OpenSHMEM

Application Programming Interface

http://www.openshmem.org/

Version 1.4

14th December 2017

Development by

- For a current list of contributors and collaborators please see http://www.openshmem.org/site/Contributors/
- For a current list of OpenSHMEM implementations and tools, please see http://openshmem.org/site/Links#impl/

Sponsored by

- U.S. Department of Defense (DoD)
  http://www.defense.gov/
- Oak Ridge National Laboratory (ORNL)
  http://www.ornl.gov/
- Los Alamos National Laboratory (LANL)
  http://www.lanl.gov/

Current Authors and Collaborators

- Matthew Baker, ORNL
- Swen Boehm, ORNL
- Aurelien Bouteiller, University of Tenneesee at Knoxville (UTK)
- Barbara Chapman, Stonybrook University (SBU)
- Robert Cernohous, Cray Inc.
- James Culhane, LANL
- Tony Curtis, SBU
- James Dinan, Intel
- Mike Dubman, Mellanox
- Karl Feind, Hewlett Packard Enterprise (HPE)
- Manjunath Gorentla Venkata, ORNL
- Max Grossman, Rice University
- Khaled Hamidouche, Advanced Micro Devices (AMD)
- Jeff Hammond, Intel
- Yossi Itigin, Mellanox
- Bryant Lam, DoD
- David Knaak, Cray Inc.
- Jeff Kuehn, LANL
- Jens Manser, DoD
- Tiffany M. Mintz, ORNL
- David Ozog, Intel
- Nicholas Park, DoD
- Steve Poole, Open Source Software Solutions (OSSS)
- Wendy Poole, OSSS
- Swaroop Pophale, ORNL

- Sreeram Potluri, NVIDIA
- Howard Pritchard, LANL
- Naveen Ravichandrasekaran, Cray Inc.
- Michael Raymond, HPE
- James Ross, Army Research Laboratory (ARL)
- Pavel Shamis, ARM Inc.
- Sameer Shende, University of Oregon (UO)
- Lauren Smith, DoD

Alumni Authors and Collaborators

- Amrita Banerjee, University of Houston (UH)
- Monika ten Bruggencate, Cray Inc.
- Eduardo D'Azevedo, ORNL
- Oscar Hernandez, ORNL
- Gregory Koenig, ORNL
- Graham Lopez, ORNL
- Ricardo Mauricio, UH
- Ram Nanjegowda, UH
- Aaron Welch, ORNL

Acknowledgments

The OpenSHMEM specification belongs to Open Source Software Solutions, Inc. (OSSS), a non-profit organization, under an agreement with HPE. For a current list of Contributors and Collaborators, please see http://www.openshmem.org/site/Contributors/. We gratefully acknowledge support from Oak Ridge National Laboratory's Extreme Scale Systems Center and the continuing support of 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.

OpenSHMEM 1.4 is dedicated to the memory of David Charles Knaak. David was a highly involved colleague and contributor to the entire OpenSHMEM project. He will be missed.

Contents

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, many existing vendors support 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[1] effort is driven by the DoD with continuous input from the OpenSHMEM community. To see all of the contributors and participants for the OpenSHMEM API, please see: `http://www.openshmem.org/site/Contributors`. In addition to the specification, the effort includes 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 only 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*[2]. The *C++* interface is currently the same as that for *C*. Unlike Unified Parallel C, *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, where the PEs are either single or multithreaded.
   (b) *Query*: The local PE may get the 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**
   (a) *Allocation*: All executing PEs must participate in the allocation of a symmetric data object with identical arguments.

[1] The OpenSHMEM specification is owned by Open Source Software Solutions Inc., a non-profit organization, under an agreement with HPE.
[2] As of OpenSHMEM 1.4, the *Fortran* interface has been deprecated.

(b) *Deallocation*: All executing PEs must participate in the deallocation of the same symmetric data object with identical arguments.

(c) *Reallocation*: All executing PEs must participate in the reallocation of the same symmetric data object with identical arguments.

3. **Communication Management**

(a) *Contexts*: Contexts are containers for communication operations. Each context provides an environment where the operations performed on that context are ordered and completed independently of other operations performed by the application.

4. **Remote Memory Access**

(a) *Put*: The local PE specifies the *source* data object (private or symmetric) that is copied to the symmetric data object on the remote PE.

(b) *Get*: The local PE specifies the symmetric data object on the remote PE that is copied to a data object (private or symmetric) on the local PE.

5. **Atomics**

(a) *Swap*: The PE initiating the swap gets the old value of a symmetric data object from a remote PE and copies a new value to that symmetric data object on the remote PE.

(b) *Increment*: The PE initiating the increment adds 1 to the symmetric data object on the remote PE.

(c) *Add*: The PE initiating the add specifies the value to be added to the symmetric data object on the remote PE.

(d) *Bitwise Operations*: The PE initiating the bitwise operation specifies the operand value to the bitwise operation to be performed on the symmetric data object on the remote PE.

(e) *Compare and Swap*: The PE initiating the swap gets the old value of the symmetric data object based on a value to be compared and copies a new value to the symmetric data object on the remote PE.

(f) *Fetch and Increment*: The PE initiating the increment adds 1 to the symmetric data object on the remote PE and returns with the old value.

(g) *Fetch and Add*: The PE initiating the add specifies the value to be added to the symmetric data object on the remote PE and returns with the old value.

(h) *Fetch and Bitwise Operations*: The PE initiating the bitwise operation specifies the operand value to the bitwise operation to be performed on the symmetric data object on the remote PE and returns the old value.

6. **Synchronization and Ordering**

(a) *Fence*: The PE calling fence ensures ordering of *Put*, AMO, and memory store operations to symmetric data objects with respect to a specific destination PE.

(b) *Quiet*: The PE calling quiet ensures remote completion of remote access operations and stores to symmetric data objects.

(c) *Barrier*: All or some PEs collectively synchronize and ensure completion of all remote and local updates prior to any PE returning from the call.

7. **Collective Communication**

(a) *Broadcast*: The *root* PE specifies a symmetric data object to be copied to a symmetric data object on one or more remote PEs (not including itself).

(b) *Collection*: All PEs participating in the routine get the result of concatenated symmetric objects contributed by each of the PEs in another symmetric data object.

(c) *Reduction*: All PEs participating in the routine get the result of an associative binary routine over elements of the specified symmetric data object on another symmetric data object.

(d) *All-to-All*: All PEs participating in the routine exchange a fixed amount of contiguous or strided data with all other PEs in the active set.

8. **Mutual Exclusion**

(a) *Set Lock*: The PE acquires exclusive access to the region bounded by the symmetric *lock* variable.

(b) *Test Lock*: The PE tests the symmetric *lock* variable for availability.

(c) *Clear Lock*: The PE which has previously acquired the *lock* releases it.

— deprecation start —

9. **Data Cache Control**

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

— deprecation end —

3 Memory Model

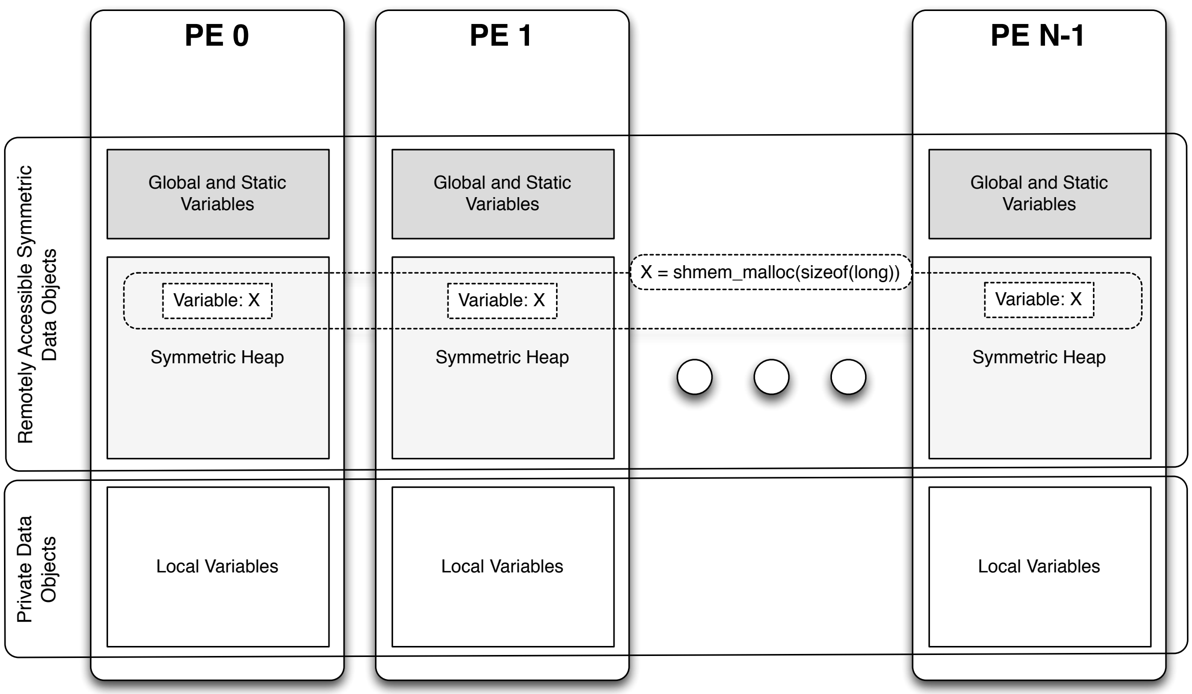


Figure 1: OpenSHMEM Memory Model

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 Data Objects*. Each symmetric data object has a corresponding object with the same name, type, and size on all PEs where that object is accessible via the OpenSHMEM API[3]. (For the definition of what is accessible, see the descriptions for *shmem_pe_accessible* and *shmem_addr_accessible* in sections 9.1.6 and 9.1.7.) Symmetric data objects accessed via typed and type-generic

[3] For efficiency reasons, the same offset (from an arbitrary memory address) for symmetric data objects might be used on all PEs. Further discussion about symmetric heap layout and implementation efficiency can be found in section 9.3.1

OpenSHMEM interfaces are required to be naturally aligned based on their type requirements and underlying architecture. In OpenSHMEM the following kinds of data objects are symmetric:

- — deprecation start —
  Fortran data objects in common blocks or with the *SAVE* attribute. These data objects must not be defined in a dynamic shared object (DSO).
  — deprecation end —
- Global and static *C* and *C++* variables. These data objects must not be defined in a DSO.
- — deprecation start —
  Fortran arrays allocated with *shpalloc*
  — deprecation end —
- *C* and *C++* data allocated by OpenSHMEM memory management routines (Section 9.3)

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.

3.1 Atomicity Guarantees

OpenSHMEM contains a number of routines that operate on symmetric data atomically (Section 9.7). These routines guarantee that accesses by OpenSHMEM's atomic operations with the same datatype 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.

4 Execution Model

An OpenSHMEM program consists of a set of OpenSHMEM processes called PEs that execute in an SPMD-like model where each PE can take a different execution path. For example, a PE can be implemented using an OS process. The PEs may be either single or multithreaded. 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*[4] or *shmem_init_thread* before using any of the other OpenSHMEM library routines. An OpenSHMEM program concludes its use of the OpenSHMEM library when all PEs call *shmem_finalize* or any PE calls *shmem_global_exit*. During a call to *shmem_finalize*, the OpenSHMEM library must complete all pending communication and release all the resources associated to the library using an implicit collective synchronization across PEs. 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 one less than the total number of PEs. 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 of the OpenSHMEM program.

[4] *start_pes* has been deprecated as of OpenSHMEM 1.2

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*, *atomic*, etc) and complete communication operations with that computationally-bound PE without that PE issuing any explicit OpenSHMEM calls. One-sided 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.

5 Language Bindings and Conformance

OpenSHMEM provides ISO *C* and *Fortran 90* language bindings. As of OpenSHMEM 1.4, the *Fortran* API is deprecated. For rationale and considerations of future *Fortran* use of OpenSHMEM, see Section 2.13.

Any implementation that provides both *C* and *Fortran* bindings can claim conformance to the specification. Alternatively, an implementation may claim conformance only with respect to one of those languages. For example, an implementation that provides 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 *shmem.h* for *C* and *shmem.fh* for *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* and *shmemx.fh* include files for *C* and *Fortran* language bindings, respectively. These header files must exist, even if no extensions are provided. Any extensions shall use the *shmemx_* prefix for all routine, variable, and constant names.

6 Library Constants

The OpenSHMEM library provides a set of compile-time constants that may be used to specify options to API routines, provide implementation-specific parameters, or return information about the implementation. All constants that start with *_SHMEM_** are deprecated, but provided for backwards compatibility.

Constant	Description
C/C++: *SHMEM_THREAD_SINGLE*	The OpenSHMEM thread support level which specifies that the program must not be multithreaded. See Section 9.2 for more detail about its use.

Constant	Description
C/C++: *SHMEM_THREAD_FUNNELED*	The OpenSHMEM thread support level which specifies that the program may be multithreaded but must ensure that only the main thread invokes the OpenSHMEM interfaces. See Section 9.2 for more detail about its use.
C/C++: *SHMEM_THREAD_SERIALIZED*	The OpenSHMEM thread support level which specifies that the program may be multithreaded but must ensure that the OpenSHMEM interfaces are not invoked concurrently by multiple threads. See Section 9.2 for more detail about its use.
C/C++: *SHMEM_THREAD_MULTIPLE*	The OpenSHMEM thread support level which specifies that the program may be multithreaded and any thread may invoke the OpenSHMEM interfaces. See Section 9.2 for more detail about its use.
C/C++: *SHMEM_CTX_SERIALIZED*	The context creation option which specifies that the given context is shareable but will not be used by multiple threads concurrently. See Section 9.4.1 for more detail about its use.
C/C++: *SHMEM_CTX_PRIVATE*	The context creation option which specifies that the given context will be used only by the thread that created it. See Section 9.4.1 for more detail about its use.
C/C++: *SHMEM_CTX_NOSTORE*	The context creation option which specifies that quiet and fence operations performed on the given context are not required to enforce completion and ordering of memory store operations. See Section 9.4.1 for more detail about its use.
C/C++/Fortran: *SHMEM_SYNC_VALUE* — deprecation start — *C/C++:* *_SHMEM_SYNC_VALUE* — deprecation end —	The value used to initialize the elements of *pSync* arrays. The value of this constant is implementation specific. See Section 9.8 for more detail about its use.
C/C++/Fortran: *SHMEM_SYNC_SIZE*	Length of a work array that can be used with any SHMEM collective communication operation. Work arrays sized for specific operations may consume less memory. The value of this constant is implementation specific. See Section 9.8 for more detail about its use.
C/C++/Fortran: *SHMEM_BCAST_SYNC_SIZE* — deprecation start — *C/C++:* *_SHMEM_BCAST_SYNC_SIZE* — deprecation end —	Length of the *pSync* arrays needed for broadcast routines. The value of this constant is implementation specific. See Section 9.8.5 for more detail about its use.

Constant	**Description**
C/C++/Fortran: *SHMEM_REDUCE_SYNC_SIZE* — deprecation start — *C/C++*: *_SHMEM_REDUCE_SYNC_SIZE* — deprecation end —	Length of the work arrays needed for reduction routines. The value of this constant is implementation specific. See Section 9.8.7 for more detail about its use.
C/C++/Fortran: *SHMEM_BARRIER_SYNC_SIZE* — deprecation start — *C/C++*: *_SHMEM_BARRIER_SYNC_SIZE* — deprecation end —	Length of the work array needed for barrier routines. The value of this constant is implementation specific. See Section 9.8.2 for more detail about its use.
C/C++/Fortran: *SHMEM_COLLECT_SYNC_SIZE* — deprecation start — *C/C++*: *_SHMEM_COLLECT_SYNC_SIZE* — deprecation end —	Length of the work array needed for collect routines. The value of this constant is implementation specific. See Section 9.8.6 for more detail about its use.
C/C++/Fortran: *SHMEM_ALLTOALL_SYNC_SIZE*	Length of the work array needed for *shmem_alltoall* routines. The value of this constant is implementation specific. See Section 9.8.8 for more detail about its use.
C/C++/Fortran: *SHMEM_ALLTOALLS_SYNC_SIZE*	Length of the work array needed for *shmem_alltoalls* routines. The value of this constant is implementation specific. See Section 9.8.9 for more detail about its use.
C/C++/Fortran: *SHMEM_REDUCE_MIN_WRKDATA_SIZE* — deprecation start — *C/C++*: *_SHMEM_REDUCE_MIN_WRKDATA_SIZE* — deprecation end —	Minimum length of work arrays used in various collective routines.

Constant	Description
C/C++/Fortran: *SHMEM_MAJOR_VERSION* — deprecation start — *C/C++*: *_SHMEM_MAJOR_VERSION* — deprecation end —	Integer representing the major version of OpenSHMEM Specification in use.
C/C++/Fortran: *SHMEM_MINOR_VERSION* — deprecation start — *C/C++*: *_SHMEM_MINOR_VERSION* — deprecation end —	Integer representing the minor version of OpenSHMEM Specification in use.
C/C++/Fortran: *SHMEM_MAX_NAME_LEN* — deprecation start — *C/C++*: *_SHMEM_MAX_NAME_LEN* — deprecation end —	Integer representing the maximum length of *SHMEM_VENDOR_STRING*.
C/C++/Fortran: *SHMEM_VENDOR_STRING* — deprecation start — *C/C++*: *_SHMEM_VENDOR_STRING* — deprecation end —	String representing vendor defined information of size at most *SHMEM_MAX_NAME_LEN*. In *C/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*.
C/C++/Fortran: *SHMEM_CMP_EQ* — deprecation start — *C/C++*: *_SHMEM_CMP_EQ* — deprecation end —	An integer constant expression corresponding to the "equal to" comparison operation. See Section 9.9 for more detail about its use.

Constant	Description
C/C++/Fortran: *SHMEM_CMP_NE* — deprecation start — *C/C++*: *_SHMEM_CMP_NE* — deprecation end —	An integer constant expression corresponding to the "not equal to" comparison operation. See Section 9.9 for more detail about its use.
C/C++/Fortran: *SHMEM_CMP_LT* — deprecation start — *C/C++*: *_SHMEM_CMP_LT* — deprecation end —	An integer constant expression corresponding to the "less than" comparison operation. See Section 9.9 for more detail about its use.
C/C++/Fortran: *SHMEM_CMP_LE* — deprecation start — *C/C++*: *_SHMEM_CMP_LE* — deprecation end —	An integer constant expression corresponding to the "less than or equal to" comparison operation. See Section 9.9 for more detail about its use.
C/C++/Fortran: *SHMEM_CMP_GT* — deprecation start — *C/C++*: *_SHMEM_CMP_GT* — deprecation end —	An integer constant expression corresponding to the "greater than" comparison operation. See Section 9.9 for more detail about its use.
C/C++/Fortran: *SHMEM_CMP_GE* — deprecation start — *C/C++*: *_SHMEM_CMP_GE* — deprecation end —	An integer constant expression corresponding to the "greater than or equal to" comparison operation. See Section 9.9 for more detail about its use.

7 Library Handles

The OpenSHMEM library provides a set of predefined named constant handles. All named constants can be used in initialization expressions or assignments, but not necessarily in array declarations or as labels in *C* switch statements. This implies named constants to be link-time but not necessarily compile-time constants.

Handle	**Description**
C/C++: *SHMEM_CTX_DEFAULT*	Handle of type *shmem_ctx_t* that corresponds to the default communication context. All point-to-point communication operations and synchronizations that do not specify a context are performed on the default context. See Section 9.4 for more detail about its use.

8 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. All environment variables that start with *SMA_** are deprecated, but currently supported for backwards compatibility. If both *SHMEM_*- and *SMA_*-prefixed environment variables are set, then the value in the *SHMEM_*-prefixed environment variable establishes the controlling value. Refer to the *SMA_** Environment Variables deprecation rationale for more details.

Variable	**Value**	**Description**
SHMEM_VERSION	Any	Print the library version at start-up
SHMEM_INFO	Any	Print helpful text about all these environment variables
SHMEM_SYMMETRIC_SIZE	Non-negative integer	Number of bytes to allocate for symmetric heap
SHMEM_DEBUG	Any	Enable debugging messages

9 OpenSHMEM Library API

9.1 Library Setup, Exit, and Query Routines

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

9.1.1 SHMEM_INIT

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

SYNOPSIS

C/C++:

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

— deprecation start —

FORTRAN:

```
CALL SHMEM_INIT()
```

— deprecation end —

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 the first call to *shmem_init*, a subsequent call to *shmem_init* or *shmem_init_thread* in the same program results in undefined behavior.

Return Values

None.

Notes

As of OpenSHMEM 1.2, the use of *start_pes* has been deprecated and calls to it should be replaced with calls to *shmem_init*. While support for *start_pes* is still required in OpenSHMEM libraries, users are encouraged to use *shmem_init*. An important difference between *shmem_init* and *start_pes* is that 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 results in a no-op.

EXAMPLES

The following *shmem_init* example is for *C11* programs:

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

int main(void) {
    static int targ = 0;
```

```
    shmem_init();
    int me = shmem_my_pe();
    int receiver = 1 % shmem_n_pes();

    if (me == 0) {
        int src = 33;
        shmem_put(&targ, &src, 1, receiver);
    }

    shmem_barrier_all(); /* Synchronizes sender and receiver */

    if (me == receiver)
        printf("PE %d targ=%d (expect 33)\n", me, targ);

    shmem_finalize();
    return 0;
}
```

9.1.2 SHMEM_MY_PE

Returns the number of the calling PE.

SYNOPSIS

C/C++:

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

— deprecation start —

FORTRAN:

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

— deprecation end —

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 1.2 the use of *_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.

9.1.3 SHMEM_N_PES

Returns the number of PEs running in a program.

SYNOPSIS

C/C++:

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

— deprecation start —

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

None.

API description

The routine returns the number of PEs running in the program.

Return Values

Integer - Number of PEs running in the OpenSHMEM program.

Notes

As of OpenSHMEM 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_my_pe* and *shmem_n_pes* example is for *C/C++* programs:

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

int main(void)
{
   shmem_init();
   int me = shmem_my_pe();
   int npes = shmem_n_pes();
   printf("I am #%d of %d PEs executing this program\n", me, npes);
   shmem_finalize();
   return 0;
}
```

9.1.4 SHMEM_FINALIZE

A collective operation that releases all 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);
```

— deprecation start —

FORTRAN:

```
CALL SHMEM_FINALIZE()
```

— deprecation end —

DESCRIPTION

Arguments

None.

API description

shmem_finalize is a collective operation that ends the OpenSHMEM portion of a program previously initialized by *shmem_init* or *shmem_init_thread* and releases all 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* to ensure that pending communications are completed and that no resources are released until all PEs have entered *shmem_finalize*. This routine destroys all shareable contexts. The user is responsible for destroying all contexts with the *SHMEM_CTX_PRIVATE* option enabled prior to calling this routine; otherwise, the behavior is undefined. *shmem_finalize* must be the last OpenSHMEM library call encountered in the OpenSHMEM portion of a program. A call to *shmem_finalize* will release all resources initialized by a corresponding call to *shmem_init* or *shmem_init_thread*. All processes that represent the PEs will still exist after the call to *shmem_finalize* returns, but they will no longer have access to 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.

EXAMPLES

The following finalize example is for *C11* programs:

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

int main(void)
{
```

```
   static long x = 10101;
   long y = -1;

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

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

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

   shmem_finalize();
   return 0;
}
```

9.1.5 SHMEM_GLOBAL_EXIT

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

SYNOPSIS

C11:

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

C/C++:

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

— deprecation start —

FORTRAN:

```
INTEGER STATUS
CALL SHMEM_GLOBAL_EXIT(status)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*status*	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., that the program encountered an error or found a 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

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

int main(void)
{
   shmem_init();
   int me = shmem_my_pe();
   if (me == 0) {
      FILE *fp = fopen("input.txt", "r");
      if (fp == NULL) {  /* Input file required by program is not available */
         shmem_global_exit(EXIT_FAILURE);
      }
      /* do something with the file */
      fclose(fp);
  }
  shmem_finalize();
  return 0;
}
```

9.1.6 SHMEM_PE_ACCESSIBLE

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

SYNOPSIS

C/C++:

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

— deprecation start —

FORTRAN:

```fortran
LOGICAL LOG, SHMEM_PE_ACCESSIBLE
INTEGER pe
LOG = SHMEM_PE_ACCESSIBLE(pe)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*pe*	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 a value indicating whether the remote PE is a process running from the same executable file as the local PE, thereby indicating whether full support for symmetric data objects, which may reside in either static memory or the symmetric heap, is available.

Return Values

C/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

This routine may be particularly useful for hybrid programming with other communication libraries (such as MPI) or parallel languages. For example, when an MPI job uses *Multiple Program Multiple Data* (MPMD) mode, multiple executable MPI programs are executed as part of the same MPI job. In such cases, OpenSHMEM support may only be available between processes running from the same executable file. In addition, some environments may allow a hybrid job to span multiple network partitions. In such scenarios, OpenSHMEM support may only be available between PEs within the same partition.

9.1.7 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(const void *addr, int pe);
```

— deprecation start —

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

IN	*addr*	Data object on the local PE.
IN	*pe*	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.

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

This routine may be particularly useful for hybrid programming with other communication libraries (such as MPI) or parallel languages. For example, when an MPI job uses MPMD mode, multiple executable MPI programs may use OpenSHMEM routines. In such cases, 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.

9.1.8 SHMEM_PTR

Returns a local pointer to a symmetric data object on the specified PE.

SYNOPSIS

C/C++:

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

— deprecation start —

FORTRAN:

```
POINTER (PTR, POINTEE)
INTEGER pe
PTR = SHMEM_PTR(dest, pe)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*dest*	The symmetric data object to be referenced.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be accessed. When using *Fortran*, 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.

The *shmem_ptr* routine can provide an efficient means to accomplish communication, for example when a sequence of reads and writes to a data object on a remote PE does not match the access pattern provided in an OpenSHMEM data transfer routine like *shmem_put* or *shmem_iget*.

Return Values

The address of the *dest* data object is returned when it is accessible using memory loads and stores. Otherwise, a null pointer 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 *C11*:

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

int main(void)
{
   static int dest[4];
   shmem_init();
   int me = shmem_my_pe();
   if (me == 0) { /* initialize PE 1's dest array */
      int* ptr = shmem_ptr(dest, 1);
      if (ptr == NULL)
         printf("can't use pointer to directly access PE 1's dest array\n");
      else
         for (int i = 0; i < 4; i++)
            *ptr++ = i + 1;
   }
   shmem_barrier_all();
   if (me == 1)
      printf("PE 1 dest: %d, %d, %d, %d\n",
         dest[0], dest[1], dest[2], dest[3]);
   shmem_finalize();
   return 0;
}
```

9.1.9 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);
```

— deprecation start

FORTRAN:

```
INTEGER MAJOR, MINOR
CALL SHMEM_INFO_GET_VERSION(MAJOR, MINOR)
```

deprecation end —

DESCRIPTION

Arguments

OUT	*major*	The major version of the OpenSHMEM Specification in use.
OUT	*minor*	The minor version of the OpenSHMEM Specification in use.

API description

This routine returns the major and minor version of the OpenSHMEM Specification in use. For a given library implementation, the major and minor version returned by these calls are consistent with the library constants *SHMEM_MAJOR_VERSION* and *SHMEM_MINOR_VERSION*.

Return Values

None.

Notes

None.

9.1.10 SHMEM_INFO_GET_NAME

This routine returns the vendor defined name string that is consistent with the library constant *SHMEM_VENDOR_STRING*.

SYNOPSIS

C/C++:

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

— deprecation start

FORTRAN:

```
CHARACTER *(*)NAME
CALL SHMEM_INFO_GET_NAME(NAME)
```

deprecation end —

DESCRIPTION

Arguments

OUT	*name*	The vendor defined string.

API description

This routine returns the vendor defined name string of size defined by the library constant *SHMEM_MAX_NAME_LEN*. The program calling this function provides the *name* memory buffer of at least size *SHMEM_MAX_NAME_LEN*. The implementation copies the vendor defined string of size at most *SHMEM_MAX_NAME_LEN* to *name*. In *C/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*. If the *name* memory buffer is provided with size less than *SHMEM_MAX_NAME_LEN*, behavior is undefined. For a given library implementation, the vendor string returned is consistent with the library constant *SHMEM_VENDOR_STRING*.

Return Values

None.

Notes

None.

9.1.11 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 routine should function properly and may notify the user about deprecation of its use.

SYNOPSIS

— deprecation start —

C/C++:

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

— deprecation end —

— deprecation start —

FORTRAN:

```
CALL START_PES(npes)
```

— deprecation end —

DESCRIPTION

Arguments

npes	*Unused*	Should be set to *0*.

API description

The *start_pes* routine initializes the OpenSHMEM execution environment. An OpenSHMEM program must call *start_pes*, *shmem_init*, or *shmem_init_thread* before calling any other OpenSHMEM routine. Unlike *shmem_init* and *shmem_init_thread*, *start_pes* does not require a call to *shmem_finalize*. Instead, the OpenSHMEM library is implicitly finalized when the program exits. Implicit finalization is collective and includes a global synchronization to ensure that all pending communication is completed before resources are released.

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.

As of OpenSHMEM 1.2 the use of *start_pes* has been deprecated. Although OpenSHMEM libraries are required to support the call, users are encouraged to use *shmem_init* or *shmem_init_thread* instead.

EXAMPLES

This is a simple program that calls *start_pes*:

```
PROGRAM PUT
INCLUDE "shmem.fh"

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
```

9.2 Thread Support

This section specifies the interaction between the OpenSHMEM interfaces and user threads. It also describes the routines that can be used for initializing and querying the thread environment. There are four levels of threading defined by the OpenSHMEM specification.

SHMEM_THREAD_SINGLE

The OpenSHMEM program must not be multithreaded.

SHMEM_THREAD_FUNNELED

The OpenSHMEM program may be multithreaded. However, the program must ensure that only the main thread invokes the OpenSHMEM interfaces. The main thread is the thread that invokes either *shmem_init* or *shmem_init_thread*.

SHMEM_THREAD_SERIALIZED

The OpenSHMEM program may be multithreaded. However, the program must ensure that the OpenSHMEM interfaces are not invoked concurrently by multiple threads.

SHMEM_THREAD_MULTIPLE

The OpenSHMEM program may be multithreaded and any thread may invoke the OpenSHMEM interfaces.

The following semantics apply to the usage of these models:

1. In the *SHMEM_THREAD_FUNNELED*, *SHMEM_THREAD_SERIALIZED*, and *SHMEM_THREAD_MULTIPLE* thread levels, the *shmem_init* and *shmem_finalize* calls must be invoked by the same thread.
2. Any OpenSHMEM operation initiated by a thread is considered an action of the PE as a whole. The symmetric heap and symmetric variables scope are not impacted by multiple threads invoking the OpenSHMEM interfaces. Each PE has a single symmetric data segment and symmetric heap that is shared by all threads within that PE. For example, a thread invoking a memory allocation routine such as *shmem_malloc* allocates memory that is accessible by all threads of the PE. The requirement that the same symmetric heap operations must be executed by all PEs in the same order also applies in a threaded environment. Similarly, the completion of collective operations is not impacted by multiple threads. For example, *shmem_barrier_all* is completed when all PEs enter and exit the *shmem_barrier_all* call, even though only one thread in the PE is participating in the collective call.
3. Blocking OpenSHMEM calls will only block the calling thread, allowing other threads, if available, to continue executing. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocking call is completed, the thread is ready to continue execution. A blocked thread will not prevent progress of other threads on the same PE and will not prevent them from executing other OpenSHMEM calls when the thread level permits. In addition, a blocked thread will not prevent the progress of OpenSHMEM calls performed on other PEs.
4. In the *SHMEM_THREAD_MULTIPLE* thread level, all OpenSHMEM calls are thread-safe. Any two concurrently running threads may make OpenSHMEM calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
5. In the *SHMEM_THREAD_SERIALIZED* and *SHMEM_THREAD_MULTIPLE* thread levels, if multiple threads call collective routines, including the symmetric heap management routines, it is the programmer's responsibility to ensure the correct ordering of collective calls.

9.2.1 SHMEM_INIT_THREAD

Initializes the OpenSHMEM library, similar to *shmem_init*, and performs any initialization required for supporting the provided thread level.

SYNOPSIS

C/C++:

```
int shmem_init_thread(int requested, int *provided);
```

DESCRIPTION

Arguments

IN	*requested*	The thread level support requested by the user.
OUT	*provided*	The thread level support provided by the OpenSHMEM implementation.

API description

shmem_init_thread initializes the OpenSHMEM library in the same way as *shmem_init*. In addition, *shmem_init_thread* also performs the initialization required for supporting the provided thread level. The argument *requested* is used to specify the desired level of thread support. The argument *provided* returns the support level provided by the library. The allowed values for *provided* and *requested* are *SHMEM_THREAD_SINGLE*, *SHMEM_THREAD_FUNNELED*, *SHMEM_THREAD_SERIALIZED*, and *SHMEM_THREAD_MULTIPLE*.

An OpenSHMEM program is initialized either by *shmem_init* or *shmem_init_thread*. Once an OpenSHMEM library initialization call has been performed, a subsequent initialization call in the same program results in undefined behavior. If the call to *shmem_init_thread* is unsuccessful in allocating and initializing resources for the OpenSHMEM library, then the behavior of any subsequent call to the OpenSHMEM library is undefined.

Return Values

shmem_init_thread returns 0 upon success; otherwise, it returns a non-zero value.

Notes

The OpenSHMEM library can be initialized either by *shmem_init* or *shmem_init_thread*. If the OpenSHMEM library is initialized by *shmem_init*, the library implementation can choose to support any one of the defined thread levels.

9.2.2 SHMEM_QUERY_THREAD

Returns the level of thread support provided by the library.

SYNOPSIS

C/C++:

```
void shmem_query_thread(int *provided);
```

DESCRIPTION

Arguments

OUT	*provided*	The thread level support provided by the OpenSHMEM implementation.

API description

The *shmem_query_thread* call returns the level of thread support currently being provided. The value returned will be same as was returned in *provided* by a call to *shmem_init_thread*, if the OpenSHMEM library was initialized by *shmem_init_thread*. If the library was initialized by *shmem_init*, the implementation can choose to provide any one of the defined thread levels, and *shmem_query_thread* returns this thread level.

Return Values

None.

Notes

None.

9.3 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.

9.3.1 SHMEM_MALLOC, SHMEM_FREE, SHMEM_REALLOC, SHMEM_ALIGN

Collective 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

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

API description

The *shmem_malloc*, *shmem_free*, *shmem_realloc*, and *shmem_align* routines are collective operations that require participation by all PEs.

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 newly allocated portion of the block is uninitialized. 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_align*, *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 includes at least one call to a procedure that is semantically equivalent to *shmem_barrier_all*: *shmem_malloc* and *shmem_align* call a barrier on exit; *shmem_free* calls a barrier on entry; and *shmem_realloc* may call barriers on both entry and exit, depending on whether an existing allocation is modified and whether new memory is allocated. 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 implicit barriers performed by these routines quiet the default context. It is the user's responsibility to ensure that no communication operations involving the given memory block are pending on other contexts prior to calling the *shmem_free* and *shmem_realloc* routines. The user is also responsible for calling these routines with identical argument(s) on all PEs; if differing *ptr*, *size*, or *alignment* arguments are used, the behavior of the call and any subsequent OpenSHMEM calls is 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 OpenSHMEM 1.2 the use of *shmalloc*, *shmemalign*, *shfree*, and *shrealloc* has been deprecated. Although OpenSHMEM libraries are required to support the calls, 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 specify the size of the heap using the *SHMEM_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 routines 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 $O(N)$ memory translation tables, where N is the number of PEs. 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.

9.3.2 SHMEM_CALLOC

Allocate a zeroed block of symmetric memory.

SYNOPSIS

C/C++:
```
void *shmem_calloc(size_t count, size_t size);
```

DESCRIPTION

Arguments

IN	*count*	The number of elements to allocate.
IN	*size*	The size in bytes of each element to allocate.

API description

The *shmem_calloc* routine is a collective operation that allocates a region of remotely-accessible memory for an array of *count* objects of *size* bytes each and returns a pointer to the lowest byte address of the allocated symmetric memory. The space is initialized to all bits zero.

If the allocation succeeds, the pointer returned shall be suitably aligned so that it may be assigned to a pointer to any type of object. If the allocation does not succeed, or either *count* or *size* is *0*, the return value is a null pointer.

The values for *count* and *size* shall each be equal across all PEs calling *shmem_calloc*; otherwise, the behavior is undefined.

The *shmem_calloc* routine calls a procedure that is semantically equivalent to *shmem_barrier_all* on exit.

Return Values

The *shmem_calloc* routine returns a pointer to the lowest byte address of the allocated space; otherwise, it returns a null pointer.

Notes

None.

9.3.3 SHPALLOC

Allocates a block of memory from the symmetric heap.

SYNOPSIS

— deprecation start —

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

OUT	*addr*	First word address of the allocated block.
IN	*length*	Number of words of memory requested. One word is 32 bits.
OUT	*errcode*	Error code is *0* if no error was detected; otherwise, it is a negative integer code for the type of error.
IN	*abort*	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, array *A* can be used 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 *SHMEM_SYMMETRIC_SIZE* environment variable (if available).

Note to implementors

The symmetric heap allocation routines 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.

9.3.4 SHPCLMOVE

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

SYNOPSIS

— deprecation start —

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

INOUT	*addr*	On entry, first word address of the block to change; on exit, the new address of the block if it was moved.
IN	*length*	Requested new total length in words. One word is *32* bits.
OUT	*status*	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.
IN	*abort*	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.

9.3.5 SHPDEALLC

Returns a memory block to the symmetric heap.

SYNOPSIS

— deprecation start —

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

IN	*addr*	First word address of the block to deallocate.
OUT	*errcode*	Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error.
IN	*abort*	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.

9.4 Communication Management Routines

All OpenSHMEM RMA, AMO, and memory ordering routines are performed on a communication context. The communication context defines an independent ordering and completion environment, allowing users to manage the overlap of communication with computation and also to manage communication operations performed by separate threads within a multithreaded PE. For example, in single-threaded environments, contexts may be used to pipeline communication and computation. In multithreaded environments, contexts may additionally provide thread isolation, eliminating overheads resulting from thread interference.

Context handles are of type *shmem_ctx_t* and are valid for language-level assignment and equality comparison. A handle to the desired context is passed as an argument in the *C shmem_ctx_** and type-generic API routines. API routines that do not accept a context argument operate on the default context. The default context can be used explicitly through the *SHMEM_CTX_DEFAULT* handle.

9.4.1 SHMEM_CTX_CREATE

Create a communication context.

SYNOPSIS

C/C++:

```
int shmem_ctx_create(long options, shmem_ctx_t *ctx);
```

DESCRIPTION

Arguments

IN	*options*	The set of options requested for the given context. Multiple options may be requested by combining them with a bitwise OR operation; otherwise, *0* can be given if no options are requested.
OUT	*ctx*	A handle to the newly created context.

API description

The *shmem_ctx_create* routine creates a new communication context and returns its handle through the *ctx* argument. If the context was created successfully, a value of zero is returned; otherwise, a nonzero value is returned. An unsuccessful context creation call is not treated as an error and the OpenSHMEM library remains in a correct state. The creation call can be reattempted with different options or after additional resources become available.

By default, contexts are *shareable* and, when it is allowed by the threading model provided by the OpenSHMEM library, they can be used concurrently by multiple threads within the PE where they were created. The following options can be supplied during context creation to restrict this usage model and enable performance optimizations. When using a given context, the application must comply with the requirements of all options set on that context; otherwise, the behavior is undefined. No options are enabled on the default context.

SHMEM_CTX_SERIALIZED	The given context is shareable; however, it will not be used by multiple threads concurrently. When the *SHMEM_CTX_SERIALIZED* option is set, the user must ensure that operations involving the given context are serialized by the application.
SHMEM_CTX_PRIVATE	The given context will be used only by the thread that created it.

SHMEM_CTX_NOSTORE	Quiet and fence operations performed on the given context are not required to enforce completion and ordering of memory store operations. When ordering of store operations is needed, the application must perform a synchronization operation on a context without the *SHMEM_CTX_NOSTORE* option enabled.

Return Values

Zero on success and nonzero otherwise.

Notes

None.

9.4.2 SHMEM_CTX_DESTROY

Destroy a communication context.

SYNOPSIS

C/C++:

```
void shmem_ctx_destroy(shmem_ctx_t ctx);
```

DESCRIPTION

Arguments

IN	*ctx*	Handle to the context that will be destroyed.

API description

shmem_ctx_destroy destroys a context that was created by a call to *shmem_ctx_create*. It is the user's responsibility to ensure that the context is not used after it has been destroyed, for example when the destroyed context is used by multiple threads. This function performs an implicit quiet operation on the given context before it is freed.

Return Values

None.

Notes

It is invalid to pass *SHMEM_CTX_DEFAULT* to this routine.

Destroying a context makes it impossible for the user to complete communication operations that are pending on that context. This includes nonblocking communication operations, whose local buffers are only returned to the user after the operations have been completed. An implicit quiet is performed when freeing a context to avoid this ambiguity.

A context with the *SHMEM_CTX_PRIVATE* option enabled must be destroyed by the thread that created it.

EXAMPLES

The following example demonstrates the use of contexts in a multithreaded *C11* program that uses OpenMP for threading. This example shows the shared counter load balancing method and illustrates the use of contexts for thread isolation.

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

long pwrk[SHMEM_REDUCE_MIN_WRKDATA_SIZE];
long psync[SHMEM_REDUCE_SYNC_SIZE];

long task_cntr  = 0; /* Next task counter */
long tasks_done = 0; /* Tasks done by this PE */
long total_done = 0; /* Total tasks done by all PEs */

int main(void) {
    int tl, i;
    long ntasks = 1024;  /* Total tasks per PE */

    for (i = 0; i < SHMEM_REDUCE_SYNC_SIZE; i++)
        psync[i] = SHMEM_SYNC_VALUE;

    shmem_init_thread(SHMEM_THREAD_MULTIPLE, &tl);
    if (tl != SHMEM_THREAD_MULTIPLE) shmem_global_exit(1);

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

#pragma omp parallel reduction (+:tasks_done)
    {
        shmem_ctx_t ctx;
        int task_pe = me, pes_done = 0;
        int ret = shmem_ctx_create(SHMEM_CTX_PRIVATE, &ctx);

        if (ret != 0) {
            printf("%d: Error creating context (%d)\n", me, ret);
            shmem_global_exit(2);
        }

        /* Process tasks on all PEs, starting with the local PE.  After
         * all tasks on a PE are completed, help the next PE. */
        while (pes_done < npes) {
            long task = shmem_atomic_fetch_inc(ctx, &task_cntr, task_pe);
            while (task < ntasks) {
                /* Perform task (task_pe, task) */
                tasks_done++;
                task = shmem_atomic_fetch_inc(ctx, &task_cntr, task_pe);
            }
            pes_done++;
            task_pe = (task_pe + 1) % npes;
        }

        shmem_ctx_destroy(ctx);
    }

    shmem_long_sum_to_all(&total_done, &tasks_done, 1, 0, 0, npes, pwrk, psync);

    int result = (total_done != ntasks * npes);
    shmem_finalize();
    return result;
}
```

The following example demonstrates the use of contexts in a single-threaded *C11* program that performs a summation reduction where the data contained in the *in_buf* arrays on all PEs is reduced into the *out_buf* arrays on all PEs. The buffers are divided into segments and processing of the segments is pipelined. Contexts are used to overlap an all-to-all exchange of data for segment *p* with the local reduction of segment *p-1*.

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

#define LEN  8192 /* Full buffer length */
#define PLEN 512  /* Length of each pipeline stage */

int in_buf[LEN], out_buf[LEN];

int main(void) {
    int i, j, *pbuf[2];
    shmem_ctx_t ctx[2];

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

    pbuf[0] = shmem_malloc(PLEN * npes * sizeof(int));
    pbuf[1] = shmem_malloc(PLEN * npes * sizeof(int));

    int ret_0 = shmem_ctx_create(0, &ctx[0]);
    int ret_1 = shmem_ctx_create(0, &ctx[1]);
    if (ret_0 || ret_1) shmem_global_exit(1);

    for (i = 0; i < LEN; i++) {
        in_buf[i] = me; out_buf[i] = 0;
    }

    int p_idx = 0, p = 0; /* Index of ctx and pbuf (p_idx) for current pipeline stage (p) */
    for (i = 1; i <= npes; i++)
        shmem_put_nbi(ctx[p_idx], &pbuf[p_idx][PLEN*me], &in_buf[PLEN*p],
                      PLEN, (me+i) % npes);

    /* Issue communication for pipeline stage p, then accumulate results for stage p-1 */
    for (p = 1; p < LEN/PLEN; p++) {
        p_idx ^= 1;
        for (i = 1; i <= npes; i++)
            shmem_put_nbi(ctx[p_idx], &pbuf[p_idx][PLEN*me], &in_buf[PLEN*p],
                          PLEN, (me+i) % npes);

        shmem_ctx_quiet(ctx[p_idx^1]);
        shmem_sync_all();
        for (i = 0; i < npes; i++)
            for (j = 0; j < PLEN; j++)
                out_buf[PLEN*(p-1)+j] += pbuf[p_idx^1][PLEN*i+j];
    }

    shmem_ctx_quiet(ctx[p_idx]);
    shmem_sync_all();
    for (i = 0; i < npes; i++)
        for (j = 0; j < PLEN; j++)
            out_buf[PLEN*(p-1)+j] += pbuf[p_idx][PLEN*i+j];

    shmem_finalize();
    return 0;
}
```

9.5 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.

Where appropriate compiler support is available, OpenSHMEM provides type-generic one-sided communication interfaces via *C11* generic selection (*C11* §6.5.1.1[5]) for block, scalar, and block-strided put and get communication. Such type-generic routines are supported for the "standard RMA types" listed in Table 3.

The standard RMA types include the exact-width integer types defined in *stdint.h* by *C99*[6] §7.18.1.1 and *C11* §7.20.1.1. When the *C* translation environment does not provide exact-width integer types with *stdint.h*, an OpenSHMEM implemementation is not required to provide support for these types.

TYPE	*TYPENAME*
float	float
double	double
long double	longdouble
char	char
signed char	schar
short	short
int	int
long	long
long long	longlong
unsigned char	uchar
unsigned short	ushort
unsigned int	uint
unsigned long	ulong
unsigned long long	ulonglong
int8_t	int8
int16_t	int16
int32_t	int32
int64_t	int64
uint8_t	uint8
uint16_t	uint16
uint32_t	uint32
uint64_t	uint64
size_t	size
ptrdiff_t	ptrdiff

Table 3: Standard RMA Types and Names

9.5.1 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

C11:

```
void shmem_put(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_put(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t nelems, int pe);
```

[5] Formally, the *C11* specification is ISO/IEC 9899:2011(E).
[6] Formally, the *C99* specification is ISO/IEC 9899:1999(E).

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
void shmem_<TYPENAME>_put(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_ctx_<TYPENAME>_put(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t
    nelems, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

```
void shmem_put<SIZE>(void *dest, const void *source, size_t nelems, int pe);
void shmem_ctx_put<SIZE>(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems, int
    pe);
```

where *SIZE* is one of *8, 16, 32, 64, 128*.

```
void shmem_putmem(void *dest, const void *source, size_t nelems, int pe);
void shmem_ctx_putmem(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems, int
    pe);
```

— deprecation start —

FORTRAN:

```
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)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	Data object to be updated on the remote PE. This data object must be remotely accessible.
IN	*source*	Data object containing the data to be copied.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*. When using *Fortran*, it must be a constant, variable, or array element of default integer type.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer. When using *Fortran*, 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*
shmem_putmem	*Fortran*: Any noncharacter type. *C*: Any data type. nelems is scaled in bytes.
shmem_put4, shmem_put32	Any noncharacter type that has a storage size equal to *32* bits.
shmem_put8	*C*: Any noncharacter type that has a storage size equal to *8* bits. *Fortran*: Any noncharacter type that has a storage size equal to *64* bits.
shmem_put64	Any noncharacter type that has a storage size equal to *64* bits.
shmem_put128	Any noncharacter type that has a storage size equal to *128* bits.
SHMEM_CHARACTER_PUT	Elements of type character. *nelems* is the number of characters to transfer. The actual character lengths of the *source* and *dest* variables are ignored.
SHMEM_COMPLEX_PUT	Elements of type complex of default size.
SHMEM_DOUBLE_PUT	Elements of type double precision.
SHMEM_INTEGER_PUT	Elements of type integer.
SHMEM_LOGICAL_PUT	Elements of type logical.
SHMEM_REAL_PUT	Elements of type real.

Return Values

None.

Notes

When 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))*. As of OpenSHMEM 1.2, 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 *C11* programs:

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

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

9.5.2 SHMEM_P

Copies one data item to a remote PE.

SYNOPSIS

C11:

```
void shmem_p(TYPE *dest, TYPE value, int pe);
void shmem_p(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
void shmem_<TYPENAME>_p(TYPE *dest, TYPE value, int pe);
void shmem_ctx_<TYPENAME>_p(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible array element or scalar data object which will receive the data on the remote PE.
IN	*value*	The value to be transferred to *dest* on the remote PE.
IN	*pe*	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 *shmem_put*, these routines start the remote transfer and may return before the data is delivered to the remote PE. Use *shmem_quiet* to force completion of all remote *Put* transfers.

Return Values

None.

Notes

None.

EXAMPLES

The following example uses *shmem_p* in a *C11* program.

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

int main(void)
{
   const double e = 2.71828182;
   const double epsilon = 0.00000001;
   static double f = 3.1415927;
   shmem_init();
   int me = shmem_my_pe();
   if (me == 0)
      shmem_p(&f, e, 1);
   shmem_barrier_all();
   if (me == 1)
      printf("%s\n", (fabs(f - e) < epsilon) ? "OK" : "FAIL");
```

```
    shmem_finalize();
    return 0;
}
```

9.5.3 SHMEM_IPUT

Copies strided data to a specified PE.

SYNOPSIS

C11:

```
void shmem_iput(TYPE *dest, const TYPE *source, ptrdiff_t dst, ptrdiff_t sst, size_t nelems,
    int pe);
void shmem_iput(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, ptrdiff_t dst, ptrdiff_t
    sst, size_t nelems, int pe);
```

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
void shmem_<TYPENAME>_iput(TYPE *dest, const TYPE *source, ptrdiff_t dst, ptrdiff_t sst,
    size_t nelems, int pe);
void shmem_ctx_<TYPENAME>_iput(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, ptrdiff_t
    dst, ptrdiff_t sst, size_t nelems, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

```
void shmem_iput<SIZE>(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int pe);
void shmem_ctx_iput<SIZE>(shmem_ctx_t ctx, void *dest, const void *source, ptrdiff_t dst,
    ptrdiff_t sst, size_t nelems, int pe);
```

where *SIZE* is one of *8, 16, 32, 64, 128*.

— deprecation start —

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)
CALL SHMEM_IPUT32(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT64(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT128(dest, source, dst, sst, nelems, pe)
CALL SHMEM_LOGICAL_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_REAL_IPUT(dest, source, dst, sst, nelems, pe)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	Array to be updated on the remote PE. This data object must be remotely accessible.
IN	*source*	Array containing the data to be copied.

IN	*dst*	The stride between consecutive elements of the *dest* array. The stride is scaled by the element size of the *dest* array. A value of *1* indicates contiguous data. *dst* must be of type *ptrdiff_t*. When using *Fortran*, it must be a default integer value.
IN	*sst*	The stride between consecutive elements of the *source* array. The stride is scaled by the element size of the *source* array. A value of *1* indicates contiguous data. *sst* must be of type *ptrdiff_t*. When using *Fortran*, it must be a default integer value.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*. When using *Fortran*, it must be a constant, variable, or array element of default integer type.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer. When using *Fortran*, 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*
shmem_iput4, shmem_iput32	Any noncharacter type that has a storage size equal to *32* bits.
shmem_iput8	*C*: Any noncharacter type that has a storage size equal to *8* bits. *Fortran*: Any noncharacter type that has a storage size equal to *64* bits.
shmem_iput64	Any noncharacter type that has a storage size equal to *64* bits.
shmem_iput128	Any noncharacter type that has a storage size equal to *128* bits.
SHMEM_COMPLEX_IPUT	Elements of type complex of default size.
SHMEM_DOUBLE_IPUT	Elements of type double precision.
SHMEM_INTEGER_IPUT	Elements of type integer.
SHMEM_LOGICAL_IPUT	Elements of type logical.
SHMEM_REAL_IPUT	Elements of type real.

Return Values

None.

Notes

When 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 Section 3 for a definition of the term remotely accessible.

EXAMPLES

Consider the following *shmem_iput* example for *C11* 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();
   int me = shmem_my_pe();
   if (me == 0) /* put 5 elements into dest on PE 1 */
      shmem_iput(dest, source, 1, 2, 5, 1);
   shmem_barrier_all(); /* sync sender and receiver */
   if (me == 1) {
      printf("dest on PE %d is %hd %hd %hd %hd %hd\n", me,
         dest[0], dest[1], dest[2], dest[3], dest[4]);
   }
   shmem_finalize();
   return 0;
}
```

9.5.4 SHMEM_GET

Copies data from a specified PE.

SYNOPSIS

C11:

```
void shmem_get(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_get(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t nelems, int pe);
```

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
void shmem_<TYPENAME>_get(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_ctx_<TYPENAME>_get(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t
    nelems, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

```
void shmem_get<SIZE>(void *dest, const void *source, size_t  nelems, int pe);
void shmem_ctx_get<SIZE>(shmem_ctx_t ctx, void *dest, const void *source, size_t  nelems,
    int pe);
```

where *SIZE* is one of *8, 16, 32, 64, 128*.

```
void shmem_getmem(void *dest, const void *source, size_t nelems, int pe);
void shmem_ctx_getmem(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems, int
    pe);
```

— deprecation start ————————————————————

FORTRAN:

```
INTEGER nelems, pe
CALL SHMEM_CHARACTER_GET(dest, source, nelems, pe)
CALL SHMEM_COMPLEX_GET(dest, source, nelems, pe)
CALL SHMEM_DOUBLE_GET(dest, source, nelems, pe)
CALL SHMEM_GET4(dest, source, nelems, pe)
CALL SHMEM_GET8(dest, source, nelems, pe)
CALL SHMEM_GET32(dest, source, nelems, pe)
CALL SHMEM_GET64(dest, source, nelems, pe)
CALL SHMEM_GET128(dest, source, nelems, pe)
CALL SHMEM_GETMEM(dest, source, nelems, pe)
CALL SHMEM_INTEGER_GET(dest, source, nelems, pe)
CALL SHMEM_LOGICAL_GET(dest, source, nelems, pe)
CALL SHMEM_REAL_GET(dest, source, nelems, pe)
```

——————————————————————— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	Local data object to be updated.
IN	*source*	Data object on the PE identified by *pe* that contains the data to be copied. This data object must be remotely accessible.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*. When using *Fortran*, it must be a constant, variable, or array element of default integer type.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer. When using *Fortran*, 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*
shmem_getmem	*Fortran*: Any noncharacter type. *C*: Any data type. nelems is scaled in bytes.
shmem_get4, shmem_get32	Any noncharacter type that has a storage size equal to *32* bits.
shmem_get8	*C*: Any noncharacter type that has a storage size equal to *8* bits. *Fortran*: Any noncharacter type that has a storage size equal to *64* bits.
shmem_get64	Any noncharacter type that has a storage size equal to *64* bits.
shmem_get128	Any noncharacter type that has a storage size equal to *128* bits.
SHMEM_CHARACTER_GET	Elements of type character. *nelems* is the number of characters to transfer. The actual character lengths of the *source* and *dest* variables are ignored.
SHMEM_COMPLEX_GET	Elements of type complex of default size.
SHMEM_DOUBLE_GET	*Fortran*: Elements of type double precision.
SHMEM_INTEGER_GET	Elements of type integer.
SHMEM_LOGICAL_GET	Elements of type logical.
SHMEM_REAL_GET	Elements of type real.

Return Values

None.

Notes

See Section 3 for a definition of the term remotely accessible. When 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
INCLUDE "shmem.fh"

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
```

9.5.5 SHMEM_G

Copies one data item from a remote PE

SYNOPSIS

C11:

```
TYPE shmem_g(const TYPE *source, int pe);
TYPE shmem_g(shmem_ctx_t ctx, const TYPE *source, int pe);
```

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
TYPE shmem_<TYPENAME>_g(const TYPE *source, int pe);
TYPE shmem_ctx_<TYPENAME>_g(shmem_ctx_t ctx, const TYPE *source, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
IN	*source*	The remotely accessible array element or scalar data object.
IN	*pe*	The number of the remote PE on which *source* 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_g* example is for *C11* programs:

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

int main(void)
{
   long y = -1;
   static long x = 10101;
   shmem_init();
   int me = shmem_my_pe();
   int npes = shmem_n_pes();
   if (me == 0)
      y = shmem_g(&x, npes-1);
   printf("%d: y = %ld\n", me, y);
   shmem_finalize();
   return 0;
}
```

9.5.6 SHMEM_IGET

Copies strided data from a specified PE.

SYNOPSIS

C11:

```
void shmem_iget(TYPE *dest, const TYPE *source, ptrdiff_t dst, ptrdiff_t sst, size_t nelems,
    int pe);
void shmem_iget(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, ptrdiff_t dst, ptrdiff_t
    sst, size_t nelems, int pe);
```

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
void shmem_<TYPENAME>_iget(TYPE *dest, const TYPE *source, ptrdiff_t dst, ptrdiff_t sst,
    size_t nelems, int pe);
void shmem_ctx_<TYPENAME>_iget(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, ptrdiff_t
    dst, ptrdiff_t sst, size_t nelems, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

```
void shmem_iget<SIZE>(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int pe);
void shmem_ctx_iget<SIZE>(shmem_ctx_t ctx, void *dest, const void *source, ptrdiff_t dst,
    ptrdiff_t sst, size_t nelems, int pe);
```

where *SIZE* is one of *8, 16, 32, 64, 128*.

— deprecation start —

FORTRAN:

```
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)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	Array to be updated on the local PE.
IN	*source*	Array containing the data to be copied on the remote PE.
IN	*dst*	The stride between consecutive elements of the *dest* array. The stride is scaled by the element size of the *dest* array. A value of *1* indicates contiguous data. *dst* must be of type *ptrdiff_t*. When using *Fortran*, it must be a default integer value.
IN	*sst*	The stride between consecutive elements of the *source* array. The stride is scaled by the element size of the *source* array. A value of *1* indicates contiguous data. *sst* must be of type *ptrdiff_t*. When using *Fortran*, it must be a default integer value.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*. When using *Fortran*, it must be a constant, variable, or array element of default integer type.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer. When using *Fortran*, 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_iget4, shmem_iget32	Any noncharacter type that has a storage size equal to *32* bits.
shmem_iget8	*C*: Any noncharacter type that has a storage size equal to *8* bits. *Fortran*: Any noncharacter type that has a storage size equal to *64* bits.
shmem_iget64	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_COMPLEX_IGET	Elements of type complex of default size.
SHMEM_DOUBLE_IGET	*Fortran*: 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

When 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
INCLUDE "shmem.fh"

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
```

9.6 Non-blocking Remote Memory Access Routines

9.6.1 SHMEM_PUT_NBI

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

SYNOPSIS

C11:

```
void shmem_put_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_put_nbi(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t nelems, int pe);
```

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
void shmem_<TYPENAME>_put_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_ctx_<TYPENAME>_put_nbi(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t
    nelems, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

```
void shmem_put<SIZE>_nbi(void *dest, const void *source, size_t nelems, int pe);
void shmem_ctx_put<SIZE>_nbi(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems,
    int pe);
```

where *SIZE* is one of *8, 16, 32, 64, 128*.

```
void shmem_putmem_nbi(void *dest, const void *source, size_t nelems, int pe);
void shmem_ctx_putmem_nbi(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems,
    int pe);
```

— deprecation start —

FORTRAN:

```
CALL SHMEM_CHARACTER_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_COMPLEX_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_DOUBLE_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_INTEGER_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_LOGICAL_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT4_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT8_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT32_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT64_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT128_NBI(dest, source, nelems, pe)
CALL SHMEM_PUTMEM_NBI(dest, source, nelems, pe)
CALL SHMEM_REAL_PUT_NBI(dest, source, nelems, pe)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	Data object to be updated on the remote PE. This data object must be remotely accessible.
IN	*source*	Data object containing the data to be copied.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*. When using *Fortran*, it must be a constant, variable, or array element of default integer type.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer. When using *Fortran*, it must be a constant, variable, or array element of default integer type.

API description

The routines return after posting the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, the data has been copied into the *dest* array on the destination 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*
shmem_putmem_nbi	*Fortran*: Any noncharacter type. *C*: Any data type. nelems is scaled in bytes.
shmem_put4_nbi, shmem_put32_nbi	Any noncharacter type that has a storage size equal to *32* bits.
shmem_put8_nbi	*C*: Any noncharacter type that has a storage size equal to *8* bits. *Fortran*: Any noncharacter type that has a storage size equal to *64* bits.
shmem_put64_nbi	Any noncharacter type that has a storage size equal to *64* bits.

shmem_put128_nbi	Any noncharacter type that has a storage size equal to *128* bits.
SHMEM_CHARACTER_PUT_NBI	Elements of type character. *nelems* is the number of characters to transfer. The actual character lengths of the *source* and *dest* variables are ignored.
SHMEM_COMPLEX_PUT_NBI	Elements of type complex of default size.
SHMEM_DOUBLE_PUT_NBI	Elements of type double precision.
SHMEM_INTEGER_PUT_NBI	Elements of type integer.
SHMEM_LOGICAL_PUT_NBI	Elements of type logical.
SHMEM_REAL_PUT_NBI	Elements of type real.

Return Values

None.

Notes

None.

9.6.2 SHMEM_GET_NBI

The nonblocking get routines provide a method for copying data from a contiguous remote data object on the specified PE to the local data object.

SYNOPSIS

C11:

```
void shmem_get_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_get_nbi(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t nelems, int pe);
```

where *TYPE* is one of the standard RMA types specified by Table 3.

C/C++:

```
void shmem_<TYPENAME>_get_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_ctx_<TYPENAME>_get_nbi(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t
    nelems, int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 3.

```
void shmem_get<SIZE>_nbi(void *dest, const void *source, size_t  nelems, int pe);
void shmem_ctx_get<SIZE>_nbi(shmem_ctx_t ctx, void *dest, const  void *source, size_t
    nelems, int pe);
```

where *SIZE* is one of *8, 16, 32, 64, 128*.

```
void shmem_getmem_nbi(void *dest, const void *source, size_t nelems, int pe);
void shmem_ctx_getmem_nbi(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems,
    int pe);
```

— deprecation start

FORTRAN:

```
INTEGER nelems, pe
CALL SHMEM_CHARACTER_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_COMPLEX_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_DOUBLE_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_GET4_NBI(dest, source, nelems, pe)
CALL SHMEM_GET8_NBI(dest, source, nelems, pe)
CALL SHMEM_GET32_NBI(dest, source, nelems, pe)
CALL SHMEM_GET64_NBI(dest, source, nelems, pe)
CALL SHMEM_GET128_NBI(dest, source, nelems, pe)
```

```
CALL SHMEM_GETMEM_NBI(dest, source, nelems, pe)
CALL SHMEM_INTEGER_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_LOGICAL_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_REAL_GET_NBI(dest, source, nelems, pe)
```

——— deprecation end ———

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	Local data object to be updated.
IN	*source*	Data object on the PE identified by *pe* that contains the data to be copied. This data object must be remotely accessible.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*. When using *Fortran*, it must be a constant, variable, or array element of default integer type.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer. When using *Fortran*, 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 posting the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, 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*
shmem_getmem_nbi	*Fortran*: Any noncharacter type. *C*: Any data type. nelems is scaled in bytes.
shmem_get4_nbi, shmem_get32_nbi	Any noncharacter type that has a storage size equal to *32* bits.
shmem_get8_nbi	*C*: Any noncharacter type that has a storage size equal to *8* bits. *Fortran*: Any noncharacter type that has a storage size equal to *64* bits.
shmem_get64_nbi	Any noncharacter type that has a storage size equal to *64* bits.
shmem_get128_nbi	Any noncharacter type that has a storage size equal to *128* bits.
SHMEM_CHARACTER_GET_NBI	Elements of type character. *nelems* is the number of characters to transfer. The actual character lengths of the *source* and *dest* variables are ignored.
SHMEM_COMPLEX_GET_NBI	Elements of type complex of default size.
SHMEM_DOUBLE_GET_NBI	*Fortran*: Elements of type double precision.
SHMEM_INTEGER_GET_NBI	Elements of type integer.
SHMEM_LOGICAL_GET_NBI	Elements of type logical.
SHMEM_REAL_GET_NBI	Elements of type real.

Return Values

None.

Notes

See Section 3 for a definition of the term remotely accessible. When 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))*.

9.7 Atomic Memory Operations

An *Atomic Memory Operation* (AMO) is a one-sided communication mechanism that combines memory read, update, or write operations with atomicity guarantees described in Section 3.1. Similar to the RMA routines, described in Section 9.5, the AMOs are performed only on symmetric objects. OpenSHMEM defines two types of AMO routines:

- The *fetching* routines return the original value of, and optionally update, the remote data object in a single atomic operation. The routines return after the data has been fetched from the target PE and delivered to the calling PE. The data type of the returned value is the same as the type of the remote data object.

  The fetching routines include: *shmem_atomic_{fetch, compare_swap, swap}* and *shmem_atomic_fetch_{inc, add, and, or, xor}*.

- The *non-fetching* routines update the remote data object in a single atomic operation. A call to a non-fetching atomic routine issues the atomic operation and may return before the operation executes on the target PE. The *shmem_quiet*, *shmem_barrier*, or *shmem_barrier_all* routines can be used to force completion for these non-fetching atomic routines.

  The non-fetching routines include: *shmem_atomic_{set, inc, add, and, or, xor}*.

Where appropriate compiler support is available, OpenSHMEM provides type-generic AMO interfaces via *C11* generic selection. The type-generic support for the AMO routines is as follows:

- *shmem_atomic_{compare_swap, fetch_inc, inc, fetch_add, add}* support the "standard AMO types" listed in Table 4,
- *shmem_atomic_{fetch, set, swap}* support the "extended AMO types" listed in Table 5, and
- *shmem_atomic_{fetch_and, and, fetch_or, or, fetch_xor, xor}* support the "bitwise AMO types" listed in Table 6.

The standard, extended, and bitwise AMO types include some of the exact-width integer types defined in *stdint.h* by *C99* §7.18.1.1 and *C11* §7.20.1.1. When the *C* translation environment does not provide exact-width integer types with *stdint.h*, an OpenSHMEM implemementation is not required to provide support for these types.

9.7.1 SHMEM_ATOMIC_FETCH

Atomically fetches the value of a remote data object.

SYNOPSIS

C11:

```
TYPE shmem_atomic_fetch(const TYPE *source, int pe);
TYPE shmem_atomic_fetch(shmem_ctx_t ctx, const TYPE *source, int pe);
```

where *TYPE* is one of the extended AMO types specified by Table 5.

C/C++:

```
TYPE shmem_<TYPENAME>_atomic_fetch(const TYPE *source, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_fetch(shmem_ctx_t ctx, const TYPE *source, int pe);
```

TYPE	*TYPENAME*
int	int
long	long
long long	longlong
unsigned int	uint
unsigned long	ulong
unsigned long long	ulonglong
int32_t	int32
int64_t	int64
uint32_t	uint32
uint64_t	uint64
size_t	size
ptrdiff_t	ptrdiff

Table 4: Standard AMO Types and Names

TYPE	*TYPENAME*
float	float
double	double
int	int
long	long
long long	longlong
unsigned int	uint
unsigned long	ulong
unsigned long long	ulonglong
int32_t	int32
int64_t	int64
uint32_t	uint32
uint64_t	uint64
size_t	size
ptrdiff_t	ptrdiff

Table 5: Extended AMO Types and Names

where *TYPE* is one of the extended AMO types and has a corresponding *TYPENAME* specified by Table 5.

— deprecation start —

C11:

```
TYPE shmem_fetch(const TYPE *source, int pe);
```

where *TYPE* is one of {*float*, *double*, *int*, *long*, *long long*}.

C/C++:

```
TYPE shmem_<TYPENAME>_fetch(const TYPE *source, int pe);
```

where *TYPE* is one of {*float*, *double*, *int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 5.

— deprecation end —

— deprecation start —

FORTRAN:

```
INTEGER pe
INTEGER*4 SHMEM_INT4_FETCH, ires_i4
ires\_i4 = SHMEM_INT4_FETCH(source, pe)
```

TYPE	*TYPENAME*
unsigned int	uint
unsigned long	ulong
unsigned long long	ulonglong
int32_t	int32
int64_t	int64
uint32_t	uint32
uint64_t	uint64

Table 6: Bitwise AMO Types and Names

```
INTEGER*8 SHMEM_INT8_FETCH, ires_i8
ires\_i8 = SHMEM_INT8_FETCH(source, pe)
REAL*4 SHMEM_REAL4_FETCH, res_r4
res\_r4 = SHMEM_REAL4_FETCH(source, pe)
REAL*8 SHMEM_REAL8_FETCH, res_r8
res\_r8 = SHMEM_REAL8_FETCH(source, pe)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
IN	*source*	The remotely accessible data object to be fetched from the remote PE.
IN	*pe*	An integer that indicates the PE number from which *source* is to be fetched.

API description

shmem_atomic_fetch performs an atomic fetch operation. It returns the contents of the *source* as an atomic operation.

Return Values

The contents at the *source* address on the remote PE. The data type of the return value is the same as the type of the remote data object.

Notes

None.

9.7.2 SHMEM_ATOMIC_SET

Atomically sets the value of a remote data object.

SYNOPSIS

C11:

```
void shmem_atomic_set(TYPE *dest, TYPE value, int pe);
void shmem_atomic_set(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the extended AMO types specified by Table 5.

C/C++:

```
void shmem_<TYPENAME>_atomic_set(TYPE *dest, TYPE value, int pe);
void shmem_ctx_<TYPENAME>_atomic_set(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the extended AMO types and has a corresponding *TYPENAME* specified by Table 5.

— deprecation start —

C11:

```
void shmem_set(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*float*, *double*, *int*, *long*, *long long*}.

C/C++:

```
void shmem_<TYPENAME>_set(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*float*, *double*, *int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 5.

— deprecation end —

— deprecation start —

FORTRAN:

```
INTEGER pe
INTEGER*4 SHMEM_INT4_SET, value_i4
CALL SHMEM_INT4_SET(dest, value_i4, pe)
INTEGER*8 SHMEM_INT8_SET, value_i8
CALL SHMEM_INT8_SET(dest, value_i8, pe)
REAL*4 SHMEM_REAL4_SET, value_r4
CALL SHMEM_REAL4_SET(dest, value_r4, pe)
REAL*8 SHMEM_REAL8_SET, value_r8
CALL SHMEM_REAL8_SET(dest, value_r8, pe)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible data object to be set on the remote PE.
IN	*value*	The value to be atomically written to the remote PE.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be updated.

API description

shmem_atomic_set performs an atomic set operation. It writes the *value* into *dest* on *pe* as an atomic operation.

Return Values

None.

Notes

None.

9.7.3 SHMEM_ATOMIC_COMPARE_SWAP

Performs an atomic conditional swap on a remote data object.

SYNOPSIS

C11:

```
TYPE shmem_atomic_compare_swap(TYPE *dest, TYPE cond, TYPE value, int pe);
TYPE shmem_atomic_compare_swap(shmem_ctx_t ctx, TYPE *dest, TYPE cond, TYPE value, int pe);
```

where *TYPE* is one of the standard AMO types specified by Table 4.

C/C++:

```
TYPE shmem_<TYPENAME>_atomic_compare_swap(TYPE *dest, TYPE cond, TYPE value, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_compare_swap(shmem_ctx_t ctx, TYPE *dest, TYPE cond, TYPE
    value, int pe);
```

where *TYPE* is one of the standard AMO types and has a corresponding *TYPENAME* specified by Table 4.

— deprecation start —

C11:

```
TYPE shmem_cswap(TYPE *dest, TYPE cond, TYPE value, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*}.

C/C++:

```
TYPE shmem_<TYPENAME>_cswap(TYPE *dest, TYPE cond, TYPE value, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 4.

— deprecation end —

— deprecation start —

FORTRAN:

```
INTEGER pe
INTEGER*4 SHMEM_INT4_CSWAP,  cond_i4, value_i4, ires_i4
ires\_i4 = SHMEM_INT4_CSWAP(dest, cond_i4, value_i4, pe)
INTEGER*8 SHMEM_INT8_CSWAP,  cond_i8, value_i8, ires_i8
ires\_i8 = SHMEM_INT8_CSWAP(dest, cond_i8, value_i8, pe)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible integer data object to be updated on the remote PE.
IN	*cond*	*cond* is compared to the remote *dest* value. If *cond* and the remote *dest* are equal, then *value* is swapped into the remote *dest*; otherwise, the remote *dest* is unchanged. In either case, the old value of the remote *dest* is returned as the routine return value. *cond* must be of the same data type as *dest*.
IN	*value*	The value to be atomically written to the remote PE. *value* must be the same data type as *dest*.

IN	*pe*	An integer that indicates the PE number upon which *dest* is to be updated. When using *Fortran*, 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.

When using *Fortran*, *dest*, *cond*, and *value* must be of the following type:

Routine	Data type of *dest*, *cond*, and *value*
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;
   shmem_init();
   int me = shmem_my_pe();
   int oldval = shmem_atomic_compare_swap(&race_winner, -1, me, 0);
   if (oldval == -1) printf("PE %d was first\n", me);
   shmem_finalize();
   return 0;
}
```

9.7.4 SHMEM_ATOMIC_SWAP

Performs an atomic swap to a remote data object.

SYNOPSIS

C11:

```
TYPE shmem_atomic_swap(TYPE *dest, TYPE value, int pe);
TYPE shmem_atomic_swap(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the extended AMO types specified by Table 5.

C/C++:

```
TYPE shmem_<TYPENAME>_atomic_swap(TYPE *dest, TYPE value, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_swap(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the extended AMO types and has a corresponding *TYPENAME* specified by Table 5.

— deprecation start —

C11:

```
TYPE shmem_swap(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*float*, *double*, *int*, *long*, *long long*}.

C/C++:

```
TYPE shmem_<TYPENAME>_swap(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*float*, *double*, *int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 5.

— deprecation end —

— deprecation start —

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)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible integer data object to be updated on the remote PE. When using *C/C++*, the type of *dest* should match that implied in the SYNOPSIS section.
IN	*value*	The value to be atomically written to the remote PE. *value* is the same type as *dest*.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be updated. When using *Fortran*, it must be a default integer value.

API description

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

When using *Fortran*, *dest* and *value* must be of the following type:

Routine	Data type of *dest* and *value*

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 swaps 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)
{
   static long dest;
   shmem_init();
   int me = shmem_my_pe();
   int npes = shmem_n_pes();
   dest = me;
   shmem_barrier_all();
   long new_val = me;
   if (me & 1) {
      long swapped_val = shmem_atomic_swap(&dest, new_val, (me + 1) % npes);
      printf("%d: dest = %ld, swapped = %ld\n", me, dest, swapped_val);
   }
   shmem_finalize();
   return 0;
}
```

9.7.5 SHMEM_ATOMIC_FETCH_INC

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

SYNOPSIS

C11:

```
TYPE shmem_atomic_fetch_inc(TYPE *dest, int pe);
TYPE shmem_atomic_fetch_inc(shmem_ctx_t ctx, TYPE *dest, int pe);
```

where *TYPE* is one of the standard AMO types specified by Table 4.

C/C++:

```
TYPE shmem_<TYPENAME>_atomic_fetch_inc(TYPE *dest, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_fetch_inc(shmem_ctx_t ctx, TYPE *dest, int pe);
```

where *TYPE* is one of the standard AMO types and has a corresponding *TYPENAME* specified by Table 4.

— deprecation start —

C11:

```
TYPE shmem_finc(TYPE *dest, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*}.

C/C++:

```
TYPE shmem_<TYPENAME>_finc(TYPE *dest, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 4.

— deprecation end —

— deprecation start —

FORTRAN:

```
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)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible integer data object to be updated on the remote PE. The type of *dest* should match that implied in the SYNOPSIS section.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be updated. When using *Fortran*, 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.

When using *Fortran*, *dest* must be of the following type:

Routine	Data type of *dest*
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_atomic_fetch_inc* example is for *C11* programs:

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

int main(void)
{
   int old = -1;
   static int dst = 22;
   shmem_init();
   int me = shmem_my_pe();
   if (me == 0)
      old = shmem_atomic_fetch_inc(&dst, 1);
   shmem_barrier_all();
   printf("%d: old = %d, dst = %d\n", me, old, dst);
   shmem_finalize();
   return 0;
}
```

9.7.6 SHMEM_ATOMIC_INC

Performs an atomic increment operation on a remote data object.

SYNOPSIS

C11:

```
void shmem_atomic_inc(TYPE *dest, int pe);
void shmem_atomic_inc(shmem_ctx_t ctx, TYPE *dest, int pe);
```

where *TYPE* is one of the standard AMO types specified by Table 4.

C/C++:

```
void shmem_<TYPENAME>_atomic_inc(TYPE *dest, int pe);
void shmem_ctx_<TYPENAME>_atomic_inc(shmem_ctx_t ctx, TYPE *dest, int pe);
```

where *TYPE* is one of the standard AMO types and has a corresponding *TYPENAME* specified by Table 4.

— deprecation start —

C11:

```
void shmem_inc(TYPE *dest, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*}.

C/C++:

```
void shmem_<TYPENAME>_inc(TYPE *dest, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 4.

— deprecation end —

— deprecation start —

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible integer data object to be updated on the remote PE. The type of *dest* should match that implied in the SYNOPSIS section.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be updated. When using *Fortran*, it must be a default integer value.

API description

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

When using *Fortran*, *dest* must be of the following type:

Routine	Data type of *dest*
SHMEM_INT4_INC	*4*-byte integer
SHMEM_INT8_INC	*8*-byte integer

Return Values

None.

Notes

None.

EXAMPLES

The following *shmem_atomic_inc* example is for *C11* programs:

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

int main(void)
{
   static int dst = 74;
   shmem_init();
   int me = shmem_my_pe();
   if (me == 0)
      shmem_atomic_inc(&dst, 1);
   shmem_barrier_all();
   printf("%d: dst = %d\n", me, dst);
   shmem_finalize();
   return 0;
}
```

9.7.7 SHMEM_ATOMIC_FETCH_ADD

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

SYNOPSIS

C11:

```
TYPE shmem_atomic_fetch_add(TYPE *dest, TYPE value, int pe);
TYPE shmem_atomic_fetch_add(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the standard AMO types specified by Table 4.

C/C++:

```
TYPE shmem_<TYPENAME>_atomic_fetch_add(TYPE *dest, TYPE value, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_fetch_add(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the standard AMO types and has a corresponding *TYPENAME* specified by Table 4.

— deprecation start —

C11:

```
TYPE shmem_fadd(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*}.

C/C++:

```
TYPE shmem_<TYPENAME>_fadd(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 4.

— deprecation end —

— deprecation start —

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)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible integer data object to be updated on the remote PE. The type of *dest* should match that implied in the SYNOPSIS section.
IN	*value*	The value to be atomically added to *dest*. The type of *value* should match that implied in the SYNOPSIS section.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be updated. When using *Fortran*, it must be a default integer value.

API description

shmem_atomic_fetch_add 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.

When using *Fortran*, *dest* and *value* must be of the following type:

Routine	Data type of *dest* and *value*
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_atomic_fetch_add* example is for *C11* programs:

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

int main(void)
{
   int old = -1;
   static int dst = 22;
   shmem_init();
   int me = shmem_my_pe();
   if (me == 1)
      old = shmem_atomic_fetch_add(&dst, 44, 0);
   shmem_barrier_all();
   printf("%d: old = %d, dst = %d\n", me, old, dst);
   shmem_finalize();
   return 0;
}
```

9.7.8 SHMEM_ATOMIC_ADD

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

SYNOPSIS

C11:

```
void shmem_atomic_add(TYPE *dest, TYPE value, int pe);
void shmem_atomic_add(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the standard AMO types specified by Table 4.

C/C++:

```
void shmem_<TYPENAME>_atomic_add(TYPE *dest, TYPE value, int pe);
void shmem_ctx_<TYPENAME>_atomic_add(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the standard AMO types and has a corresponding *TYPENAME* specified by Table 4.

— deprecation start —

C11:

```
void shmem_add(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*}.

C/C++:

```
void shmem_<TYPENAME>_add(TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of {*int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 4.

— deprecation end —

— deprecation start —

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)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	The remotely accessible integer data object to be updated on the remote PE. When using *C/C++*, the type of *dest* should match that implied in the SYNOPSIS section.
IN	*value*	The value to be atomically added to *dest*. When using *C/C++*, the type of *value* should match that implied in the SYNOPSIS section. When using *Fortran*, it must be of type integer with an element size of *dest*.
IN	*pe*	An integer that indicates the PE number upon which *dest* is to be updated. When using *Fortran*, it must be a default integer value.

API description

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

When using *Fortran*, *dest* and *value* must be of the following type:

Routine	Data type of *dest* and *value*
SHMEM_INT4_ADD	*4*-byte integer
SHMEM_INT8_ADD	*8*-byte integer

Return Values

None.

Notes

None.

EXAMPLES

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

int main(void)
{
   static int dst = 22;
   shmem_init();
   int me = shmem_my_pe();
   if (me == 1)
      shmem_atomic_add(&dst, 44, 0);
   shmem_barrier_all();
   printf("%d: dst = %d\n", me, dst);
   shmem_finalize();
   return 0;
}
```

9.7.9 SHMEM_ATOMIC_FETCH_AND

Atomically perform a fetching bitwise AND operation on a remote data object.

SYNOPSIS

C11:

```c
TYPE shmem_atomic_fetch_and(TYPE *dest, TYPE value, int pe);
TYPE shmem_atomic_fetch_and(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types specified by Table 6.

C/C++:

```c
TYPE shmem_<TYPENAME>_atomic_fetch_and(TYPE *dest, TYPE value, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_fetch_and(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 6.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	A pointer to the remotely accessible data object to be updated.
IN	*value*	The operand to the bitwise AND operation.
IN	*pe*	An integer value for the PE on which *dest* is to be updated.

API description

shmem_atomic_fetch_and atomically performs a fetching bitwise AND on the remotely accessible data object pointed to by *dest* at PE *pe* with the operand *value*.

Return Values

The value pointed to by *dest* on PE *pe* immediately before the operation is performed.

Notes

None.

9.7.10 SHMEM_ATOMIC_AND

Atomically perform a non-fetching bitwise AND operation on a remote data object.

SYNOPSIS

C11:

```
void shmem_atomic_and(TYPE *dest, TYPE value, int pe);
void shmem_atomic_and(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types specified by Table 6.

C/C++:

```
void shmem_<TYPENAME>_atomic_and(TYPE *dest, TYPE value, int pe);
void shmem_ctx_<TYPENAME>_atomic_and(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 6.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	A pointer to the remotely accessible data object to be updated.
IN	*value*	The operand to the bitwise AND operation.
IN	*pe*	An integer value for the PE on which *dest* is to be updated.

API description

shmem_atomic_and atomically performs a non-fetching bitwise AND on the remotely accessible data object pointed to by *dest* at PE *pe* with the operand *value*.

Return Values

None.

Notes

None.

9.7.11 SHMEM_ATOMIC_FETCH_OR

Atomically perform a fetching bitwise OR operation on a remote data object.

SYNOPSIS

C11:

```
TYPE shmem_atomic_fetch_or(TYPE *dest, TYPE value, int pe);
TYPE shmem_atomic_fetch_or(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types specified by Table 6.

C/C++:

```
TYPE shmem_<TYPENAME>_atomic_fetch_or(TYPE *dest, TYPE value, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_fetch_or(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 6.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	A pointer to the remotely accessible data object to be updated.
IN	*value*	The operand to the bitwise OR operation.
IN	*pe*	An integer value for the PE on which *dest* is to be updated.

API description

shmem_atomic_fetch_or atomically performs a fetching bitwise OR on the remotely accessible data object pointed to by *dest* at PE *pe* with the operand *value*.

Return Values

The value pointed to by *dest* on PE *pe* immediately before the operation is performed.

Notes

None.

9.7.12 SHMEM_ATOMIC_OR

Atomically perform a non-fetching bitwise OR operation on a remote data object.

SYNOPSIS

C11:

```
void shmem_atomic_or(TYPE *dest, TYPE value, int pe);
void shmem_atomic_or(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types specified by Table 6.

C/C++:

```
void shmem_<TYPENAME>_atomic_or(TYPE *dest, TYPE value, int pe);
void shmem_ctx_<TYPENAME>_atomic_or(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 6.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	A pointer to the remotely accessible data object to be updated.

IN	*value*	The operand to the bitwise OR operation.
IN	*pe*	An integer value for the PE on which *dest* is to be updated.

API description

shmem_atomic_or atomically performs a non-fetching bitwise OR on the remotely accessible data object pointed to by *dest* at PE *pe* with the operand *value*.

Return Values

None.

Notes

None.

9.7.13 SHMEM_ATOMIC_FETCH_XOR

Atomically perform a fetching bitwise exclusive OR (XOR) operation on a remote data object.

SYNOPSIS

C11:

```
TYPE shmem_atomic_fetch_xor(TYPE *dest, TYPE value, int pe);
TYPE shmem_atomic_fetch_xor(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types specified by Table 6.

C/C++:

```
TYPE shmem_<TYPENAME>_atomic_fetch_xor(TYPE *dest, TYPE value, int pe);
TYPE shmem_ctx_<TYPENAME>_atomic_fetch_xor(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 6.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	A pointer to the remotely accessible data object to be updated.
IN	*value*	The operand to the bitwise XOR operation.
IN	*pe*	An integer value for the PE on which *dest* is to be updated.

API description

shmem_atomic_fetch_xor atomically performs a fetching bitwise XOR on the remotely accessible data object pointed to by *dest* at PE *pe* with the operand *value*.

Return Values

The value pointed to by *dest* on PE *pe* immediately before the operation is performed.

Notes

None.

9.7.14 SHMEM_ATOMIC_XOR

Atomically perform a non-fetching bitwise exclusive OR (XOR) operation on a remote data object.

SYNOPSIS

C11:

```
void shmem_atomic_xor(TYPE *dest, TYPE value, int pe);
void shmem_atomic_xor(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types specified by Table 6.

C/C++:

```
void shmem_<TYPENAME>_atomic_xor(TYPE *dest, TYPE value, int pe);
void shmem_ctx_<TYPENAME>_atomic_xor(shmem_ctx_t ctx, TYPE *dest, TYPE value, int pe);
```

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 6.

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.
OUT	*dest*	A pointer to the remotely accessible data object to be updated.
IN	*value*	The operand to the bitwise XOR operation.
IN	*pe*	An integer value for the PE on which *dest* is to be updated.

API description

shmem_atomic_xor atomically performs a non-fetching bitwise XOR on the remotely accessible data object pointed to by *dest* at PE *pe* with the operand *value*.

Return Values

None.

Notes

None.

9.8 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 in the active set calling the collective routine results in undefined behavior. All collective routines have an active set as an input parameter except *shmem_barrier_all* and *shmem_sync_all*. Both *shmem_barrier_all* and *shmem_sync_all* must be 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 routine must 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* and *shmem_sync* routines allow the same *pSync* array to be used on consecutive calls as long as the PEs in the active set do 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_barrier_all*
- *shmem_barrier*
- *shmem_sync_all*
- *shmem_sync*
- *shmem_broadcast{32, 64}*
- *shmem_collect{32, 64}*
- *shmem_fcollect{32, 64}*
- Reductions for the following operations: AND, MAX, MIN, SUM, PROD, OR, XOR
- *shmem_alltoall{32, 64}*
- *shmem_alltoalls{32, 64}*

9.8.1 SHMEM_BARRIER_ALL

Registers the arrival of a PE at a barrier and blocks the PE until all other PEs arrive at the barrier and all local updates and remote memory updates on the default context are completed.

SYNOPSIS

C/C++:

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

— deprecation start —

FORTRAN:

```
CALL SHMEM_BARRIER_ALL
```

— deprecation end —

DESCRIPTION

Arguments

None.

API description

The *shmem_barrier_all* routine registers the arrival of a PE at a barrier. Barriers are a mechanism for synchronizing all PEs at once. This routine blocks the PE until all PEs have called *shmem_barrier_all*. In a multithreaded OpenSHMEM program, only the calling thread is blocked.

Prior to synchronizing with other PEs, *shmem_barrier_all* ensures completion of all previously issued memory stores and remote memory updates issued on the default context via OpenSHMEM AMOs and RMA routine calls such as *shmem_int_add*, *shmem_put32*, *shmem_put_nbi*, and *shmem_get_nbi*.

Return Values

None.

Notes

The *shmem_barrier_all* routine can be used to portably ensure that memory access operations observe remote updates in the order enforced by initiator PEs.

Calls to *shmem_ctx_quiet* can be performed prior to calling the barrier routine to ensure completion of operations issued on additional contexts.

EXAMPLES

The following *shmem_barrier_all* example is for *C11* programs:

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

int main(void)
{
   static int x = 1010;

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

   /* put to next  PE in a circular fashion */
   shmem_p(&x, 4, (me + 1) % npes);

   /* synchronize all PEs */
   shmem_barrier_all();
   printf("%d: x = %d\n", me, x);
   shmem_finalize();
   return 0;
}
```

9.8.2 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);
```

— deprecation start

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

IN	*PE_start*	The lowest PE number of the active set of PEs. *PE_start* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*logPE_stride*	The log (base 2) of the stride between consecutive PE numbers in the active set. *logPE_stride* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*PE_size*	The number of PEs in the active set. *PE_size* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*pSync*	A symmetric work array of size *SHMEM_BARRIER_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. In *Fortran*, *pSync* must be an array of elements of default integer type. Every element of this array must be initialized to *SHMEM_SYNC_VALUE* before any of the PEs in the active set enter *shmem_barrier* 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, the behavior is undefined.

The values of arguments *PE_start*, *logPE_stride*, and *PE_size* must be the same value 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 on the default context are complete before returning.

The same *pSync* array may be reused on consecutive calls to *shmem_barrier* if the same active set is used.

Return Values

None.

Notes

If the *pSync* array is initialized at the run time, all PEs must be synchronized before the first call to *shmem_barrier* (e.g., by *shmem_barrier_all*) to ensure the array has been initialized by all PEs before it is used.

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.

The *shmem_barrier* routine can be used to portably ensure that memory access operations observe remote updates in the order enforced by initiator PEs.

Calls to *shmem_ctx_quiet* can be performed prior to calling the barrier routine to ensure completion of operations issued on additional contexts.

EXAMPLES

The following barrier example is for *C11* programs:

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

int main(void)
{
   static int x = 10101;
   static long pSync[SHMEM_BARRIER_SYNC_SIZE];
   for (int i = 0; i < SHMEM_BARRIER_SYNC_SIZE; i++)
      pSync[i] = SHMEM_SYNC_VALUE;

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

   if (me % 2 == 0) {
      /* put to next even PE in a circular fashion */
      shmem_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);
   shmem_finalize();
   return 0;
}
```

9.8.3 SHMEM_SYNC_ALL

Registers the arrival of a PE at a barrier and suspends PE execution until all other PEs arrive at the barrier.

SYNOPSIS

C/C++:

```
void shmem_sync_all(void);
```

DESCRIPTION

Arguments

None.

API description

The *shmem_sync_all* routine registers the arrival of a PE at a barrier. Barriers are a fast mechanism for synchronizing all PEs at once. This routine blocks the PE until all PEs have called *shmem_sync_all*. In a multithreaded OpenSHMEM program, only the calling thread is blocked.

In contrast with the *shmem_barrier_all* routine, *shmem_sync_all* only ensures completion and visibility of previously issued memory stores and does not ensure completion of remote memory updates issued via OpenSHMEM routines.

Return Values

None.

Notes

The *shmem_sync_all* routine can be used to portably ensure that memory access operations observe remote updates in the order enforced by the initiator PEs, provided that the initiator PE ensures completion of remote updates with a call to *shmem_quiet* prior to the call to the *shmem_sync_all* routine.

9.8.4 SHMEM_SYNC

Performs all operations described in the *shmem_sync_all* interface but with respect to a subset of PEs defined by the active set.

SYNOPSIS

C/C++:

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

DESCRIPTION

Arguments

IN	*PE_start*	The lowest PE number of the active set of PEs. *PE_start* must be of type integer.
IN	*logPE_stride*	The log (base 2) of the stride between consecutive PE numbers in the active set. *logPE_stride* must be of type integer.
IN	*PE_size*	The number of PEs in the active set. *PE_size* must be of type integer.
IN	*pSync*	A symmetric work array. In *C/C++*, *pSync* must be of type *long* and size *SHMEM_BARRIER_SYNC_SIZE*. Every element of this array must be initialized to *SHMEM_SYNC_VALUE* before any of the PEs in the active set enter *shmem_sync* the first time.

API description

shmem_sync is a collective synchronization routine over an active set. Control returns from *shmem_sync* after all PEs in the active set (specified by *PE_start*, *logPE_stride*, and *PE_size*) have called *shmem_sync*.

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, 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 work array must be passed in *pSync* to all PEs in the active set.

In contrast with the *shmem_barrier* routine, *shmem_sync* only ensures completion and visibility of previously issued memory stores and does not ensure completion of remote memory updates issued via OpenSHMEM routines.

The same *pSync* array may be reused on consecutive calls to *shmem_sync* if the same active set is used.

Return Values

None.

Notes

If the *pSync* array is initialized at run time, another method of synchronization (e.g., *shmem_sync_all*) must be used before the initial use of that *pSync* array by *shmem_sync*.

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

The *shmem_sync* routine can be used to portably ensure that memory access operations observe remote updates in the order enforced by the initiator PEs, provided that the initiator PE ensures completion of remote updates with a call to *shmem_quiet* prior to the call to the *shmem_sync* routine.

EXAMPLES

The following *shmem_sync_all* and *shmem_sync* example is for *C11* programs:

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

int main(void)
{
   static int x = 10101;
   static long pSync[SHMEM_BARRIER_SYNC_SIZE];

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

   for (int i = 0; i < SHMEM_BARRIER_SYNC_SIZE; i++)
      pSync[i] = SHMEM_SYNC_VALUE;

   shmem_sync_all();

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

9.8.5 SHMEM_BROADCAST

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

SYNOPSIS

C/C++:

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

— deprecation start —

FORTRAN:

```
INTEGER nelems, PE_root, PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BCAST_SYNC_SIZE)
CALL SHMEM_BROADCAST4(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST8(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
```

```
CALL SHMEM_BROADCAST32(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size,pSync)
CALL SHMEM_BROADCAST64(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size,pSync)
```

deprecation end

DESCRIPTION

Arguments

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

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, the behavior is undefined.

The values of arguments *PE_root*, *PE_start*, *logPE_stride*, and *PE_size* must be the same value on all PEs in the active set. The same *dest* and *source* data objects and the same *pSync* work array must be passed by all PEs in the active set.

Before any PE calls a broadcast routine, the following conditions must be ensured:

- 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.

Otherwise, the behavior is undefined.

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 *source* data object may be safely reused.
- 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 *dest* and *source*
shmem_broadcast8, shmem_broadcast64	Any noncharacter type that has an element size of *64* bits. No *Fortran* derived types or *C/C++* structures are allowed.
shmem_broadcast4, shmem_broadcast32	Any noncharacter type that has an element size of *32* bits. No *Fortran* derived types or *C/C++* structures are allowed.

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.

The user 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, before its first use, 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>

int main(void)
{
   static long pSync[SHMEM_BCAST_SYNC_SIZE];
   for (int i = 0; i < SHMEM_BCAST_SYNC_SIZE; i++)
      pSync[i] = SHMEM_SYNC_VALUE;
   static long source[4], dest[4];

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

   if (me == 0)
      for (int i = 0; i < 4; i++)
         source[i] = i;

   shmem_broadcast64(dest, source, 4, 0, 0, 0, npes, pSync);
   printf("%d: %ld, %ld, %ld, %ld\n", me, dest[0], dest[1], dest[2], dest[3]);
   shmem_finalize();
   return 0;
}
```

Fortran example:

```
INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_BCAST_SYNC_SIZE)
INTEGER DEST, SOURCE, NLONG, PE_ROOT, PE_START,
&   LOGPE_STRIDE, PE_SIZE, PSYNC
COMMON /COM/ DEST, SOURCE

DATA PSYNC /SHMEM_BCAST_SYNC_SIZE*SHMEM_SYNC_VALUE/

CALL SHMEM_BROADCAST64(DEST, SOURCE, NLONG, 0, 4, 0, 4, PSYNC)
```

9.8.6 SHMEM_COLLECT, SHMEM_FCOLLECT

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

SYNOPSIS

C/C++:

```
void shmem_collect32(void *dest, const void *source, size_t nelems, int PE_start, int
    logPE_stride, int PE_size, long *pSync);
void shmem_collect64(void *dest, const void *source, size_t nelems, int PE_start, int
    logPE_stride, int PE_size, long *pSync);
void shmem_fcollect32(void *dest, const void *source, size_t nelems, int PE_start, int
    logPE_stride, int PE_size, long *pSync);
void shmem_fcollect64(void *dest, const void *source, size_t nelems, int PE_start, int
    logPE_stride, int PE_size, long *pSync);
```

— deprecation start —

FORTRAN:

```
INTEGER nelems
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)
CALL SHMEM_COLLECT4(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT8(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT32(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT64(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT4(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT8(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT32(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_FCOLLECT64(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
```

— deprecation end —

DESCRIPTION

Arguments

OUT	*dest*	A symmetric array. The *dest* argument must be large enough to accept the concatenation of the *source* arrays on all participating PEs. The data types are as follows: For *shmem_collect8*, *shmem_collect64*, *shmem_fcollect8*, and *shmem_fcollect64*, any data type with an element size of 64 bits. *Fortran* derived types, *Fortran* character type, and *C/C++* structures are not permitted. For *shmem_collect4*, *shmem_collect32*, *shmem_fcollect4*, and *shmem_fcollect32*, any data type with an element size of *32* bits. *Fortran* derived types, *Fortran* character type, and *C/C++* structures are not permitted.

IN	*source*	A symmetric data object that can be of any type permissible for the *dest* argument.
IN	*nelems*	The number of elements in the *source* array. *nelems* must be of type *size_t* for *C*. When using *Fortran*, it must be a default integer value.
IN	*PE_start*	The lowest PE number of the active set of PEs. *PE_start* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*logPE_stride*	The log (base *2*) of the stride between consecutive PE numbers in the active set. *logPE_stride* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*PE_size*	The number of PEs in the active set. *PE_size* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*pSync*	A symmetric work array of size *SHMEM_COLLECT_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. In *Fortran*, *pSync* must be an array of elements of default integer type. Every element of this array must be initialized with the value *SHMEM_SYNC_VALUE* before any of the PEs in the active set enter *shmem_collect* or *shmem_fcollect*.

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 the same value on all PEs in the active set. The same *dest* and *source* arrays and the same *pSync* work array must be passed by 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 and the *source* array may be safely reused. 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.

The user 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:

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

int main(void)
{
   static long lock = 0;
   static long pSync[SHMEM_COLLECT_SYNC_SIZE];
   for (int i = 0; i < SHMEM_COLLECT_SYNC_SIZE; i++)
      pSync[i] = SHMEM_SYNC_VALUE;

   shmem_init();
   int me = shmem_my_pe();
   int npes = shmem_n_pes();
   int my_nelem = me + 1; /* linearly increasing number of elements with PE */
   int total_nelem = (npes * (npes + 1)) / 2;

   int* source = (int*) shmem_malloc(npes*sizeof(int)); /* symmetric alloc */
   int* dest = (int*) shmem_malloc(total_nelem*sizeof(int));

   for (int i = 0; i < my_nelem; i++)
      source[i] = (me * (me + 1)) / 2 + i;
   for (int i = 0; i < total_nelem; i++)
      dest[i] = -9999;

   shmem_barrier_all(); /* Wait for all PEs to update source/dest */

   shmem_collect32(dest, source, my_nelem, 0, 0, npes, pSync);

   shmem_set_lock(&lock); /* Lock prevents interleaving printfs */
   printf("%d: %d", me, dest[0]);
   for (int i = 1; i < total_nelem; i++)
      printf(", %d", dest[i]);
   printf("\n");
   shmem_clear_lock(&lock);
   shmem_finalize();
   return 0;
}
```

The following *SHMEM_COLLECT* example is for *Fortran* programs:

```
INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_COLLECT_SYNC_SIZE)
DATA PSYNC /SHMEM_COLLECT_SYNC_SIZE*SHMEM_SYNC_VALUE/

CALL SHMEM_COLLECT4(DEST, SOURCE, 64, PE_START, LOGPE_STRIDE,
&  PE_SIZE, PSYNC)
```

9.8.7 SHMEM_REDUCTIONS

The following functions perform reduction operations across all PEs in a set of PEs.

SYNOPSIS

9.8.7.1 AND Performs a bitwise AND reduction across a set of PEs.

C/C++:

```
void shmem_short_and_to_all(short *dest, const 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, const 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, const 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, const long long *source, int nreduce, int
     PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
```

— deprecation start —

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)
```

— deprecation end —

9.8.7.2 MAX Performs a maximum-value reduction across a set of PEs.

C/C++:

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

— deprecation start —

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)
```

— deprecation end —

9.8.7.3 MIN Performs a minimum-value reduction across a set of PEs.

C/C++:

```
void shmem_short_min_to_all(short *dest, const short *source, int nreduce, int PE_start, int
     logPE_stride, int PE_size, short *pWrk, long *pSync);
```

```
void shmem_int_min_to_all(int *dest, const int *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_double_min_to_all(double *dest, const double *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_min_to_all(float *dest, const float *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_long_min_to_all(long *dest, const long *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *dest, const long double *source, int nreduce,
    int PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *dest, const long long *source, int nreduce, int
    PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
```

— deprecation start

FORTRAN:

```
CALL SHMEM_INT4_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_INT8_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL4_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL8_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_REAL16_MIN_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
```

deprecation end —

9.8.7.4 SUM Performs a sum reduction across a set of PEs.

C/C++:

```
void shmem_complexd_sum_to_all(double _Complex *dest, const double _Complex *source, int
    nreduce, int PE_start, int logPE_stride, int PE_size, double _Complex *pWrk, long
    *pSync);
void shmem_complexf_sum_to_all(float _Complex *dest, const float _Complex *source, int
    nreduce, int PE_start, int logPE_stride, int PE_size, float _Complex *pWrk, long
    *pSync);
void shmem_short_sum_to_all(short *dest, const short *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_int_sum_to_all(int *dest, const int *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_double_sum_to_all(double *dest, const double *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_sum_to_all(float *dest, const float *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_long_sum_to_all(long *dest, const long *source, int nreduce, int PE_start, int
    logPE_stride,int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_sum_to_all(long double *dest, const long double *source, int nreduce,
    int PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_sum_to_all(long long *dest, const long long *source, int nreduce, int
    PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
```

— deprecation start

FORTRAN:

```
CALL SHMEM_COMP4_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_COMP8_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
```

```
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)
```

— deprecation end —

9.8.7.5 PROD Performs a product reduction across a set of PEs.

C/C++:

```
void shmem_complexd_prod_to_all(double _Complex *dest, const 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, const float _Complex *source, int
    nreduce, int PE_start, int logPE_stride, int PE_size, float _Complex *pWrk, long
    *pSync);
void shmem_short_prod_to_all(short *dest, const short *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_int_prod_to_all(int *dest, const int *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_double_prod_to_all(double *dest, const 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, const float *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_long_prod_to_all(long *dest, const 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, const 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, const long long *source, int nreduce, int
    PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
```

— deprecation start —

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)
```

— deprecation end —

9.8.7.6 OR Performs a bitwise OR reduction across a set of PEs.

C/C++:

```
void shmem_short_or_to_all(short *dest, const short *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_int_or_to_all(int *dest, const int *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_or_to_all(long *dest, const long *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlong_or_to_all(long long *dest, const long long *source, int nreduce, int
    PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
```

— deprecation start

FORTRAN:

```
CALL SHMEM_INT4_OR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_INT8_OR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
```

deprecation end —

9.8.7.7 XOR Performs a bitwise exclusive OR (XOR) reduction across a set of PEs.

C/C++:

```
void shmem_short_xor_to_all(short *dest, const short *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, short *pWrk, long *pSync);
void shmem_int_xor_to_all(int *dest, const int *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_xor_to_all(long *dest, const long *source, int nreduce, int PE_start, int
    logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlong_xor_to_all(long long *dest, const long long *source, int nreduce, int
    PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
```

— deprecation start

FORTRAN:

```
CALL SHMEM_INT4_XOR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
CALL SHMEM_INT8_XOR_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk,
    pSync)
```

deprecation end —

DESCRIPTION

Arguments

OUT	*dest*	A symmetric array, of length *nreduce* elements, to receive the result of the reduction routines. The data type of *dest* varies with the version of the reduction routine being called. When calling from *C/C++*, refer to the SYNOPSIS section for data type information.
IN	*source*	A symmetric array, of length *nreduce* elements, that contains one element for each separate reduction routine. The *source* argument must have the same data type as *dest*.
IN	*nreduce*	The number of elements in the *dest* and *source* arrays. *nreduce* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*PE_start*	The lowest PE number of the active set of PEs. *PE_start* must be of type integer. When using *Fortran*, it must be a default integer value.

IN	*logPE_stride*	The log (base 2) of the stride between consecutive PE numbers in the active set. *logPE_stride* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*PE_size*	The number of PEs in the active set. *PE_size* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*pWrk*	A symmetric work array of size at least max(*nreduce*/2 + 1, *SHMEM_REDUCE_MIN_WRKDATA_SIZE*) elements.
IN	*pSync*	A symmetric work array of size *SHMEM_REDUCE_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. In *Fortran*, *pSync* must be an array of elements of default integer type. Every element of this array must be initialized with the value *SHMEM_SYNC_VALUE* before any of the PEs in the active set 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, the behavior is undefined.

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, the following conditions must be ensured:

- 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*.

Otherwise, the behavior is undefined.

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

The complex-typed interfaces are only provided for sum and product reductions. When the *C* translation environment does not support complex types [7], an OpenSHMEM implementation is not required to provide support for these complex-typed interfaces.

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

Routine	Data type
shmem_int8_and_to_all	Integer, with an element size of 8 bytes.
shmem_int4_and_to_all	Integer, with an element size of 4 bytes.
shmem_comp8_max_to_all	Complex, with an element size equal to two 8-byte real values.
shmem_int4_max_to_all	Integer, with an element size of 4 bytes.

[7] That is, under *C* language standards prior to *C99* or under *C11* when *__STDC_NO_COMPLEX__* is defined to 1

shmem_int8_max_to_all	Integer, with an element size of 8 bytes.
shmem_real4_max_to_all	Real, with an element size of 4 bytes.
shmem_real16_max_to_all	Real, with an element size of 16 bytes.
shmem_int4_min_to_all	Integer, with an element size of 4 bytes.
shmem_int8_min_to_all	Integer, with an element size of 8 bytes.
shmem_real4_min_to_all	Real, with an element size of 4 bytes.
shmem_real8_min_to_all	Real, with an element size of 8 bytes.
shmem_real16_min_to_all	Real,with an element size of 16 bytes.
shmem_comp4_sum_to_all	Complex, with an element size equal to two 4-byte real values.
shmem_comp8_sum_to_all	Complex, with an element size equal to two 8-byte real values.
shmem_int4_sum_to_all	Integer, with an element size of 4 bytes.
shmem_int8_sum_to_all	Integer, with an element size of 8 bytes..
shmem_real4_sum_to_all	Real, with an element size of 4 bytes.
shmem_real8_sum_to_all	Real, with an element size of 8 bytes.
shmem_real16_sum_to_all	Real, with an element size of 16 bytes.
shmem_comp4_prod_to_all	Complex, with an element size equal to two 4-byte real values.
shmem_comp8_prod_to_all	Complex, with an element size equal to two 8-byte real values.
shmem_int4_prod_to_all	Integer, with an element size of 4 bytes.
shmem_int8_prod_to_all	Integer, with an element size of 8 bytes.
shmem_real4_prod_to_all	Real, with an element size of 4 bytes.
shmem_real8_prod_to_all	Real, with an element size of 8 bytes.
shmem_real16_prod_to_all	Real, with an element size of 16 bytes.
shmem_int8_or_to_all	Integer, with an element size of 8 bytes.
shmem_int4_or_to_all	Integer, with an element size of 4 bytes.
shmem_int8_xor_to_all	Integer, with an element size of 8 bytes.
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. The user 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.

```
INCLUDE "shmem.fh"

INTEGER PSYNC(SHMEM_REDUCE_SYNC_SIZE)
DATA PSYNC /SHMEM_REDUCE_SYNC_SIZE*SHMEM_SYNC_VALUE/
PARAMETER (NR=1)
INTEGER*4 PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
INTEGER FOO, FOOAND
SAVE FOO, FOOAND, PWRK
```

```
INTRINSIC SHMEM_MY_PE()

FOO = SHMEM_MY_PE()
IF ( MOD(SHMEM_MY_PE() .EQ. 0) THEN
    IF ( MOD(SHMEM_N_PES()(),2) .EQ. 0) THEN
       CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2, &
   PWRK, PSYNC)
    ELSE
       CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2+1, &
   PWRK, PSYNC)

    ENDIF
    PRINT*,'Result on PE ',SHMEM_MY_PE(),' is ',FOOAND
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.

```
INCLUDE "shmem.fh"

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

IF ( MOD(SHMEM_MY_PE() .EQ. 0) THEN
        CALL SHMEM_INT8_OR_TO_ALL(FOOOR, FOO, NR, 0, 1, N$PES/2,
&  PWRK, PSYNC)
        PRINT*,'Result on PE ',SHMEM_MY_PE(),' is ',FOOOR
ENDIF
```

This *Fortran* example statically initializes the *pSync* array and computes the exclusive *XOR* of 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, FOOXOR, PWRK(MAX(NR/2+1,SHMEM_REDUCE_MIN_WRKDATA_SIZE))
COMMON /COM/ FOO, FOOXOR, PWRK
INTRINSIC SHMEM_MY_PE()

IF ( MOD(SHMEM_MY_PE() .EQ. 0) THEN
       CALL SHMEM_REAL8_XOR_TO_ALL(FOOXOR, FOO, NR, 0, 1, N$PES/2,
&  PWRK, PSYNC)
       PRINT*,'Result on PE ',SHMEM_MY_PE(),' is ',FOOXOR
ENDIF
```

9.8.8 SHMEM_ALLTOALL

shmem_alltoall is a collective routine where each PE exchanges a fixed amount of data with all other PEs in the active set.

SYNOPSIS

C/C++:

```
void shmem_alltoall32(void *dest, const void *source, size_t nelems, int PE_start, int
    logPE_stride, int PE_size, long *pSync);
```

```
void shmem_alltoall64(void *dest, const void *source, size_t nelems, int PE_start, int
    logPE_stride, int PE_size, long *pSync);
```

— deprecation start —

FORTRAN:

```
INTEGER pSync(SHMEM_ALLTOALL_SYNC_SIZE)
INTEGER PE_start, logPE_stride, PE_size, nelems
CALL SHMEM_ALLTOALL32(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_ALLTOALL64(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
```

— deprecation end —

DESCRIPTION

Arguments

OUT	*dest*	A symmetric data object large enough to receive the combined total of *nelems* elements from each PE in the active set.
IN	*source*	A symmetric data object that contains *nelems* elements of data for each PE in the active set, ordered according to destination PE.
IN	*nelems*	The number of elements to exchange for each PE. *nelems* must be of type size_t for *C/C++*. When using *Fortran*, it must be a default integer value.
IN	*PE_start*	The lowest PE number of the active set of PEs. *PE_start* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*logPE_stride*	The log (base 2) of the stride between consecutive PE numbers in the active set. *logPE_stride* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*PE_size*	The number of PEs in the active set. *PE_size* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*pSync*	A symmetric work array of size *SHMEM_ALLTOALL_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. In *Fortran*, *pSync* must be an array of elements of default integer type. Every element of this array must be initialized with the value *SHMEM_SYNC_VALUE* before any of the PEs in the active set enter the routine.

API description

The *shmem_alltoall* routines are collective routines. Each PE in the active set exchanges *nelems* data elements of size 32 bits (for *shmem_alltoall32*) or 64 bits (for *shmem_alltoall64*) with all other PEs in the set. The data being sent and received are stored in a contiguous symmetric data object. The total size of each PEs *source* object and *dest* object is *nelems* times the size of an element (32 bits or 64 bits) times *PE_size*. The *source* object contains *PE_size* blocks of data (the size of each block defined by *nelems*) and each block of data is sent to a different PE. Given a PE i that is the k^{th}PE in the active set and a PE j that is the l^{th}PE in the active set, PE i sends the l^{th}block of its *source* object to the k^{th}block of the *dest* object of PE j.

As with all OpenSHMEM collective routines, this routine assumes that only PEs in the active set call the routine. If a PE not in the active set calls an OpenSHMEM collective routine, the behavior is undefined.

The values of arguments *nelems*, *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 *shmem_alltoall* routine, the following conditions must be ensured:

- The *pSync* array on all PEs in the active set is not still in use from a prior call to a *shmem_alltoall* routine.
- The *dest* data object on all PEs in the active set is ready to accept the *shmem_alltoall* data.

Otherwise, the behavior is undefined.

Upon return from a *shmem_alltoall* routine, the following is true for the local PE: Its *dest* symmetric data object is completely updated and the data has been copied out of the *source* data object. 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 *dest* and *source*
shmem_alltoall64	*64* bits aligned.
shmem_alltoall32	*32* bits aligned.

Return Values

None.

Notes

This routine restores *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. The user 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 *shmem_alltoall* 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 *shmem_alltoall* routine only if none of the PEs in the active set are still processing a prior OpenSHMEM *shmem_alltoall* routine call that used the same *pSync* array. In general, this can be ensured only by doing some type of synchronization.

EXAMPLES

This example shows a *shmem_alltoall64* on two long elements among all PEs.

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

int main(void)
{
   static long pSync[SHMEM_ALLTOALL_SYNC_SIZE];
   for (int i = 0; i < SHMEM_ALLTOALL_SYNC_SIZE; i++)
      pSync[i] = SHMEM_SYNC_VALUE;

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

   const int count = 2;
   int64_t* dest = (int64_t*) shmem_malloc(count * npes * sizeof(int64_t));
   int64_t* source = (int64_t*) shmem_malloc(count * npes * sizeof(int64_t));

   /* assign source values */
   for (int pe = 0; pe < npes; pe++) {
      for (int i = 0; i < count; i++) {
```

```
         source[(pe * count) + i] = me + pe;
         dest[(pe * count) + i] = 9999;
      }
   }
   /* wait for all PEs to update source/dest */
   shmem_barrier_all();

   /* alltoall on all PES */
   shmem_alltoall64(dest, source, count, 0, 0, npes, pSync);

   /* verify results */
   for (int pe = 0; pe < npes; pe++) {
      for (int i = 0; i < count; i++) {
         if (dest[(pe * count) + i] != pe + me) {
            printf("[%d] ERROR: dest[%d]=%" PRId64 ", should be %d\n",
               me, (pe * count) + i, dest[(pe * count) + i], pe + me);
         }
      }
   }

   shmem_free(dest);
   shmem_free(source);
   shmem_finalize();
   return 0;
}
```

9.8.9 SHMEM_ALLTOALLS

shmem_alltoalls is a collective routine where each PE exchanges a fixed amount of strided data with all other PEs in the active set.

SYNOPSIS

C/C++:

```
void shmem_alltoalls32(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
void shmem_alltoalls64(void *dest, const void *source, ptrdiff_t dst, ptrdiff_t sst, size_t
    nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
```

— deprecation start —

FORTRAN:

```
INTEGER pSync(SHMEM_ALLTOALLS_SYNC_SIZE)
INTEGER dst, sst, PE_start, logPE_stride, PE_size
INTEGER nelems
CALL SHMEM_ALLTOALLS32(dest, source, dst, sst, nelems, PE_start, logPE_stride, PE_size,
    pSync)
CALL SHMEM_ALLTOALLS64(dest, source, dst, sst, nelems, PE_start, logPE_stride, PE_size,
    pSync)
```

— deprecation end —

DESCRIPTION

Arguments

OUT	*dest*	A symmetric data object large enough to receive the combined total of *nelems* elements from each PE in the active set.
IN	*source*	A symmetric data object that contains *nelems* elements of data for each PE in the active set, ordered according to destination PE.

IN	*dst*	The stride between consecutive elements of the *dest* data object. The stride is scaled by the element size. A value of *1* indicates contiguous data. *dst* must be of type *ptrdiff_t*. When using *Fortran*, it must be a default integer value.
IN	*sst*	The stride between consecutive elements of the *source* data object. The stride is scaled by the element size. A value of *1* indicates contiguous data. *sst* must be of type *ptrdiff_t*. When using *Fortran*, it must be a default integer value.
IN	*nelems*	The number of elements to exchange for each PE. *nelems* must be of type size_t for *C/C++*. When using *Fortran*, it must be a default integer value.
IN	*PE_start*	The lowest PE number of the active set of PEs. *PE_start* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*logPE_stride*	The log (base 2) of the stride between consecutive PE numbers in the active set. *logPE_stride* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*PE_size*	The number of PEs in the active set. *PE_size* must be of type integer. When using *Fortran*, it must be a default integer value.
IN	*pSync*	A symmetric work array of size *SHMEM_ALLTOALLS_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. In *Fortran*, *pSync* must be an array of elements of default integer type. Every element of this array must be initialized with the value *SHMEM_SYNC_VALUE* before any of the PEs in the active set enter the routine.

API description

The *shmem_alltoalls* routines are collective routines. Each PE in the active set exchanges *nelems* strided data elements of size 32 bits (for *shmem_alltoalls32*) or 64 bits (for *shmem_alltoalls64*) with all other PEs in the set. Both strides, *dst* and *sst*, must be greater than or equal to *1*. Given a PE *i* that is the k^{th}PE in the active set and a PE *j* that is the l^{th}PE in the active set, PE *i* sends the $sst*l^{th}$block of the *source* data object to the $dst*k^{th}$block of the *dest* data object on PE *j*.

As with all OpenSHMEM collective routines, these routines assume 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 *dst*, *sst*, *nelems*, *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 *shmem_alltoalls* routine, the following conditions must be ensured:

- The *pSync* array on all PEs in the active set is not still in use from a prior call to a *shmem_alltoall* routine.
- The *dest* data object on all PEs in the active set is ready to accept the *shmem_alltoalls* data.

Otherwise, the behavior is undefined.

Upon return from a *shmem_alltoalls* routine, the following is true for the local PE: Its *dest* symmetric data object is completely updated and the data has been copied out of the *source* data object. 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 *dest* and *source*

shmem_alltoalls64	*64* bits aligned.
shmem_alltoalls32	*32* bits aligned.

Return Values

None.

Notes

This routine restores *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. The user 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 *shmem_alltoalls* 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 *shmem_alltoalls* routine only if none of the PEs in the active set are still processing a prior OpenSHMEM *shmem_alltoalls* routine call that used the same *pSync* array. In general, this can be ensured only by doing some type of synchronization.

EXAMPLES

This example shows a *shmem_alltoalls64* on two long elements among all PEs.

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

int main(void)
{
   static long pSync[SHMEM_ALLTOALLS_SYNC_SIZE];
   for (int i = 0; i < SHMEM_ALLTOALLS_SYNC_SIZE; i++)
      pSync[i] = SHMEM_SYNC_VALUE;

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

   const int count = 2;
   const ptrdiff_t dst = 2;
   const ptrdiff_t sst = 3;
   int64_t* dest = (int64_t*) shmem_malloc(count * dst * npes * sizeof(int64_t));
   int64_t* source = (int64_t*) shmem_malloc(count * sst * npes * sizeof(int64_t));

   /* assign source values */
   for (int pe = 0; pe < npes; pe++) {
      for (int i = 0; i < count; i++) {
         source[sst * ((pe * count) + i)] = me + pe;
         dest[dst * ((pe * count) + i)] = 9999;
      }
   }
   /* wait for all PEs to update source/dest */
   shmem_barrier_all();

   /* alltoalls on all PES */
   shmem_alltoalls64(dest, source, dst, sst, count, 0, 0, npes, pSync);

   /* verify results */
   for (int pe = 0; pe < npes; pe++) {
      for (int i = 0; i < count; i++) {
         int j = dst * ((pe * count) + i);
         if (dest[j] != pe + me) {
            printf("[%d] ERROR: dest[%d]=%" PRId64 ", should be %d\n",
```

```
                    me, j, dest[j], pe + me);
            }
        }
    }

    shmem_free(dest);
    shmem_free(source);
    shmem_finalize();
    return 0;
}
```

9.9 Point-To-Point Synchronization Routines

The following section discusses OpenSHMEM APIs that provide a mechanism for synchronization between two PEs based on the value of a symmetric data object. The point-to-point synchronization routines can be used to portably ensure that memory access operations observe remote updates in the order enforced by the initiator PE using the *shmem_fence* and *shmem_quiet* routines.

Where appropriate compiler support is available, OpenSHMEM provides type-generic point-to-point synchronization interfaces via *C11* generic selection. Such type-generic routines are supported for the “point-to-point synchronization types” identified in Table 7.

The point-to-point synchronization types include some of the exact-width integer types defined in *stdint.h* by *C99* §7.18.1.1 and *C11* §7.20.1.1. When the *C* translation environment does not provide exact-width integer types with *stdint.h*, an OpenSHMEM implemementation is not required to provide support for these types.

TYPE	*TYPENAME*
short	short
int	int
long	long
long long	longlong
unsigned short	ushort
unsigned int	uint
unsigned long	ulong
unsigned long long	ulonglong
int32_t	int32
int64_t	int64
uint32_t	uint32
uint64_t	uint64
size_t	size
ptrdiff_t	ptrdiff

Table 7: Point-to-Point Synchronization Types and Names

The point-to-point synchronization interface provides named constants whose values are integer constant expressions that specify the comparison operators used by OpenSHMEM synchronization routines. The constant names and associated operations are presented in Table 8. For Fortran, the constant names of Table 8 shall be identifiers for integer parameters of default kind corresponding to the associated comparison operation.

9.9.1 SHMEM_WAIT_UNTIL

Wait for a variable on the local PE to change.

SYNOPSIS

C11:

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

Table 8: Point-to-Point Comparison Constants

```
void shmem_wait_until(TYPE *ivar, int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 7.

C/C++:

```
void shmem_<TYPENAME>_wait_until(TYPE *ivar, int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 7.

— deprecation start —

```
void shmem_wait_until(long *ivar, int cmp, long cmp_value);
void shmem_wait(long *ivar, long cmp_value);
void shmem_<TYPENAME>_wait(TYPE *ivar, TYPE cmp_value);
```

where *TYPE* is one of {*short*, *int*, *long*, *long long*} and has a corresponding *TYPENAME* specified by Table 7.

— deprecation end —

— deprecation start —

FORTRAN:

```
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)
```

— deprecation end —

DESCRIPTION

Arguments

OUT	*ivar*	A remotely accessible integer variable. When using *C/C++*, the type of *ivar* should match that implied in the SYNOPSIS section.
IN	*cmp*	The compare operator that compares *ivar* with *cmp_value*. When using *Fortran*, it must be of default kind. When using *C/C++*, it must be of type *int*.
IN	*cmp_value*	*cmp_value* must be of type integer. When using *C/C++*, the type of *cmp_value* should match that implied in the SYNOPSIS section. When using *Fortran*, cmp_value must be an integer of the same size and kind as *ivar*.

API description

shmem_wait and *shmem_wait_until* wait for *ivar* to be changed by a write or an atomic operation issued by a PE. These routines can be used for point-to-point direct synchronization. A call to *shmem_wait* does not return until a 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 a PE changes *ivar* to satisfy the condition implied by *cmp* and *cmp_value*. 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*.

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

Routine	Data type
shmem_wait, shmem_wait_until	default INTEGER
shmem_int4_wait, shmem_int4_wait_until	INTEGER*4
shmem_int8_wait, shmem_int8_wait_until	INTEGER*8

Return Values

None.

Notes

As of OpenSHMEM 1.4, the *shmem_wait* routine is deprecated, however, *shmem_wait* is equivalent to *shmem_wait_until* where *cmp* is *SHMEM_CMP_NE*.

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:

```
INCLUDE "shmem.fh"

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:

```
INCLUDE "shmem.fh"

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

The following *C/C++* call waits until the value in *ivar* is set to be less than zero 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:

```
INCLUDE "shmem.fh"

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
```

9.9.2 SHMEM_TEST

Test whether a variable on the local PE has changed.

SYNOPSIS

C11:

```
int shmem_test(TYPE *ivar, int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 7.

C/C++:

```
int shmem_<TYPENAME>_test(TYPE *ivar, int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 7.

DESCRIPTION

Arguments

OUT	*ivar*	A pointer to a remotely accessible data object.
IN	*cmp*	The comparison operator that compares *ivar* with *cmp_value*.
IN	*cmp_value*	The value against which the object pointed to by *ivar* will be compared.

API description

shmem_test tests the numeric comparison of the symmetric object pointed to by *ivar* with the value *cmp_value* according to the comparison operator *cmp*.

Return Values

shmem_test returns 1 if the comparison of the symmetric object pointed to by *ivar* with the value *cmp_value* according to the comparison operator *cmp* evalutes to true; otherwise, it returns 0.

Notes

None.

EXAMPLES

The following example demonstrates the use of *shmem_test* to wait on an array of symmetric objects and return the index of an element that satisfies the specified condition.

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

int user_wait_any(long *ivar, int count, int cmp, long value)
{
  int idx = 0;
  while (!shmem_test(&ivar[idx], cmp, value))
    idx = (idx + 1) % count;
  return idx;
}

int main(void)
{
  shmem_init();
  const int mype = shmem_my_pe();
  const int npes = shmem_n_pes();

  long *wait_vars = shmem_calloc(npes, sizeof(long));
  if (mype == 0)
  {
    int who = user_wait_any(wait_vars, npes, SHMEM_CMP_NE, 0);
    printf("PE %d observed first update from PE %d\n", mype, who);
  }
  else
    shmem_p(&wait_vars[mype], mype, 0);

  shmem_free(wait_vars);
  shmem_finalize();
  return 0;
}
```

9.10 Memory Ordering Routines

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

9.10.1 SHMEM_FENCE

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

SYNOPSIS

C/C++:

```
void shmem_fence(void);
void shmem_ctx_fence(shmem_ctx_t ctx);
```

— deprecation start —

FORTRAN:

```
CALL SHMEM_FENCE
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.

API description

This routine assures ordering of delivery of *Put*, AMO, memory store, and nonblocking *Put* routines to symmetric data objects. All *Put*, AMO, memory store, and nonblocking *Put* routines to symmetric data objects issued to a particular remote PE on the given context prior to the call to *shmem_fence* are guaranteed to be delivered before any subsequent *Put*, AMO, memory store, and nonblocking *Put* routines to symmetric data objects to the same PE. *shmem_fence* guarantees order of delivery, not completion. It does not guarantee order of delivery of nonblocking *Get* routines.

Return Values

None.

Notes

shmem_fence only provides per-PE ordering guarantees and does not guarantee completion of delivery. *shmem_fence* also does not have an effect on the ordering between memory accesses issued by the target PE. *shmem_wait_until*, *shmem_test*, *shmem_barrier*, *shmem_barrier_all* routines can be called by the target PE to guarantee ordering of its memory accesses. There is a subtle difference between *shmem_fence* and *shmem_quiet*, in that, *shmem_quiet* guarantees completion of *Put*, AMO, memory store, and nonblocking *Put* 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*, AMO, memory store, and nonblocking *Put* routines to symmetric data objects is desired when multiple remote PEs are involved.

In an OpenSHMEM program with multithreaded PEs, it is the user's responsibility to ensure ordering between operations issued by the threads in a PE that target symmetric memory (e.g. *Put*, AMO, memory stores, and nonblocking routines) and calls by threads in that PE to *shmem_fence*. The *shmem_fence* routine can enforce memory store ordering only for the calling thread. Thus, to ensure ordering for memory stores performed by a thread that is not the thread calling *shmem_fence*, the update must be made visible to the calling thread according to the rules of the memory model associated with the threading environment.

EXAMPLES

The following example uses *shmem_fence* in a *C11* program:

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

int main(void)
{
   int src = 99;
   long source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   static long dest[10];
   static int targ;
   shmem_init();
   int me = shmem_my_pe();
   if (me == 0) {
      shmem_put(dest, source, 10, 1); /* put1 */
      shmem_put(dest, source, 10, 2); /* put2 */
      shmem_fence();
      shmem_put(&targ, &src, 1, 1); /* put3 */
      shmem_put(&targ, &src, 1, 2); /* put4 */
   }
   shmem_barrier_all();  /* sync sender and receiver */
   printf("dest[0] on PE %d is %ld\n", me, dest[0]);
   shmem_finalize();
   return 0;
}
```

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

9.10.2 SHMEM_QUIET

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

SYNOPSIS

C/C++:

```
void shmem_quiet(void);
void shmem_ctx_quiet(shmem_ctx_t ctx);
```

— deprecation start —

FORTRAN:

```
CALL SHMEM_QUIET
```

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	The context on which to perform the operation. When this argument is not provided, the operation is performed on *SHMEM_CTX_DEFAULT*.

API description

The *shmem_quiet* routine ensures completion of *Put*, AMO, memory store, and nonblocking *Put* and *Get* routines on symmetric data objects issued by the calling PE on the given context. All *Put*, AMO, memory store, and nonblocking *Put* and *Get* 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*, AMO, memory store, and nonblocking *Put* and *Get* routines to symmetric data objects initiated by the calling PE. For example, one might use *shmem_quiet* to await delivery of a block of data before issuing another *Put* or nonblocking *Put* routine, 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, and nonblocking *Put* and *Get* routines) to symmetric data objects on all PEs.

In an OpenSHMEM program with multithreaded PEs, it is the user's responsibility to ensure ordering between operations issued by the threads in a PE that target symmetric memory (e.g. *Put*, AMO, memory stores, and nonblocking routines) and calls by threads in that PE to *shmem_quiet*. The *shmem_quiet* routine can enforce memory store ordering only for the calling thread. Thus, to ensure ordering for memory stores performed by a thread that is not the thread calling *shmem_quiet*, the update must be made visible to the calling thread according to the rules of the memory model associated with the threading environment.

A call to *shmem_quiet* by a thread completes the operations posted prior to calling *shmem_quiet*. If the user intends to also complete operations issued by a thread that is not the thread calling *shmem_quiet*, the user must ensure that the operations are performed prior to the call to *shmem_quiet*. This may require the use of a synchronization operation provided by the threading package. For example, when using POSIX Threads, the user may call the *pthread_barrier_wait* routine to ensure that all threads have issued operations before a thread calls *shmem_quiet*.

shmem_quiet does not have an effect on the ordering between memory accesses issued by the target PE. *shmem_wait_until*, *shmem_test*, *shmem_barrier*, *shmem_barrier_all* routines can be called by the target PE to guarantee ordering of its memory accesses.

EXAMPLES

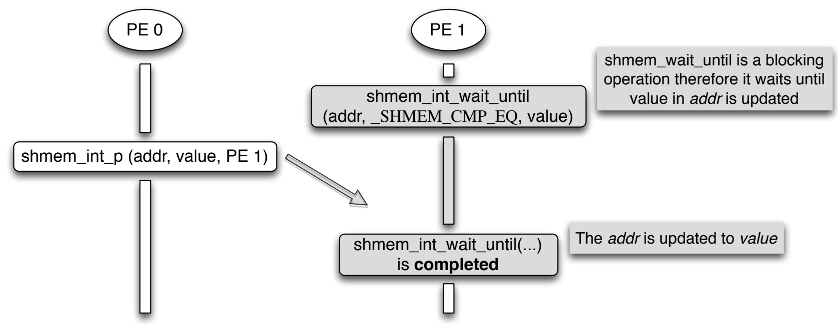
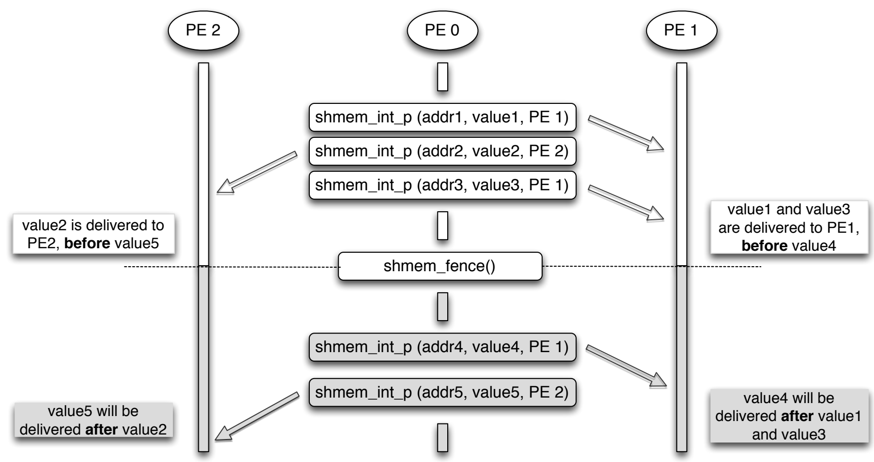
The following example uses *shmem_quiet* in a *C11* program:

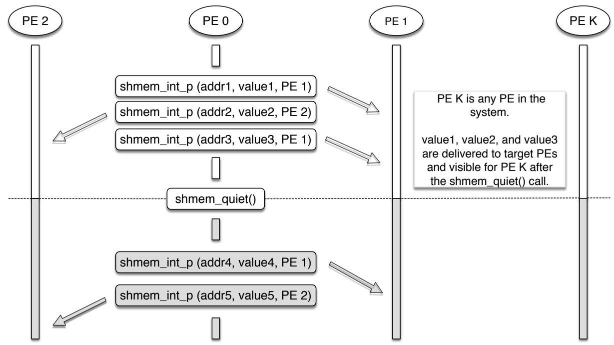
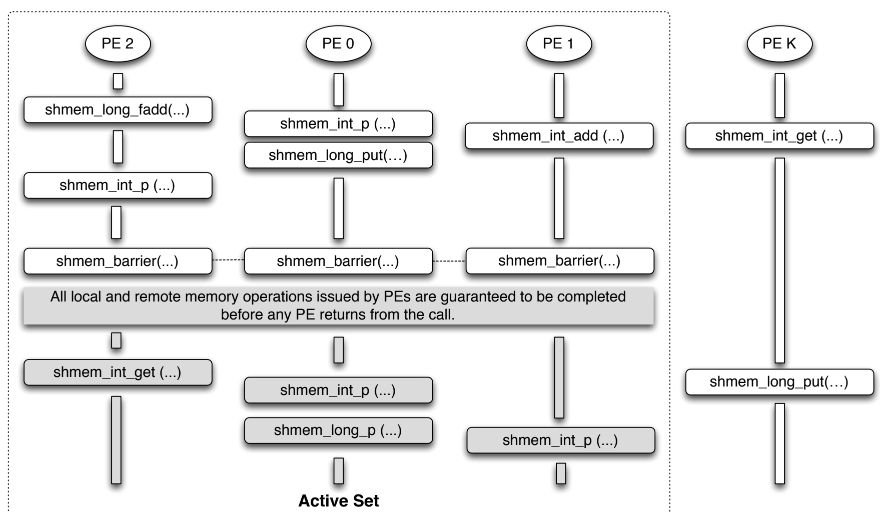
```
#include <stdio.h>
#include <shmem.h>

int main(void)
{
   static long dest[3];
   static long source[3] = { 1, 2, 3 };
   static int targ;
   static int src = 90;
   long x[3] = { 0 };
   int y = 0;
   shmem_init();
   int me = shmem_my_pe();
   if (me == 0) {
      shmem_put(dest, source, 3, 1); /* put1 */
      shmem_put(&targ, &src, 1, 2);  /* put2 */
      shmem_quiet();
      shmem_get(x, dest, 3, 1);   /* gets updated value from dest on PE 1 to local array x */
      shmem_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_put(&targ, &src, 1, 1); /* put3 */
      shmem_put(&targ, &src, 1, 2); /* put4 */
   }
   shmem_finalize();
   return 0;
}
```

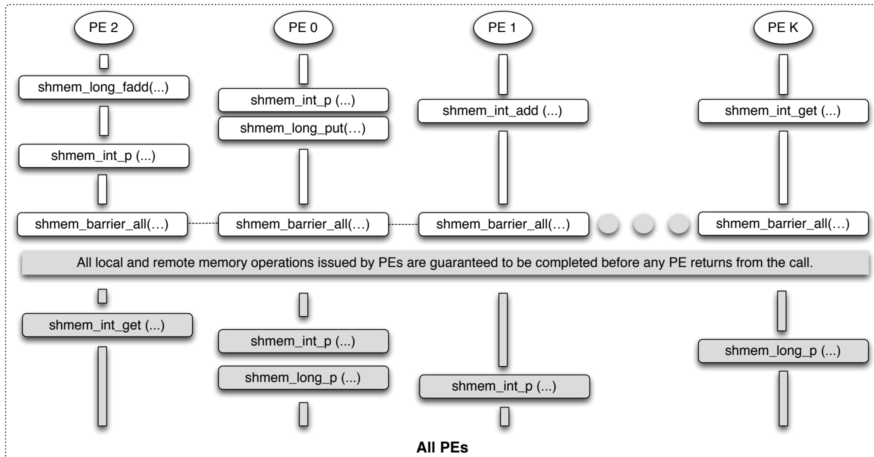
Put1 and *put2* will be completed and visible before *put3* and *put4*.

9.10.3 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, stores, and nonblocking *Put* and *Get* routines on symmetric data cannot be guaranteed until some form of synchronization or ordering is introduced in the user's program. 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 *shmem_wait_until*	
	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.
Ordering puts issued by a local PE *shmem_fence*	
	All *Put*, AMO, store, and nonblocking *Put* 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*, AMO, store, and nonblocking *Put* and *Get* 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 shall 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.

9.11 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*.

9.11.1 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);
```

— deprecation start —

FORTRAN:

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

— deprecation end —

DESCRIPTION

Arguments

IN	*lock*	A symmetric data object that is a scalar variable or an array of length *1*. This data object must be set to *0* on all PEs prior to the first use. *lock* must be of type *long*. When using *Fortran*, 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 Section 3. 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 *C11* program.

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

int main(void)
{
   static long lock = 0;
   static int count = 0;
   shmem_init();
   int me = shmem_my_pe();
   shmem_set_lock(&lock);
   int val = shmem_g(&count, 0); /* get count value on PE 0 */
   printf("%d: count is %d\n", me, val);
   val++; /* incrementing and updating count on PE 0 */
   shmem_p(&count, val, 0);
   shmem_quiet();
   shmem_clear_lock(&lock);
   shmem_finalize();
   return 0;
}
```

9.12 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 and may notify the user about deprecation of their use.

9.12.1 SHMEM_CACHE

Controls data cache utilities.

SYNOPSIS

— deprecation start —

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);
```

— deprecation end —

— deprecation start —

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)
```

— deprecation end —

DESCRIPTION

Arguments

IN	*dest*	A data object that is local to the PE. *dest* can be of any noncharacter type. When using *Fortran*, 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.

shmem_clear_cache_inv disables automatic cache coherency mode previously enabled by *shmem_set_cache _inv* or *shmem_set_cache_line_inv*.

shmem_udcflush makes the entire user data cache coherent.

shmem_udcflush_line makes coherent the cache line that corresponds with the address specified by *dest*.

Return Values

None.

Notes

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

EXAMPLES

None.

Annex A

Writing OpenSHMEM Programs

Incorporating OpenSHMEM into Programs

The following section describes how to write a “Hello World" OpenSHMEM program. To write a “Hello World" OpenSHMEM program, the user must:

- Include the header file *shmem.h* for *C* or *shmem.fh* for *Fortran*.
- Add the initialization call *shmem_init*.
- Use OpenSHMEM calls to query the local PE number (*shmem_my_pe*) and the total number of PEs (*shmem_n_pes*).
- Add the finalization call *shmem_finalize*.

In OpenSHMEM, the order in which lines appear in the output is not deterministic because PEs execute asynchronously in parallel.

Listing A.1: “Hello World” example program in *C*

```c
#include <stdio.h>
#include <shmem.h> /* The OpenSHMEM header file */

int main (void)
{
   shmem_init();
   int me = shmem_my_pe();
   int npes = shmem_n_pes();
   printf("Hello from %d of %d\n", me, npes);
   shmem_finalize();
   return 0;
}
```

Listing A.2: Possible ordering of expected output with 4 PEs from the program in Listing A.1

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

— deprecation start —

OpenSHMEM also provides a *Fortran* API. Listing A.3 shows a similar program written in *Fortran*.

Listing A.3: "Hello World" example program in *Fortran*

```
program hello

  include "shmem.fh"
  integer :: shmem_my_pe, shmem_n_pes

  integer :: npes, me

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

  write (*, 1000) me, npes

 1000 format ('Hello from', 1X, I4, 1X, 'of', 1X, I4)

end program hello
```

Listing A.4: Possible ordering of expected output with 4 PEs from the program in Listing A.3

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

— deprecation end —

The example in Listing A.5 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 *shmem_put*.

The *static* keyword makes the *dest* array symmetric on all PEs. Each PE is able to transfer data to a remote *dest* array by simply specifying to an OpenSHMEM routine such as *shmem_put* the local address of the symmetric data object that will receive the data. This local address resolution aids programmability because the address of the *dest* need not be exchanged with the active side (PE *0*) prior to the *Remote Memory Access* (RMA) routine.

Conversely, the declaration of the *short source* array is asymmetric (local only). The *source* object does not need to be symmetric because *Put* handles the references to the *source* array only on the active (local) side.

Listing A.5: Example program with symmetric data objects

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

#define SIZE 16

int main(void)
{
   short source[SIZE];
   static short dest[SIZE];
   static long lock = 0;
   shmem_init();
   int me = shmem_my_pe();
   int npes = shmem_n_pes();
   if (me == 0) {
      /* initialize array */
      for (int i = 0; i < SIZE; i++)
         source[i] = i;
      /* local, not symmetric */
      /* static makes it symmetric */
      /* put "size" words into dest on each PE */
      for (int i = 1; i < npes; i++)
         shmem_put(dest, source, SIZE, i);
   }
   shmem_barrier_all(); /* sync sender and receiver */
   if (me != 0) {
      shmem_set_lock(&lock);
      printf("dest on PE %d is \t", me);
      for (int i = 0; i < SIZE; i++)
         printf("%hd \t", dest[i]);
      printf("\n");
      shmem_clear_lock(&lock);
   }
   shmem_finalize();
   return 0;
}
```

Listing A.6: Possible ordering of expected output with 4 PEs from the program in Listing A.5

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

The OpenSHMEM Specification does not specify 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 may be called as follows:

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

Where the ⟨compiler options⟩ are options understood by the underlying *C* compiler called by **oshcc**.

Programs written in *C++*

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

```
oshc++ <compiler options> -o myprogram myprogram.cpp
```

Where the ⟨compiler options⟩ are options understood by the underlying *C++* compiler called by **oshc++**.

Programs written in *Fortran*

— deprecation start —

The OpenSHMEM Reference Implementation provides a wrapper program, named **oshfort**, to aid in the compilation of *Fortran* programs. The wrapper may 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**.

— deprecation end —

2 Running Programs

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

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

The arguments for **oshrun** are:

⟨runner 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.

Annex C

Undefined Behavior in OpenSHMEM

The OpenSHMEM Specification formalizes the expected behavior of its library routines. In cases where routines are improperly used or the input is not in accordance with the Specification, the behavior is undefined.

Inappropriate Usage	**Undefined Behavior**
Uninitialized library	If the OpenSHMEM library is not initialized, calls to non-initializing OpenSHMEM routines have undefined behavior. For example, an implementation may try to continue or may abort immediately upon an OpenSHMEM call into the uninitialized library.
Multiple calls to initialization routines	In an OpenSHMEM program where the initialization routines *shmem_init* or *shmem_init_thread* have already been called, any subsequent calls to these initialization routines result in undefined behavior.
Accessing non-existent PEs	If a communications routine accesses a non-existent PE, then the OpenSHMEM library may handle this situation in an implementation-defined way. For example, the library may report 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. For example, a *Put* to a non-symmetric variable may be trapped where possible and the library may abort the program. Another implementation may choose to continue execution with or without a warning.
Non-symmetric allocation of symmetric memory	The symmetric memory management routines are collectives. For example, all PEs in the program must call *shmem_malloc* with the same *size* argument. Program behavior after a mismatched *shmem_malloc* call is undefined.
Use of null pointers with non-zero *len* specified	In any OpenSHMEM routine that takes a pointer and *len* describing the number of elements in that pointer, a null pointer may not be given unless the corresponding *len* is also specified as zero. Otherwise, the resulting behavior is undefined. The following cases summarize this behavior: • *len* is 0, pointer is null: supported. • *len* is not 0, pointer is null: undefined behavior. • *len* is 0, pointer is non-null: supported. • *len* is not 0, pointer is non-null: supported.

Annex D

Interoperability with other Programming Models

1 MPI Interoperability

OpenSHMEM routines may be used in conjunction with MPI routines in the same program. For example, on Silicon Graphics International (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 indexing precludes use of OpenSHMEM routines between processes in different *MPI_COMM_WORLD*s. For example, MPI processes started using the *MPI_Comm_spawn* routine cannot use OpenSHMEM routines to communicate with their parent MPI processes.

On SGI systems where MPI jobs use *Transmission Control Protocol* (TCP)/sockets for inter-host communication, OpenSHMEM routines may be used to communicate with processes running on the same host. The *shmem_pe_accessible* routine should 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 may 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 (e.g., *shmem_malloc*, *shpalloc*) is symmetric across the same or different executable files. The *shmem_addr_accessible* routine should 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 and has been extensively used on a number of products since 1993, including the Cray T3D, Cray X1E, Cray XT3 and XT4, SGI Origin, SGI Altix, Quadrics-based clusters, and InfiniBand-based clusters.

- SHMEM Timeline
  - Cray SHMEM
    - SHMEM first introduced by Cray Research, Inc. in 1993 for Cray T3D
    - Cray was acquired by SGI in 1996
    - Cray was acquired by Tera in 2000 (MTA)
    - Platforms: Cray T3D, T3E, C90, J90, SV1, SV2, X1, X2, XE, XMT, XT
  - SGI SHMEM
    - SGI acquired 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 acquired by Rackable Systems in 2009
    - SGI and OSSS signed a SHMEM trademark licensing agreement in 2010
    - HPE acquired SGI in 2016

A listing of OpenSHMEM implementations can be found on http://www.openshmem.org/.

Annex F

OpenSHMEM Specification and Deprecated API

1 Overview

For the OpenSHMEM Specification, deprecation is the process of identifying API that is supported but no longer recommended for use by users. The deprecated API **must** be supported until clearly indicated as otherwise by the Specification. This chapter records the API or functionality that have been deprecated, the version of the OpenSHMEM Specification that effected the deprecation, and the most recent version of the OpenSHMEM Specification in which the feature was supported before removal.

Deprecated API	Deprecated Since	Last Version Supported	Replaced By
Header Directory: *mpp*	1.1	Current	(none)
C/C++: *start_pes*	1.2	Current	*shmem_init*
Fortran: *START_PES*	1.2	Current	*SHMEM_INIT*
Implicit finalization	1.2	Current	*shmem_finalize*
C/C++: *_my_pe*	1.2	Current	*shmem_my_pe*
C/C++: *_num_pes*	1.2	Current	*shmem_n_pes*
Fortran: *MY_PE*	1.2	Current	*SHMEM_MY_PE*
Fortran: *NUM_PES*	1.2	Current	*SHMEM_N_PES*
C/C++: *shmalloc*	1.2	Current	*shmem_malloc*
C/C++: *shfree*	1.2	Current	*shmem_free*
C/C++: *shrealloc*	1.2	Current	*shmem_realloc*
C/C++: *shmemalign*	1.2	Current	*shmem_align*
Fortran: *SHMEM_PUT*	1.2	Current	*SHMEM_PUT8* or *SHMEM_PUT64*
C/C++: *shmem_clear_cache_inv* *Fortran*: *SHMEM_CLEAR_CACHE_INV*	1.3	Current	(none)
C/C++: *shmem_clear_cache_line_inv*	1.3	Current	(none)
C/C++: *shmem_set_cache_inv* *Fortran*: *SHMEM_SET_CACHE_INV*	1.3	Current	(none)
C/C++: *shmem_set_cache_line_inv* *Fortran*: *SHMEM_SET_CACHE_LINE_INV*	1.3	Current	(none)
C/C++: *shmem_udcflush* *Fortran*: *SHMEM_UDCFLUSH*	1.3	Current	(none)
C/C++: *shmem_udcflush_line* *Fortran*: *SHMEM_UDCFLUSH_LINE*	1.3	Current	(none)
_SHMEM_SYNC_VALUE	1.3	Current	*SHMEM_SYNC_VALUE*
_SHMEM_BARRIER_SYNC_SIZE	1.3	Current	*SHMEM_BARRIER_SYNC_SIZE*
_SHMEM_BCAST_SYNC_SIZE	1.3	Current	*SHMEM_BCAST_SYNC_SIZE*
_SHMEM_COLLECT_SYNC_SIZE	1.3	Current	*SHMEM_COLLECT_SYNC_SIZE*
_SHMEM_REDUCE_SYNC_SIZE	1.3	Current	*SHMEM_REDUCE_SYNC_SIZE*
_SHMEM_REDUCE_MIN_WRKDATA_SIZE	1.3	Current	*SHMEM_REDUCE_MIN_WRKDATA_SIZE*
_SHMEM_MAJOR_VERSION	1.3	Current	*SHMEM_MAJOR_VERSION*
_SHMEM_MINOR_VERSION	1.3	Current	*SHMEM_MINOR_VERSION*
_SHMEM_MAX_NAME_LEN	1.3	Current	*SHMEM_MAX_NAME_LEN*
_SHMEM_VENDOR_STRING	1.3	Current	*SHMEM_VENDOR_STRING*
_SHMEM_CMP_EQ	1.3	Current	*SHMEM_CMP_EQ*
_SHMEM_CMP_NE	1.3	Current	*SHMEM_CMP_NE*
_SHMEM_CMP_LT	1.3	Current	*SHMEM_CMP_LT*
_SHMEM_CMP_LE	1.3	Current	*SHMEM_CMP_LE*

Deprecated API	Deprecated Since	Last Version Supported	Replaced By
_SHMEM_CMP_GT	1.3	Current	*SHMEM_CMP_GT*
_SHMEM_CMP_GE	1.3	Current	*SHMEM_CMP_GE*
SMA_VERSION	1.4	Current	*SHMEM_VERSION*
SMA_INFO	1.4	Current	*SHMEM_INFO*
SMA_SYMMETRIC_SIZE	1.4	Current	*SHMEM_SYMMETRIC_SIZE*
SMA_DEBUG	1.4	Current	*SHMEM_DEBUG*
C/C++: *shmem_wait* *C/C++*: *shmem_***<TYPENAME>***_wait*	1.4	Current	See **Notes** for *shmem_wait_until*
C/C++: *shmem_wait_until*	1.4	Current	*C11*: *shmem_wait_until*, *C/C++*: *shmem_long_wait_until*
C11: *shmem_fetch* *C/C++*: *shmem_***<TYPENAME>***_fetch*	1.4	Current	*shmem_atomic_fetch*
C11: *shmem_set* *C/C++*: *shmem_***<TYPENAME>***_set*	1.4	Current	*shmem_atomic_set*
C11: *shmem_cswap* *C/C++*: *shmem_***<TYPENAME>***_cswap*	1.4	Current	*shmem_atomic_compare_swap*
C11: *shmem_swap* *C/C++*: *shmem_***<TYPENAME>***_swap*	1.4	Current	*shmem_atomic_swap*
C11: *shmem_finc* *C/C++*: *shmem_***<TYPENAME>***_finc*	1.4	Current	*shmem_atomic_fetch_inc*
C11: *shmem_inc* *C/C++*: *shmem_***<TYPENAME>***_inc*	1.4	Current	*shmem_atomic_inc*
C11: *shmem_fadd* *C/C++*: *shmem_***<TYPENAME>***_fadd*	1.4	Current	*shmem_atomic_fetch_add*
C11: *shmem_add* *C/C++*: *shmem_***<TYPENAME>***_add*	1.4	Current	*shmem_atomic_add*
Entire *Fortran* API	1.4	Current	(none)

2 Deprecation Rationale

2.1 Header Directory: *mpp*

In addition to the default system header paths, OpenSHMEM implementations must provide all OpenSHMEM-specified header files from the *mpp* header directory such that these headers can be referenced in *C/C++* as

```
#include <mpp/shmem.h>
#include <mpp/shmemx.h>
```

and in *Fortran* as

```
include 'mpp/shmem.fh'
include 'mpp/shmemx.fh'
```

for backwards compatibility with SGI SHMEM.

2.2 *C/C++*: *start_pes*

The *C/C++* routine *start_pes* includes an unnecessary initialization argument that is remnant of historical *SHMEM* implementations and no longer reflects the requirements of modern OpenSHMEM implementations. Furthermore, the naming of *start_pes* does not include the standardized *shmem_* naming prefix. This routine has been deprecated and OpenSHMEM users are encouraged to use *shmem_init* instead.

2.3 Implicit Finalization

Implicit finalization was deprecated and replaced with explicit finalization using the *shmem_finalize* routine. Explicit finalization improves portability and also improves interoperability with profiling and debugging tools.

2.4 *C/C++*: *_my_pe*, *_num_pes*, *shmalloc*, *shfree*, *shrealloc*, *shmemalign*

The *C/C++* routines *_my_pe*, *_num_pes*, *shmalloc*, *shfree*, *shrealloc*, and *shmemalign* were deprecated in order to normalize the OpenSHMEM API to use *shmem_* as the standard prefix for all routines.

2.5 *Fortran*: *START_PES*, *MY_PE*, *NUM_PES*

The *Fortran* routines *START_PES*, *MY_PE*, and *NUM_PES* were deprecated in order to minimize the API differences from the deprecation of *C/C++* routines *start_pes*, *_my_pe*, and *_num_pes*.

2.6 *Fortran*: *SHMEM_PUT*

The *Fortran* routine *SHMEM_PUT* is defined only for the *Fortran* API and is semantically identical to *Fortran* routines *SHMEM_PUT8* and *SHMEM_PUT64*. Since *SHMEM_PUT8* and *SHMEM_PUT64* have defined equivalents in the *C/C++* interface, *SHMEM_PUT* is ambiguous and has been deprecated.

2.7 SHMEM_CACHE

The *SHMEM_CACHE* API

C/C++:	*Fortran*:
shmem_clear_cache_inv	*SHMEM_CLEAR_CACHE_INV*
shmem_set_cache_inv	*SHMEM_SET_CACHE_INV*
shmem_set_cache_line_inv	*SHMEM_SET_CACHE_LINE_INV*
shmem_udcflush	*SHMEM_UDCFLUSH*
shmem_udcflush_line	*SHMEM_UDCFLUSH_LINE*
shmem_clear_cache_line_inv	

was originally implemented for systems with cache-management instructions. This API has largely gone unused on cache-coherent system architectures. *SHMEM_CACHE* has been deprecated.

2.8 *_SHMEM_** Library Constants

The library constants

_SHMEM_SYNC_VALUE	*_SHMEM_MAX_NAME_LEN*
_SHMEM_BARRIER_SYNC_SIZE	*_SHMEM_VENDOR_STRING*
_SHMEM_BCAST_SYNC_SIZE	*_SHMEM_CMP_EQ*
_SHMEM_COLLECT_SYNC_SIZE	*_SHMEM_CMP_NE*
_SHMEM_REDUCE_SYNC_SIZE	*_SHMEM_CMP_LT*
_SHMEM_REDUCE_MIN_WRKDATA_SIZE	*_SHMEM_CMP_LE*
_SHMEM_MAJOR_VERSION	*_SHMEM_CMP_GT*
_SHMEM_MINOR_VERSION	*_SHMEM_CMP_GE*

do not adhere to the *C* standard's reserved identifiers and the *C++* standard's reserved names. These constants were deprecated and replaced with corresponding constants of prefix *SHMEM_* that adhere to *C/C++* and *Fortran* naming conventions.

2.9 *SMA_** Environment Variables

The environment variables *SMA_VERSION*, *SMA_INFO*, *SMA_SYMMETRIC_SIZE*, and *SMA_DEBUG* were deprecated in order to normalize the OpenSHMEM API to use *SHMEM_* as the standard prefix for all environment variables.

2.10 *C/C++*: *shmem_wait*

The *C/C++* interface for *shmem_wait* and *shmem_<**TYPENAME**>_wait* was identified as unintuitive with respect to the comparison operation it performed. As *shmem_wait* can be trivially replaced by *shmem_wait_until* where *cmp* is *SHMEM_CMP_NE*, the *shmem_wait* interface was deprecated in favor of *shmem_wait_until*, which makes the comparison operation explicit and better communicates the developer's intent.

2.11 *C/C++*: *shmem_wait_until*

The *long*-typed *C/C++* routine *shmem_wait_until* was deprecated in favor of the *C11* type-generic interface of the same name or the explicitly typed *C/C++* routine *shmem_long_wait_until*.

2.12 *C11* and *C/C++*: *shmem_fetch*, *shmem_set*, *shmem_cswap*, *shmem_swap*, *shmem_finc*, *shmem_inc*, *shmem_fadd*, *shmem_add*

The *C11* and *C/C++* interfaces for

C11:	*C/C++*:
shmem_fetch	*shmem_**<TYPENAME>**_fetch*
shmem_set	*shmem_**<TYPENAME>**_set*
shmem_cswap	*shmem_**<TYPENAME>**_cswap*
shmem_swap	*shmem_**<TYPENAME>**_swap*
shmem_finc	*shmem_**<TYPENAME>**_finc*
shmem_inc	*shmem_**<TYPENAME>**_inc*
shmem_fadd	*shmem_**<TYPENAME>**_fadd*
shmem_add	*shmem_**<TYPENAME>**_add*

were deprecated and replaced with similarly named interfaces within the *shmem_atomic_** namespace in order to more clearly identify these calls as performing atomic operations. In addition, the abbreviated names “cswap”, “finc”, and “fadd” were expanded for clarity to “compare_swap”, “fetch_inc”, and “fetch_add”.

2.13 *Fortran* API

The entire OpenSHMEM *Fortran* API was deprecated because of a general lack of use and a lack of conformance with legacy *Fortran* standards. In lieu of an extensive update of the *Fortran* API, *Fortran* users are encouraged to leverage the OpenSHMEM Specification’s *C* API through the *Fortran–C* interoperability initially standardized by *Fortran 2003*[1].

[1] Formally, *Fortran 2003* is known as ISO/IEC 1539-1:2004(E).

Annex G

Changes to this Document

1 Version 1.4

Major changes in OpenSHMEM 1.4 include multithreading support, *contexts* for communication management, *shmem_sync*, *shmem_calloc*, expanded type support, a new namespace for atomic operations, atomic bitwise operations, *shmem_test* for nonblocking point-to-point synchronization, and *C11* type-generic interfaces for point-to-point synchronization.

The following list describes the specific changes in OpenSHMEM 1.4:

- New communication management API, including *shmem_ctx_create*; *shmem_ctx_destroy*; and additional RMA, AMO, and memory ordering routines that accept *shmem_ctx_t* arguments.
  See Section 9.4.
- New API *shmem_sync_all* and *shmem_sync* to provide PE synchronization without completing pending communication operations.
  See Sections 9.8.3 and 9.8.4.
- Clarified that the OpenSHMEM extensions header files are required, even when empty.
  See Section 5.
- Clarified that the *SHMEM_GET64* and *SHMEM_GET64_NBI* routines are included in the *Fortran* language bindings.
  See Sections 9.5.4 and 9.6.2.
- Clarified that *shmem_init* must be matched with a call to *shmem_finalize*.
  See Sections 9.1.1 and 9.1.4.
- Added the *SHMEM_SYNC_SIZE* constant.
  See Section 6.
- Added type-generic interfaces for *shmem_wait_until*.
  See Section 9.9.1.
- Removed the *volatile* qualifiers from the *ivar* arguments to *shmem_wait* routines and the *lock* arguments in the lock API. *Rationale: Volatile qualifiers were added to several API routines in OpenSHMEM 1.3; however, they were later found to be unnecessary.*
  See Sections 9.9.1 and 9.11.1.
- Deprecated the *SMA_** environment variables and added equivalent *SHMEM_** environment variables.
  See Section 8.
- Added the *C11 _Noreturn* function specifier to *shmem_global_exit*.
  See Section 9.1.5.

- Clarified ordering semantics of memory ordering, point-to-point synchronization, and collective synchronization routines.
- Clarified deprecation overview and added deprecation rationale in Annex F.
  See Section F.
- Deprecated header directory *mpp*.
  See Section F.
- Deprecated the *shmem_wait* functions and the *long*-typed *C/C++ shmem_wait_until* function.
  See Section 9.9.
- Added the *shmem_test* functions.
  See Section 9.9.
- Added the *shmem_calloc* function.
  See Section 9.3.2.
- Introduced the thread safe semantics that define the interaction between OpenSHMEM routines and user threads.
  See Section 9.2.
- Added the new routine *shmem_init_thread* to initialize the OpenSHMEM library with one of the defined thread levels.
  See Section 9.2.1.
- Added the new routine *shmem_query_thread* to query the thread level provided by the OpenSHMEM implementation.
  See Section 9.2.2.
- Clarified the semantics of *shmem_quiet* for a multithreaded OpenSHMEM PE.
  See Section 9.10.2
- Revised the description of *shmem_barrier_all* for a multithreaded OpenSHMEM PE.
  See Section 9.8.1
- Revised the description of *shmem_wait* for a multithreaded OpenSHMEM PE.
  See Section 9.9.1
- Clarified description for *SHMEM_VENDOR_STRING*.
  See Section 6.
- Clarified description for *SHMEM_MAX_NAME_LEN*.
  See Section 6.
- Clarified API description for *shmem_info_get_name*.
  See Section 9.1.10.
- Expanded the type support for RMA, AMO, and point-to-point synchronization operations.
  See Tables 3, 4, 5, and 7
- Renamed AMO operations to use *shmem_atomic_** prefix and deprecated old AMO routines.
  See Section 9.7.
- Added fetching and non-fetching bitwise AND, OR, and XOR atomic operations.
  See Section 9.7.
- Deprecated the entire *Fortran* API.
- Replaced the *complex* macro in complex-typed reductions with the *C99* (and later) type specifier *_Complex* to remove an implicit dependence on *complex.h*.
  See Section 9.8.7.
- Clarified that complex-typed reductions in C are optionally supported.
  See Section 9.8.7.

2 Version 1.3

Major changes in OpenSHMEM 1.3 include the addition of nonblocking RMA operations, atomic *Put* and *Get* operations, all-to-all collectives, and *C11* type-generic interfaces for RMA and AMO operations.

The following list describes the specific changes in OpenSHMEM 1.3:

- Clarified implementation of PEs as threads.
- Added *const* to every read-only pointer argument.
- Clarified definition of *Fence*.
  See Section 2.
- Clarified implementation of symmetric memory allocation.
  See Section 3.
- Restricted atomic operation guarantees to other atomic operations with the same datatype.
  See Section 3.1.
- Deprecation of all constants that start with *_SHMEM_**.
  See Section 6.
- Added a type-generic interface to OpenSHMEM RMA and AMO operations based on *C11* Generics.
  See Sections 9.5, 9.6 and 9.7.
- New nonblocking variants of remote memory access, *SHMEM_PUT_NBI* and *SHMEM_GET_NBI*.
  See Sections 9.6.1 and 9.6.2.
- New atomic elemental read and write operations, *SHMEM_FETCH* and *SHMEM_SET*.
  See Sections 9.7.1 and 9.7.2
- New alltoall data exchange operations, *SHMEM_ALLTOALL* and *SHMEM_ALLTOALLS*.
  See Sections 9.8.8 and 9.8.9.
- Added *volatile* to remotely accessible pointer argument in *SHMEM_WAIT* and *SHMEM_LOCK*.
  See Sections 9.9.1 and 9.11.1.
- Deprecation of *SHMEM_CACHE*.
  See Section 9.12.1.

3 Version 1.2

Major changes in OpenSHMEM 1.2 include a new initialization routine (*shmem_init*), improvements to the execution model with an explicit library-finalization routine (*shmem_finalize*), an early-exit routine (*shmem_global_exit*), namespace standardization, and clarifications to several API descriptions.

The following list describes the specific changes in OpenSHMEM 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*'s *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 9.1.1.

- Deprecation of *_my_pe* and *_num_pes* routines.
  See Sections 9.1.2 and 9.1.3.

- New API *shmem_finalize* to provide collective mechanism to cleanly exit an OpenSHMEM program and release resources.
  See Section 9.1.4.

- New API *shmem_global_exit* to provide mechanism to exit an OpenSHMEM program.
  See Section 9.1.5.

- Clarification related to the address of the referenced object in *shmem_ptr*.
  See Section 9.1.8.

- New API to query the version and name information.
  See Section 9.1.9 and 9.1.10.

- OpenSHMEM library API normalization. All *C* symmetric memory management API begins with *shmem_*.
  See Section 9.3.1.

- Notes and clarifications added to *shmem_malloc*.
  See Section 9.3.1.

- Deprecation of *Fortran* API routine *SHMEM_PUT*.
  See Section 9.5.1.

- Clarification related to *shmem_wait*.
  See Section 9.9.1.

- Undefined behavior for null pointers without zero counts added.
  See Annex C

- Addition of new Annex for clearly specifying deprecated API and its support across versions of the OpenSHMEM Specification.
  See Annex F.

4 Version 1.1

Major changes from OpenSHMEM 1.0 to OpenSHMEM 1.1 include the introduction of the *shmemx.h* header file for non-standard API extensions, clarifications to completion semantics and API descriptions in agreement with the SGI SHMEM specification, and general readabilty and usability improvements to the document structure.

The following list describes the specific changes in OpenSHMEM 1.1:

- Clarifications of the completion semantics of memory synchronization interfaces.
  See Section 9.10.

- Clarification of the completion semantics of memory load and store operations in context of *shmem_barrier_all* and *shmem_barrier* routines.
  See Section 9.8.1 and 9.8.2.

- Clarification of the completion and ordering semantics of *shmem_quiet* and *shmem_fence*.
  See Section 9.10.2 and 9.10.1.

- Clarifications of the completion semantics of RMA and AMO routines.
  See Sections 9.5 and 9.7

- 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 9.1.6 and 9.1.7.
- 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 9.9.1.
- Added API calls: *shmem_char_p*, *shmem_char_g*.
  See Sections 9.5.2 and 9.5.5.
- Removed API calls: *shmem_char_put*, *shmem_char_get*.
  See Sections 9.5.1 and 9.5.4.
- The usage of *ptrdiff_t*, *size_t*, and *int* in the interface signature was made consistent with the description.
  See Sections 9.8, 9.5.3, and 9.5.6.
- Revised *shmem_barrier* example.
  See Section 9.8.2.
- Clarification of the initial value of *pSync* work arrays for *shmem_barrier*.
  See Section 9.8.2.
- Clarification of the expected behavior when multiple *start_pes* calls are encountered.
  See Section 9.1.11.
- Corrected the definition of atomic increment operation.
  See Section 9.7.6.
- Clarification of the size of the symmetric heap and when it is set.
  See Section 9.3.1.
- Clarification of the integer and real sizes for *Fortran* API.
  See Sections 9.7.8, 9.7.3, 9.7.4, 9.7.5, 9.7.6, and 9.7.7.
- Clarification of the expected behavior on program *exit*.
  See Section 4, Execution Model.
- More detailed description for the progress of OpenSHMEM operations provided.
  See Section 4.1.
- Clarification of naming convention for non-standard interfaces and their inclusion in *shmemx.h*.
  See Section 5.
- Various fixes to OpenSHMEM code examples across the Specification to include appropriate header files.
- Removing requirement that implementations should detect size mismatch and return error information for *shmalloc* and ensuring consistent language.
  See Sections 9.3.1 and Annex C.
- *Fortran* programming fixes for examples.
  See Sections 9.8.7 and 9.9.1.

- Clarifications of the reuse *pSync* and *pWork* across collectives.
  See Sections 9.8, 9.8.5, 9.8.6 and 9.8.7.

- Name changes for UV and ICE for SGI systems.
  See Annex E.

Index