
See Section 4
- Clarifications of the semantics of *shmem\_pe\_accessible* and *shmem\_addr\_accessible*.  
See Section 8.1.6 and 8.1.7.
- Added an annex on interoperability with MPI.  
See Annex D.
- Added examples to the different interfaces.
- Clarification of the naming conventions for constant in *C* and *Fortran*.  
See Section 6 and 8.6.1.
- Added API calls: *shmem\_char\_p*, *shmem\_char\_g*.  
See Sections 8.3.1 and 8.3.4.
- Removed API calls: *shmem\_char\_put*, *shmem\_char\_get*.  
See Sections 8.3 and 8.3.3.

- 1 • The usage of *ptrdiff\_t*, *size\_t*, and *int* in the interface signature was made consistent with the description.  
2 See Sections 8.5, 8.3.3, and 8.3.6.
- 3 • Revised *shmem\_barrier* example.  
4 See Section 8.5.1.
- 5 • Clarification of the initial value of *pSync* work arrays for *shmem\_barrier*.  
6 See Section 8.5.1.
- 7 • Clarification of the expected behavior when multiple *start\_pes* calls are encountered has been clarified.  
8 See Section 8.1.11.
- 9 • Corrected the definition of atomic increment operation.  
10 See Section 8.4.4.
- 11 • Clarification of the size of the symmetric heap and when it is set.  
12 See Section 8.2.1.
- 13 • Clarification of the integer and real sizes for *Fortran* API.  
14 See Sections 8.4, 8.4.1, 8.4.2, 8.4.3, 8.4.4, and 8.4.5.
- 15 • Clarification of the expected behavior on program *exit*.  
16 See Section 4, Execution Model.
- 17 • More detailed description for the progress of OpenSHMEM operations provided.  
18 See Section 4.1.
- 19 • Clarification of naming convention for non-standard interfaces and their inclusion in *shmemx.h*.  
20 See Section 5.
- 21 • Various fixes to OpenSHMEM code examples across the specification to include appropriate header files.
- 22 • Removing requirement that implementations should detect size mismatch and return error information for *shmal-*  
23 *loc* and ensuring consistent language.  
24 See Sections 8.2.1 and Annex C.
- 25 • Fortran programming fixes for examples.  
26 See Sections 8.5.4 and 8.6.1.
- 27 • Clarifications of the reuse *pSync* and *pWork* across collectives.  
28 See Sections 8.5, 8.5.2, 8.5.4 and 8.5.4.
- 29 • Name changes for UV and ICE for SGI systems.  
30 See Annex E.