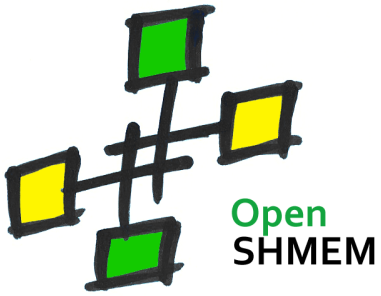
OpenSHMEM

Application Programming Interface



http://www.openshmem.org/

Version 1.5

13th November 2019

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
- Megan Grodowitz, Arm Inc.
- 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*. 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.

(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.

(d) *Wait and Test*: A PE calling a point-to-point synchronization routine ensures the value of a local symmetric object meets a specified condition. Wait operations block until the specified condition is met, whereas test operations return immediately and indicate whether or not the specified condition is met.

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

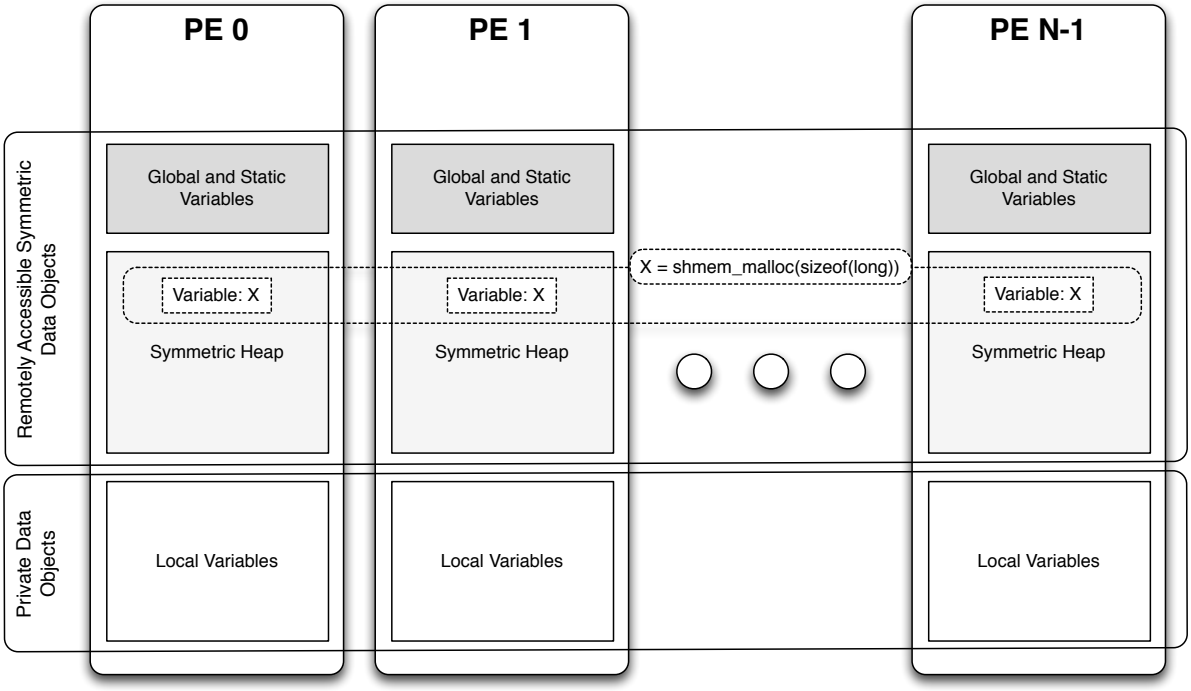


Figure 1: OpenSHMEM Memory Model

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*. 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[2]. (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 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:

- Global and static *C* and *C++* variables. These data objects must not be defined in a dynamic shared object (DSO).
- *C* and *C++* data allocated by OpenSHMEM memory management routines (Section 9.3)

OpenSHMEM dynamic memory allocation routines (e.g., *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 perform atomic operations on symmetric data objects, which are defined in Section 9.8. The atomic routines guarantee that concurrent accesses by any of these routines to the same location and using the same datatype (specified in Tables 5 and 6) will be exclusive. Exclusivity is also guaranteed when the target PE performs a wait or test operation on the same location and with the same datatype as one or more atomic operations. OpenSHMEM atomic operations do not guarantee exclusivity in the following scenarios, all of which result in undefined behavior.

1. When concurrent accesses to the same location are performed using OpenSHMEM atomic operations using different datatypes.
2. When atomic and non-atomic OpenSHMEM operations are used to access the same location concurrently.
3. When OpenSHMEM atomic operations and non-OpenSHMEM operations (e.g. load and store operations) are used to access the same location concurrently.

For example, during the execution of an atomic remote integer increment, i.e. *shmem_atomic_inc*, 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.

The following *C/C++* example illustrates scenario 1. In this example, different datatypes are used to access the same location concurrently, resulting in undefined behavior. The undefined behavior can be resolved by using the same datatype in all concurrent operations. For example, the 32-bit value can be left-shifted and a 64-bit atomic OR operation can be used.

```
#include <shmem.h>

int main(void) {
    static uint64_t x = 0;

    shmem_init();
    /* Undefined behavior: The following AMOs access the same location concurrently using
     * different types. */
    if (shmem_my_pe() > 0)
      shmem_uint32_atomic_or((uint32_t*)&x, shmem_my_pe()+1, 0);
    else
      shmem_uint64_atomic_or(&x, shmem_my_pe()+1, 0);
```

[2]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

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

The following *C/C++* example illustrates scenario 2. In this example, atomic increment operations are concurrent with a non-atomic reduction operation, resulting in undefined behavior. The undefined behavior can be resolved by inserting a barrier operation before the reduction. The barrier ensures that all local and remote AMOs have completed before the reduction operation accesses *x*.

```
#include <shmem.h>

int main(void) {
    static long psync[SHMEM_REDUCE_SYNC_SIZE];
    static int pwrk[SHMEM_REDUCE_MIN_WRKDATA_SIZE];
    static int x = 0, y = 0;

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

    shmem_init();
    shmem_int_atomic_inc(&x, (shmem_my_pe()+1) % shmem_n_pes());
    /* Undefined behavior: The following reduction operation performs accesses to symmetric
     * variable 'x' that are concurrent with previously issued atomic increment operations
     * on the same variable. */
    shmem_int_sum_to_all(&y, &x, 1, 0, 0, shmem_n_pes(), pwrk, psync);

    shmem_finalize();
    return 0;
}
```

The following *C/C++* example illustrates scenario 3. In this example, an OpenSHMEM atomic increment operation is concurrent with a local increment operation, resulting in undefined behavior. The undefined behavior can be resolved by replacing the local increment operation with an OpenSHMEM atomic increment.

```
#include <shmem.h>

int main(void) {
    static int x = 0;

    shmem_init();
    /* Undefined behavior: OpenSHMEM atomic increment operations are concurrent with the local
     * increment of symmetric variable 'x'. */
    if (shmem_my_pe() > 0)
      shmem_int_atomic_inc(&x, 0);
    else
      x++;

    shmem_finalize();
    return 0;
}
```

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*[3] 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

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

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*. The identifiers are fixed for the life of the OpenSHMEM program.

4.1 Progress of OpenSHMEM Operations

The OpenSHMEM model assumes that computation and communication are naturally overlapped. OpenSHMEM programs are expected to exhibit progression of communication both with and without OpenSHMEM calls. Consider a PE that is engaged in a computation with no OpenSHMEM calls. Other PEs should be able to communicate (*put*, *get*, *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* language bindings. Any implementation that provides *C* bindings can claim conformance to the specification. The OpenSHMEM header file *shmem.h* for *C* 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_*, *_SHMEM_*, or with OpenSHMEM API names.

All OpenSHMEM extension APIs that are not part of this specification must be defined in the *shmemx.h* include file for language bindings. This header file 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.
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_TEAM_NUM_CONTEXTS*	The bitwise flag which specifies that a team creation routine should use the *num_contexts* member of the provided *shmem_team_config_t* configuration parameter as a request. See Sections 9.4.3 and 9.4.6 for more detail about its use.
C/C++: *SHMEM_TEAM_INVALID*	A value corresponding to an invalid team. This value can be used to initialize or update team handles to indicate that they do not reference a valid team. When managed in this way, applications can use an equality comparison to test whether a given team handle references a valid team. See Section 9.4 for more detail about its use.
C/C++: *SHMEM_CTX_INVALID*	A value corresponding to an invalid communication context. This value can be used to initialize or update context handles to indicate that they do not reference a valid context. When managed in this way, applications can use an equality comparison to test whether a given context handle references a valid context. See Section 9.5 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 9.5.1 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 9.5.1 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 9.5.1 for more detail about its use.
C/C++: *SHMEM_SIGNAL_SET*	An integer constant expression corresponding to the signal update set operation. See Section 9.10.3 and Section 9.10.4 for more detail about its use.
C/C++: *SHMEM_SIGNAL_ADD*	An integer constant expression corresponding to the signal update add operation. See Section 9.10.3 and Section 9.10.4 for more detail about its use.

Constant	Description
C/C++: *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 9.11 for more detail about its use.
C/C++: *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 9.11 for more detail about its use.
C/C++: *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 9.11.5 for more detail about its use.
C/C++: *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 9.11.7 for more detail about its use.
C/C++: *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 9.11.2 for more detail about its use.

Constant	Description
C/C++: *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 9.11.6 for more detail about its use.
C/C++: *SHMEM_ALLTOALL_SYNC_SIZE* — deprecation start — — deprecation end —	Length of the work array needed for *shmem_alltoall* routines. The value of this constant is implementation specific. See Section 9.11.8 for more detail about its use.
C/C++: *SHMEM_ALLTOALLS_SYNC_SIZE* — deprecation start — — deprecation end —	Length of the work array needed for *shmem_alltoalls* routines. The value of this constant is implementation specific. See Section 9.11.9 for more detail about its use.
C/C++: *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.
C/C++: *SHMEM_MAJOR_VERSION* — deprecation start — *C/C++*: *_SHMEM_MAJOR_VERSION* — deprecation end —	Integer representing the major version of OpenSHMEM Specification in use.
C/C++: *SHMEM_MINOR_VERSION* — deprecation start — *C/C++*: *_SHMEM_MINOR_VERSION* — deprecation end —	Integer representing the minor version of OpenSHMEM Specification in use.

Constant	Description
C/C++: *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++: *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.
C/C++: *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 9.12 for more detail about its use.
C/C++: *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 9.12 for more detail about its use.
C/C++: *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 9.12 for more detail about its use.

Constant	Description
C/C++: *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 9.12 for more detail about its use.
C/C++: *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 9.12 for more detail about its use.
C/C++: *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 9.12 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_TEAM_WORLD*	Handle of type *shmem_team_t* that corresponds to the default team of all PEs in the OpenSHMEM program. All point-to-point communication operations and collective synchronizations that do not specify a team are performed on the default team. See Section 9.4 for more detail about its use.
C/C++: *SHMEM_TEAM_SHARED*	Handle of type *shmem_team_t* that corresponds to a team of PEs that share a memory domain. When this handle is used by some PE, it will refer to the team of all PEs that would return a non-null pointer from *shmem_ptr* for symmetric objects on that PE, and vice versa. This means that symmetric objects on each PE are directly load/store accessible by all PEs in the team. See Section 9.4 for more detail about its use.

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 9.5 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 or floating point value with an optional character suffix	Specifies the size (in bytes) of the symmetric heap memory per PE. The resulting size is implementation-defined and must be least as large as the integer ceiling of the product of the numeric prefix and the scaling factor. The allowed character suffixes for the scaling factor are as follows: • k or K multiplies by 2^{10} (kibibytes) • m or M multiplies by 2^{20} (mebibytes) • g or G multiplies by 2^{30} (gibibytes) • t or T multiplies by 2^{40} (tebibytes) For example, string “20m” is equivalent to the integer value 20971520, or 20 mebibytes. Similarly the string “3.1m” is equivalent to the integer value 3250586. Only one multiplier is recognized and any characters following the multiplier are ignored, so “20kk” will not produce the same result as “20m”. Usage of string “.5m” will yield the same result as the string “0.5m”. An invalid value for SHMEM_SYMMETRIC_SIZE is an error, which the OpenSHMEM library shall report by either returning a nonzero value from *shmem_init_thread* or causing program termination.
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);
```

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);
```

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);
```

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);
```

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 teams created by the OpenSHMEM program. As a result, all shareable contexts are destroyed. 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);
```

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++* 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++*, 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++* 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

```
#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++:

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

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

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

Notes

This routine may be particularly useful for hybrid programming with other communication libraries (such as *Message Passing Interface* (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);
```

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

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*.

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 *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., using *shmem_malloc*) 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);
```

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.

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

In the following *C11* example, PE 0 uses the *shmem_ptr* routine to query a pointer and directly access the *dest* array on PE 1:

```
#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);
```

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);
```

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. 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 —

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.

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*	Pointer 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 in the default team.

The *shmem_malloc* routine returns a pointer to a block of at least *size* bytes, which shall be suitably aligned so that it may be assigned to a pointer to any type of object. This space is allocated from the symmetric heap (in contrast to *malloc*, which allocates from the private heap). When *size* is zero, the *shmem_malloc* routine performs no action and returns a null pointer.

The *shmem_align* routine allocates a block in the symmetric heap that has a byte alignment specified by the *alignment* argument. The value of *alignment* shall be a multiple of *sizeof(void *)* that is also a power of two. Otherwise, the behavior is undefined. When *size* is zero, the *shmem_align* routine performs no action and returns a null pointer.

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 is performed.

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. When no action is performed, these routines return without performing a barrier. Otherwise, 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, respectively. 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, all PEs return a null pointer.

The *shmem_align* routine returns an aligned pointer whose value is a multiple of *alignment*; 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).

When the *ptr* argument in a call to *shmem_realloc* corresponds to a buffer allocated using *shmem_align*, the buffer returned by *shmem_realloc* is not guaranteed to maintain the alignment requested in the original call to *shmem_align*.

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 on the default team 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.

When *count* or *size* is *0*, the *shmem_calloc* routine returns without performing a barrier. Otherwise, this 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.

9.4 Team Management Routines

The PEs in an OpenSHMEM program communicate using either point-to-point routines—such as *Remote Memory Access* (RMA) and *Atomic Memory Operation* (AMO) routines—which specify the PE number of the target PE, or collective routines, which operate over a set of PEs. In OpenSHMEM, teams allow programs to group a set of PEs for communication. Team-based collective communications operate across all the PEs in a valid team. Point-to-point communication can make use of team-relative PE numbering through team-based contexts (see Section 9.5) or PE number translation.

Predefined and Program-Defined Teams

An OpenSHMEM team may be predefined (i.e., provided by the OpenSHMEM library) or defined by the OpenSHMEM program. A program-defined team is created by "splitting" a parent team into one or more new teams—each with some subset of PEs of the parent team—via one of the *shmem_team_split_** routines.

All predefined teams are valid for the duration of the OpenSHMEM portion of an application. Any team successfully created by a *shmem_team_split_** routine is valid until it is destroyed. All valid teams have a least one member.

Team Handles

A "team handle" is an opaque object with type *shmem_team_t* that is used to reference a team. Team handles are not remotely accessible objects. The predefined teams may be accessed via the team handles listed in Section 7.

OpenSHMEM communication routines that do not accept a team handle argument operate on the default team, which may be accessed through the *SHMEM_TEAM_WORLD* handle. The default team encompasses the set of all PEs in the OpenSHMEM program, and a PE number in the default team is the same as the value returned by *shmem_my_pe*.

A team handle may be initialized to or assigned the value *SHMEM_TEAM_INVALID* to indicate that handle does not reference a valid team. When managed in this way, applications can use an equality comparison to test whether a given team handle references a valid team.

Thread Safety

When it is allowed by the threading model provided by the OpenSHMEM library, a team may be used concurrently in non-collective operations (e.g., *shmem_team_my_pe*) by multiple threads within the PE where it was created. For collective operations, a team may not be used concurrently by multiple threads in the same PE.

Collective Ordering

In OpenSHMEM, a team object encapsulates resources used to communicate between PEs in collective operations. When calling multiple subsequent collective operations on a team, the collective operations—along with any relevant team based resources—are matched across the PEs in the team based on ordering of collective routine calls. It is the responsibility of the OpenSHMEM program to ensure the same ordering of collective routine calls across all PEs in a team.

A full discussion of collective semantics follows in Section 9.11.

Team Creation

Team creation is a collective operation on the parent team object. New teams result from a *shmem_team_split_** routine, which takes a parent team and other arguments and produces new teams that are a subset of the parent team. All PEs in a parent team must participate in a split operation to create new teams. If a PE from the parent team is not a member of any resulting new teams, it will receive a value of *SHMEM_TEAM_INVALID* as the value for the new team handle.

Teams that are created by a *shmem_team_split_** routine may be provided a configuration argument that specifies attributes of each new team. This configuration argument is of type *shmem_team_config_t*, which is detailed further in Section 9.4.3.

PEs in a newly created teams are consecutively numbered with starting with PE number 0. PEs are always ordered by the existing global PE number that would be returned by the *shmem_my_pe* routine. Team relative PE numbers can be used for point-to-point operations through team-based contexts (see Section 9.5) or using the translation routine *shmem_team_translate_pe*.

As with any collective routine on a team, the program must ensure that there are no simultaneous split operations occurring on the same parent team on a given PE, i.e. in separate threads.

As with any collective routine on a team, team creation is matched across PEs based on ordering. So, team creation events must occur in the same order on all PEs in the parent team.

Upon completion of a team creation operation, the parent and any resulting child teams will be immediately usable for any team-based operations, including creating new child teams, without any intervening synchronization.

9.4.1 SHMEM_TEAM_MY_PE

Returns the number of the calling PE within a specified team.

SYNOPSIS

C/C++:

```
int shmem_team_my_pe(shmem_team_t team);
```

DESCRIPTION

Arguments

IN	*team*	An OpenSHMEM team handle.

API description

When *team* specifies a valid team, the *shmem_team_my_pe* routine returns the number of the calling PE within the specified team. The number is an integer between 0 and $N-1$ for a team containing N PEs. Each member of the team has a unique number.

If *team* compares equal to *SHMEM_TEAM_INVALID*, then the value *-1* is returned. If *team* is otherwise invalid, the behavior is undefined.

Return Values

The number of the calling PE within the specified team, or the value *-1* if the team handle compares equal to *SHMEM_TEAM_INVALID*.

Notes

For the default team, this routine will return the same value as *shmem_my_pe*.

9.4.2 SHMEM_TEAM_N_PES

Returns the number of PEs in a specified team.

SYNOPSIS

C/C++:

```
int shmem_team_n_pes(shmem_team_t team);
```

DESCRIPTION

Arguments

IN	*team*	An OpenSHMEM team handle.

API description

When *team* specifies a valid team, the *shmem_team_n_pes* routine returns the number of PEs in the team. This will always be a value between 1 and *N*, where *N* is the total number of PEs running in the OpenSHMEM program.

If *team* compares equal to *SHMEM_TEAM_INVALID*, then the value *-1* is returned. If *team* is otherwise invalid, the behavior is undefined.

Return Values

The number of PEs in the specified team, or the value *-1* if the team handle compares equal to *SHMEM_TEAM_INVALID*.

Notes

For the default team, this routine will return the same value as *shmem_n_pes*.

9.4.3 SHMEM_TEAM_CONFIG_T

A structure type representing team configuration arguments

SYNOPSIS

C/C++:

```
typedef struct {
  int num_contexts;
} shmem_team_config_t;
```

DESCRIPTION

Arguments

None.

API description

A team configuration argument acts as an input *shmem_team_split_** routines. It specifies the requested capabilities of the team to be created.

The *num_contexts* member specifies the total number of simultaneously existing contexts that the program requests to create from this team. These contexts may be created in any number of threads. Successful creation of a team configured with *num_contexts* of *N* means that the implementation will make a best effort to reserve enough resources to allow the team to have *N* contexts created from the team in existance at any given time. It is not a guaruntee that *N* calls to *shmem_team_create_ctx* will succeed. See Section 9.5 for more on communication contexts and Section 9.5.2 for team-based context creation.

When using the configuration structure to create teams, a mask parameter controls which fields to use. Any configuration parameter value that is not indicated in the mask will be ignored, and the default value will be used instead. Therefore, a program does not have to set all fields in the config struct; only those for which it does not want the default values.

A configuration mask value is created by combining individual field masks with through a bitwise OR operation of the following library constants:

SHMEM_TEAM_NUM_CONTEXTS The team should be created using the value of the *num_contexts* member of the configuration parameter *config* as a requirement.

A configuration mask value of *0* indicates that the team should be created with the default values for all configuration parameters.

The default values for configuration parameters are:

num_contexts = *0* By default, no contexts can be created on a new team

Notes

None.

9.4.4 SHMEM_TEAM_GET_CONFIG

Return the configuration parameters of a given team

SYNOPSIS

C/C++:

```
int shmem_team_get_config(shmem_team_t team, shmem_team_config_t *config);
```

DESCRIPTION

Arguments

IN	*team*	An OpenSHMEM team handle.
OUT	*config*	A pointer to the configuration parameters for the given team.

API description

shmem_team_get_config returns through the *config* argument the configuration parameters of the given team, which were assigned according to input configuration parameters when the team was created.

If *team* compares equal to *SHMEM_TEAM_INVALID*, then no operation is performed. If *team* is otherwise invalid, the behavior is undefined.

Return Values

If *team* does not compare equal to *SHMEM_TEAM_INVALID*, then *shmem_team_get_config* returns *0*; otherwise, it returns nonzero.

Notes

None.

9.4.5 SHMEM_TEAM_TRANSLATE_PE

Translate a given PE number from one team to the corresponding PE number in another team.

SYNOPSIS

C/C++:

```
int shmem_team_translate_pe(shmem_team_t src_team, int src_pe,
    shmem_team_t dest_team);
```

DESCRIPTION

Arguments

IN	*src_team*	An OpenSHMEM team handle.
IN	*src_pe*	A PE number in *src_team*.
IN	*dest_team*	An OpenSHMEM team handle.

API description

The *shmem_team_translate_pe* routine will translate a given PE number in one team into the corresponding PE number in another team. Specifically, given the *src_pe* in *src_team*, this routine returns that PE's number in *dest_team*. If *src_pe* is not a member of both the *src_team* and *dest_team*, a value of *-1* is returned.

If at least one of *src_team* and *dest_team* compares equal to *SHMEM_TEAM_INVALID*, then *-1* is returned. If either of the *src_team* or *dest_team* handles are otherwise invalid, the behavior is undefined.

Return Values

The specified PE's number in the *dest_team*, or a value of *-1* if any team handle arguments are invalid or the *src_pe* is not in both the source and destination teams.

Notes

If *SHMEM_TEAM_WORLD* is provided as the *dest_team* parameter, this routine acts as a global PE number translator and will return the corresponding *SHMEM_TEAM_WORLD* number.

EXAMPLES

The following example demonstrates the use of the team PE number translation routine. The program makes a new team of all of the even number PEs in the default team. Then, all PEs in the new team acquire their PE number in the new team and translate it to the PE number in the default team.

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

int main(void)
{
    int                 my_pe;
    int                 n_pes;
    int                 t_pe;
```

```
    int      t_global;
    shmem_team_t         new_team;
    shmem_team_config_t *config;

    shmem_init();
    config = NULL;
    my_pe  = shmem_my_pe();
    n_pes  = shmem_n_pes();

    shmem_team_split_strided(SHMEM_TEAM_WORLD, 0, 2, (n_pes + 1) / 2,
                             config, 0, &new_team);

    if (new_team != SHMEM_TEAM_INVALID) {
        t_pe     = shmem_team_my_pe(new_team);
        t_global = shmem_team_translate_pe(new_team, t_pe, SHMEM_TEAM_WORLD);

        if (t_global != my_pe) {
            shmem_global_exit(1);
        }
    }

    shmem_finalize();
    return 0;
}
```

9.4.6 SHMEM_TEAM_SPLIT_STRIDED

Create a new OpenSHMEM team from a subset of the existing parent team PEs, where the subset is defined by the PE triplet (*start*, *stride*, and *size*) supplied to the routine.

SYNOPSIS

C/C++:

```
int shmem_team_split_strided(shmem_team_t parent_team, int start, int stride, int size,
    const shmem_team_config_t *config, long config_mask, shmem_team_t *new_team);
```

DESCRIPTION

Arguments

IN	*parent_team*	An OpenSHMEM team.
IN	*start*	The lowest PE number of the subset of PEs from the parent team that will form the new team.
IN	*stride*	The stride between team PE numbers in the parent team that comprise the subset of PEs that will form the new team.
IN	*size*	The number of PEs from the parent team in the subset of PEs that will form the new team.
IN	*config*	A pointer to the configuration parameters for the new team.
IN	*config_mask*	The bitwise mask representing the set of configuration parameters to use from *config*.
OUT	*new_team*	A new OpenSHMEM team handle, representing a PE subset of all the PEs in the parent team that is created from the PE triplet provided.

API description

The *shmem_team_split_strided* routine is a collective routine. It creates a new OpenSHMEM team from a subset of the existing parent team, where the PE subset is defined by the triplet of arguments (*start*, *stride*, *size*). A valid triplet is one such that:

$$start + stride \cdot i \in \mathbb{Z}_N \ \forall \ i \in \mathbb{Z}_{size}$$

where N is the number of PEs in the parent team.

This routine must be called by all PEs in the parent team. All PEs must provide the same values for the PE triplet. This routine will return a *new_team* containing the PE subset specified by the triplet and ordered by the existing global PE number.

On successful creation of the new team:

- The *new_team* handle will reference a valid team for the subset of PEs in the parent team specified by the triplet.
- Those PEs in the parent team that are not in the subset specified by the triplet will have *new_team* assigned to *SHMEM_TEAM_INVALID*.
- *shmem_team_split_strided* will return zero to all PEs in the parent team.

If the new team cannot be created, then *new_team* will be assigned the value *SHMEM_TEAM_INVALID* and *shmem_team_split_strided* will return a nonzero value on all PEs in the parent team.

The *config* argument specifies team configuration parameters, which are described in Section 9.4.3.

The *config_mask* argument is a bitwise mask representing the set of configuration parameters to use from *config*. A *config_mask* value of *0* indicates that the team should be created with the default values for all configuration parameters. See Section 9.4.3 for field mask names and default configuration parameters.

If *parent_team* compares equal to *SHMEM_TEAM_INVALID*, then no new team will be created and *new_team* will be assigned the value *SHMEM_TEAM_INVALID*. If *parent_team* is otherwise invalid, the behavior is undefined.

If an invalid PE triplet is provided, then the *new_team* will not be created.

Return Values

Zero on successful creation of *new_team*; otherwise, nonzero.

Notes

It is important to note the use of the less restrictive *stride* argument instead of *logPE_stride*. This method of creating a team with an arbitrary set of PEs is inherently restricted by its parameters, but allows for many additional use-cases over using a *logPE_stride* parameter, and may provide an easier transition for existing OpenSHMEM programs to create and use OpenSHMEM teams.

See the description of team handles and predefined teams at the top of Section 9.4 for more information about semantics and usage.

EXAMPLES

The following example demonstrates the use of strided split in a *C11* program. The program creates a new team of all even number PEs from the default team, then retrieves the PE number and team size on all PEs that are members of the new team.

```
/*
 * OpenSHMEM shmem_team_split_strided example to create a team of all even
 * ranked PEs from SHMEM_TEAM_WORLD
 */
```

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

int main(int argc, char *argv[])
{
    int                  rank, npes;
    int                  t_pe, t_size;
    shmem_team_t         new_team;
    shmem_team_config_t *config;

    shmem_init();
    config = NULL;
    rank   = shmem_my_pe();
    npes   = shmem_n_pes();

    shmem_team_split_strided(SHMEM_TEAM_WORLD, 0, 2, npes / 2, config, 0,
                             &new_team);

    if (new_team != SHMEM_TEAM_INVALID) {
        t_size = shmem_team_n_pes(new_team);
        t_pe   = shmem_team_my_pe(new_team);

        if ((rank % 2 != 0) || (rank / 2 != t_pe) || (npes / 2 != t_size)) {
            shmem_global_exit(1);
        }
    }

    shmem_finalize();
    return 0;
}
```

9.4.7 SHMEM_TEAM_SPLIT_2D

Create two new teams by splitting an existing parent team into two subsets based on a 2D Cartesian space defined by the *xrange* argument and a *y* dimension derived from *xrange* and the parent team size. These ranges describe the Cartesian space in *x*- and *y*-dimensions.

SYNOPSIS

C/C++:

```c
int shmem_team_split_2d(shmem_team_t parent_team, int xrange,
    const shmem_team_config_t *xaxis_config, long xaxis_mask, shmem_team_t *xaxis_team,
    const shmem_team_config_t *yaxis_config, long yaxis_mask, shmem_team_t *yaxis_team);
```

DESCRIPTION

Arguments

IN	*parent_team*	A valid OpenSHMEM team. Any predefined teams, such as *SHMEM_TEAM_WORLD*, may be used, or any team created by the user.
IN	*xrange*	A nonnegative integer representing the number of elements in the first dimension.
IN	*xaxis_config*	A pointer to the configuration parameters for the new *x*-axis team.
IN	*xaxis_mask*	The bitwise mask representing the set of configuration parameters to use from *xaxis_config*.
OUT	*xaxis_team*	A new PE team handle representing a PE subset consisting of all the PEs that have the same coordinate along the *x*-axis as the calling PE.

IN	*yaxis_config*	A pointer to the configuration parameters for the new *y*-axis team.
IN	*yaxis_mask*	The bitwise mask representing the set of configuration parameters to use from *yaxis_config*.
OUT	*yaxis_team*	A new PE team handle representing a PE subset consisting of all the PEs that have the same coordinate along the *y*-axis as the calling PE.

API description

The *shmem_team_split_2d* routine is a collective routine. It creates two new teams by splitting an existing parent team into up to two subsets based on a 2D Cartesian space. The user provides the size of the x dimension, which is then used to derive the size of the y dimension based on the size of the parent team. The size of the y dimension will be equal to $\lceil N \div xrange \rceil$, where N is the size of the parent team. In other words, $xrange \times yrange \geq N$, so that every PE in the parent team has a unique *(x,y)* location the 2D Cartesian space.

The mapping of PE number to coordinates is $(x,y) = (pe \mod xrange, \lfloor pe \div xrange \rfloor)$, where *pe* is the PE number in the parent team. So, if $xrange = 3$, then the first 3 PEs in the parent team will form the first *xteam*, the second three PEs in the parent team form the second *xteam*, and so on.

Thus, after the split operation, each of the new *xteam*s will contain all PEs that have the same coordinate along the *y*-axis as the calling PE. Each of the new *yteam*s will contain all PEs with the same coordinate along the *x*-axis as the calling PE.

The PEs are numbered in the new teams based on the coordinate of the PE along the given axis. So, another way to think of the result of the split operation is that the value returned by *shmem_team_my_pe(xteam)* is the x-coordinate and the value returned by *shmem_team_my_pe(yteam)* is the y-coordinate of the calling PE.

Any valid OpenSHMEM team can be used as the parent team. This routine must be called by all PEs in the parent team. The value of *xrange* must be nonnegative and all PEs in the parent team must pass the same value for *xrange*.

The *xaxis_config* and *yaxis_config* arguments specify team configuration parameters for the *x*- and *y*-axis teams, respectively. These parameters are described in Section 9.4.3. All PEs that will be in the same resultant team must specify the same configuration parameters. The PEs in the parent team *do not* have to all provide the same parameters for new teams.

The *xaxis_mask* and *yaxis_mask* arguments are a bitwise masks representing the set of configuration parameters to use from *xaxis_config* and *yaxis_config*, respectively. A mask value of *0* indicates that the team should be created with the default values for all configuration parameters. See Section 9.4.3 for field mask names and default configuration parameters.

If *parent_team* compares equal to *SHMEM_TEAM_INVALID*, then no new teams will be created and both *xaxis_team* and *yaxis_team* will be assigned the value *SHMEM_TEAM_INVALID*. If *parent_team* is otherwise invalid, the behavior is undefined.

If any *xaxis_team* or *yaxis_team* on any PE in *parent_team* cannot be created, then both team handles on all PEs in *parent_team* will be assigned the value *SHMEM_TEAM_INVALID* and *shmem_team_split_2d* will return a nonzero value.

Return Values

Zero on successful creation of all *xaxis_team*s and *yaxis_team*s; otherwise, nonzero.

Notes

Since the split may result in a 2D space with more points than there are members of the parent team, there may be a final, incomplete row of the 2D mapping of the parent team. This means that the resultant *yteam*s may vary in size by up to 1 PE, and that there may be one resultant *xteam* of smaller size than all of the other *xteam*s.

The following grid shows the 12 teams that would result from splitting a parent team of size 10 with *xrange* of 3. The numbers in the grid cells are the PE numbers in the parent team. The rows are the *xteam*s. The columns are the *yteam*s.

	yteam x=0	yteam x=1	yteam x=2
xteam, y=0	0	1	2
xteam, y=1	3	4	5
xteam, y=2	6	7	8
xteam, y=3	9		

It would be legal, for example, if PEs 0, 3, 6, 9 specified a different value for *yaxis_config* than all of the other PEs, as long as the configuration parameters match for all PEs in each of the new teams.

See the description of team handles and predefined teams at the top of section 9.4 for more information about team handle semantics and usage.

EXAMPLES

The following example demonstrates the use of 2D Cartesian split in a *C11* program. This example shows how multiple 2D splits can be used to generate a 3D Cartesian split. This method can be extrapolated to generate splits of any number of dimensions.

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

int main(void)
{
  int xdim = 3;
  int ydim = 4;

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

  if (npes < (xdim*ydim)) {
    printf ("Not enough PEs to create 4x3xN layout\n");
    exit(1);
  }

  int zdim = (npes / (xdim*ydim)) + ( ((npes % (xdim*ydim)) > 0) ? 1 : 0 );
  shmem_team_t xteam, yzteam, yteam, zteam;

  shmem_team_split_2d(SHMEM_TEAM_WORLD, xdim, NULL, 0, &xteam, NULL, 0, &yzteam);
  // yzteam is immediately ready to be used in collectives
  shmem_team_split_2d(yzteam, ydim, NULL, 0, &yteam, NULL, 0, &zteam);

  // We don't need the yzteam anymore
  shmem_team_destroy(yzteam);

  int my_x = shmem_team_my_pe(xteam);
  int my_y = shmem_team_my_pe(yteam);
  int my_z = shmem_team_my_pe(zteam);

  for (int zdx = 0; zdx < zdim; zdx++)
    for (int ydx = 0; ydx < ydim; ydx++)
      for (int xdx = 0; xdx < xdim; xdx++) {
        if ((my_x == xdx) && (my_y == ydx) && (my_z == zdx)) {
          printf ("(%d, %d, %d) is me = %d\n", my_x, my_y, my_z, pe);
        }
        shmem_team_sync(SHMEM_TEAM_WORLD);
      }

  shmem_finalize();
}
```

The example above splits *SHMEM_TEAM_WORLD* into a 3D team with dimensions 3x4xN. For example, if *npes* = 16, *xdim* = 3, and *ydim* = 4, then the final dimensions are 3x4x2. In this case, the first split of *SHMEM_TEAM_WORLD* results in 6 *xteams* and 3 *yzteams*:

		yzteam		
		$x = 0$	$x = 1$	$x = 2$
xteam	$yz = 0$	0	1	2
	$yz = 1$	3	4	5
	$yz = 2$	6	7	8
	$yz = 3$	9	10	11
	$yz = 4$	12	13	14
	$yz = 5$	15		

The second split of *yzteam* for $x = 0$, *ydim* = 4 results in 2 *yteams* and 4 *zteams*:

		zteam			
		$y = 0$	$y = 1$	$y = 2$	$y = 3$
yteam	$z = 0$	0	3	6	9
	$z = 1$	12	15		

The second split of *yzteam* for $x = 1$, *ydim* = 4 results in 2 *yteams* and 4 *zteams*:

		zteam			
		$y = 0$	$y = 1$	$y = 2$	$y = 3$
yteam	$z = 0$	1	4	7	10
	$z = 1$	13			

The second split of *yzteam* for $x = 2$, *ydim* = 4 results in 2 *yteams* and 4 *zteams*:

		zteam			
		$y = 0$	$y = 1$	$y = 2$	$y = 3$
yteam	$z = 0$	2	5	8	11
	$z = 1$	14			

The final number of teams for each dimension are:

- 6 *xteams*: these are teams where (*z*,*y*) is fixed and *x* varies.
- 6 *yteams*: these are teams where (*x*,*z*) is fixed and *y* varies.
- 12 *zteams*: these are teams where (*x*,*y*) is fixed and *z* varies.

The expected output is:

```
(0, 0, 0) is me = 0
(1, 0, 0) is me = 1
(2, 0, 0) is me = 2
(0, 1, 0) is me = 3
(1, 1, 0) is me = 4
(2, 1, 0) is me = 5
(0, 2, 0) is me = 6
(1, 2, 0) is me = 7
(2, 2, 0) is me = 8
(0, 3, 0) is me = 9
(1, 3, 0) is me = 10
(2, 3, 0) is me = 11
(0, 0, 1) is me = 12
(1, 0, 1) is me = 13
(2, 0, 1) is me = 14
(0, 1, 1) is me = 15
```

9.4.8 SHMEM_TEAM_DESTROY

Destroy an existing team.

SYNOPSIS

C/C++:

```
void shmem_team_destroy(shmem_team_t team);
```

DESCRIPTION

Arguments

IN	*team*	An OpenSHMEM team handle.

API description

The *shmem_team_destroy* routine is a collective operation that destroys the team referenced by the team handle argument *team*. Upon return, the referenced team is invalid.

This routine destroys all shareable contexts created from the referenced team. The user is responsible for destroying all contexts created from this team with the *SHMEM_CTX_PRIVATE* option enabled prior to calling this routine; otherwise, the behavior is undefined.

It is an error to destroy the default team or any other predefined team.

If *team* compares equal to *SHMEM_TEAM_INVALID*, then no operation is performed. If *team* is otherwise invalid, the behavior is undefined.

Return Values

None.

Notes

None.

9.5 Communication Management Routines

All OpenSHMEM RMA, AMO, and memory ordering routines must be performed on a valid 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.

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

The default context is valid for the duration of the OpenSHMEM portion of an application. Contexts created by a successful call to *shmem_ctx_create* remain valid until they are destroyed. A handle value that does not correspond to a valid context is considered to be invalid, and its use in RMA and AMO routines results in undefined behavior. A context handle may be initialized to or assigned the value *SHMEM_CTX_INVALID* to indicate that handle does not reference a valid communication context. When managed in this way, applications can use an equality comparison to test whether a given context handle references a valid context.

Every communication context is associated with a team. This association is established at context creation. Communication contexts created by *shmem_ctx_create* are associated with the default team, while contexts created by *shmem_team_create_ctx* are associated with and created from a team specified at context creation. The default context is associated with the default team. A context's associated team specifies the set of PEs over which PE-specific routines that operate on a communication context, explicitly or implicitly, are performed. All point-to-point routines that operate on this context will do so with respect to the team-relative PE numbering of the associated team. If the PE number passed to such a routine is invalid, being negative or greater than or equal to the size of the OpenSHMEM team, then the behavior is undefined.

9.5.1 SHMEM_CTX_CREATE

Create a communication context locally.

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 and the context handle pointed to by *ctx* specifies a valid context; otherwise, a nonzero value is returned and the context handle pointed to by *ctx* is not modified. 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.

A newly created communication context has a fixed association with the default team. All OpenSHMEM routines that operate on this context will do so with respect to the associated PE team. That is, all point-to-point routines operating on this context will use team-relative PE numbering.

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.

9.5.2 SHMEM_TEAM_CREATE_CTX

Create a communication context from a team locally.

SYNOPSIS

C/C++:

```
int shmem_team_create_ctx(shmem_team_t team, long options, shmem_ctx_t *ctx);
```

DESCRIPTION

Arguments

IN	*team*	A handle to the specified PE team.
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_team_create_ctx* routine creates a new communication context and returns its handle through the *ctx* argument. This context is created from the team specified by the *team* argument.

In addition to the team, the *shmem_team_create_ctx* routine accepts the same arguments and provides all the same return conditions as the *shmem_ctx_create* routine.

The *shmem_team_create_ctx* routine may be called any number of times to create multiple simultaneously existing contexts for the team. Programs should request the total number of simultaneous contexts to be created from the team during team creation. See Section 9.4.3 for more information on how to request contexts during team creation.

A call to *shmem_team_create_ctx* on a team may fail, regardless of the configuration request for contexts, if the implementation is unable to create a context at the time when *shmem_team_create_ctx* is called.

All explicitly created resources associated with a team must be destroyed before the *shmem_team_destroy* routine is called. If a context returned from *shmem_team_create_ctx* is not explicitly destroyed before the team is destroyed, behavior is undefined.

All OpenSHMEM routines that operate on this context will do so with respect to the associated PE team. That is, all point-to-point routines operating on this context will use team-relative PE numbering.

Return Values

Zero on success and nonzero otherwise.

Notes

None.

EXAMPLES

See example in Section 9.5.4

9.5.3 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* or *shmem_team_create_ctx*. 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. If *ctx* has the value *SHMEM_CTX_INVALID*, no operation is performed.

Return Values

None.

Notes

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.

```c
#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*.

```c
#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;
}
```

The following example demonstrates the use of *SHMEM_CTX_INVALID* in a *C11* program that uses thread-local storage to provide each thread an implicit context handle via a "library" put routine without explicit management of the context handle from "user" code.

```
#include <stddef.h>
#include <shmem.h>
#include <omp.h>

_Thread_local shmem_ctx_t thread_ctx = SHMEM_CTX_INVALID;

void lib_thread_register(void) {
  if (thread_ctx == SHMEM_CTX_INVALID)
    if (shmem_ctx_create(SHMEM_CTX_PRIVATE, &thread_ctx) &&
        shmem_ctx_create(                0, &thread_ctx))
      thread_ctx = SHMEM_CTX_DEFAULT;
}
```

```
void lib_thread_unregister(void) {
  if (thread_ctx != SHMEM_CTX_DEFAULT) {
    shmem_ctx_destroy(thread_ctx);
    thread_ctx = SHMEM_CTX_INVALID;
  }
}

void lib_thread_putmem(void *dst, const void *src, size_t nbytes, int pe) {
  shmem_ctx_putmem(thread_ctx, dst, src, nbytes, pe);
}

int main() {
  int provided;
  if (shmem_init_thread(SHMEM_THREAD_MULTIPLE, &provided))
    return 1;
  if (provided != SHMEM_THREAD_MULTIPLE)
    shmem_global_exit(2);

  const int my_pe = shmem_my_pe();
  const int n_pes = shmem_n_pes();
  const int count = 1 << 15;

  int *src_bufs[n_pes];
  int *dst_bufs[n_pes];
  for (int i = 0; i < n_pes; i++) {
    src_bufs[i] = shmem_calloc(count, sizeof(*src_bufs[i]));
    if (src_bufs[i] == NULL)
      shmem_global_exit(3);
    dst_bufs[i] = shmem_calloc(count, sizeof(*dst_bufs[i]));
    if (dst_bufs[i] == NULL)
      shmem_global_exit(4);
  }

#pragma omp parallel
  {
    int my_thrd = omp_get_thread_num();
#pragma omp for
    for (int i = 0; i < n_pes; i++)
      for (int j = 0; j < count; j++)
        src_bufs[i][j] = (my_pe << 10) + my_thrd;

    lib_thread_register();

#pragma omp for
    for (int i = 0; i < n_pes; i++)
      lib_thread_putmem(dst_bufs[my_pe], src_bufs[i],
                        count * sizeof(*src_bufs[i]), i);

    lib_thread_unregister();
  }

  shmem_finalize();
  return 0;
}
```

9.5.4 SHMEM_CTX_GET_TEAM

Retrieve the team associated with the communication context.

SYNOPSIS

C/C++:

```
int shmem_ctx_get_team(shmem_ctx_t ctx, shmem_team_t *team);
```

DESCRIPTION

Arguments

IN	*ctx*	A handle to a communication context.
OUT	*team*	A pointer to a handle to the associated PE team.

API description

The *shmem_ctx_get_team* routine returns a handle to the team associated with the specified communication context *ctx*. The team handle is returned through the pointer argument *team*.

If *ctx* is the default context or one created by a call to *shmem_ctx_create*, the *team* is assigned the handle value *SHMEM_TEAM_WORLD*.

When *ctx* is an invalid context, if *ctx* compares equal to *SHMEM_CTX_INVALID*, then *team* is assigned the value *SHMEM_TEAM_INVALID* and a nonzero value is returned; otherwise, the behavior is undefined.

If *team* is a null pointer, the behavior is undefined.

Return Values

Zero on success; otherwise, nonzero.

Notes

None.

EXAMPLES

The following example demonstrates the use of contexts for multiple teams in a *C11* program. This example shows contexts being used to communicate within a team using team PE numbers, and across teams using translated PE numbers.

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

int isum, ival;

int my_ctx_translate_pe(shmem_ctx_t src_ctx, int src_pe, shmem_ctx_t dest_ctx)
{
  if (src_ctx == SHMEM_CTX_INVALID) {
    return -1;
  }
  if (dest_ctx == SHMEM_CTX_INVALID) {
    return -1;
  }

  shmem_team_t src_team, dest_team;
  shmem_ctx_get_team(src_ctx, &src_team);
  shmem_ctx_get_team(dest_ctx, &dest_team);
  return shmem_team_translate_pe(src_team, src_pe, dest_pe);
}

shmem_ctx_t my_team_create_ctx(shmem_team_t team) {
  if (team == SHMEM_TEAM_INVALID) {
    return SHMEM_CTX_INVALID;
  }

  shmem_ctx_t ctx;
  if (shmem_team_create_ctx(team, 0, &ctx) != 0) {
    fprintf (stderr, "Failed to create context for PE team\n");
    return SHMEM_CTX_INVALID;
```

```
  }
  return ctx;
}

void my_send_to_neighbor(shmem_ctx_t ctx, int *val)
{
  if (ctx == SHMEM_CTX_INVALID) {
    fprintf (stderr, "Send to neighbor fail due to invalid context\n");
    return;
  }

  shmem_team_t team;
  shmem_ctx_get_team(ctx, &team);
  int pe = shmem_team_my_pe(team);
  int npes = shmem_team_n_pes(team);
  int rpe = (pe + 1) % npes;

  // put my pe number in the buffer on my right hand neighbor
  shmem_ctx_int_put(ctx, val, &pe, 1, rpe);
}



int main()
{
  shmem_init();

  int npes = shmem_n_pes();
  isum = 0;

  shmem_team_t team_2s, team_3s;
  shmem_ctx_t ctx_2s, ctx_3s;
  shmem_team_config_t conf;
  conf.num_contexts = 1;
  long cmask = SHMEM_TEAM_NUM_CONTEXTS;

  // Create team with PEs numbered 0, 2, 4, ...
  shmem_team_spit_strided(SHMEM_TEAM_WORLD, 0, 2, npes / 2, &conf, cmask, &team_2s);
  // Create team with PEs numbered 0, 3, 6, ...
  shmem_team_split_strided(SHMEM_TEAM_WORLD, 0, 3, npes / 3, &conf, cmask, &team_3s);

  ctx_2s = my_team_create_ctx(team_2s);
  ctx_3s = my_team_create_ctx(team_3s);

  // Send some values using the two team contexts contexts
  my_send_to_neighbor(ctx_2s, &ival2);
  my_send_to_neighbor(ctx_3s, &ival3);

  // Quiet all contexts and synchronize all PEs to complete the data transfers
  shmem_ctx_quiet(ctx_2s);
  shmem_ctx_quiet(ctx_3s);
  shmem_team_sync(SHMEM_TEAM_WORLD);

  // We will add up some results on pe 4 of team_3s using ctx_2s
  if ((team_3s != SHMEM_TEAM_INVALID) && (team_2s != SHMEM_TEAM_INVALID)) {
    int pe4_of_3s_in_2s = my_ctx_translate_pe(ctx_3s, 4, ctx_2s);

    if (pe4_of_3s_in_2s < 0) {
      fprintf (stderr, "Fail to translate pe 4 from 3s context to 2s context\n");
    }
    else {
      // Add up the results on pe 4 of the 3s team, using the 2s team context
      shmem_ctx_int_atomic_add(ctx_2s, &isum, ival2 + ival3, _pe4_of_3s_in_2s);
    }
  }

  // Quiet the context and synchronize PEs to complete the operation
  shmem_ctx_quiet(ctx_2s);
```

```
  shmem_team_sync(SHMEM_TEAM_WORLD);

  if (shmem_team_my_pe(team_3s) == 4) {
    printf ("The total value on PE 4 of the 3s team is %d\n", isum);
  }

  // Destroy contexts before teams
  shmem_ctx_destroy(ctx_2s);
  shmem_team_destroy(team_2s);

  shmem_ctx_destroy(ctx_3s);
  shmem_team_destroy(team_3s);

  shmem_finalize();
}
```

9.6 Remote Memory Access Routines

The RMA routines described in this section can be used to perform reads from and writes to symmetric data objects. These operations are one-sided, meaning that the PE invoking an operation provides all communication parameters and the targeted PE is passive. A characteristic of one-sided communication is that it decouples communication from synchronization. One-sided communication mechanisms transfer data; however, they do not synchronize the sender of the data with the receiver of the data.

OpenSHMEM RMA routines are performed on symmetric data objects. The initiator PE of a call is designated as the *origin* PE and the PE targeted by an operation is designated as the *destination* PE. The *source* and *dest* designators refer to the data objects that an operation reads from and writes to. In the case of the remote update routine, *Put*, the origin PE provides the *source* data object and the destination PE provides the *dest* data object. In the case of the remote read routine, *Get*, the origin PE provides the *dest* data object and the destination PE provides the *source* data object.

The destination PE is specified as an integer representing the PE number. This PE number is relative to the team associated with the communication context being using for the operation. If no context argument is passed to the routine, then the routine operates on the default context, which implies that the PE number is relative to the default team. If the PE number passed to the routine is invalid, being negative or greater than or equal to the size of the OpenSHMEM team, then the behavior is undefined.

Where appropriate compiler support is available, OpenSHMEM provides type-generic one-sided communication interfaces via *C11* generic selection (*C11* §6.5.1.1[4]) for block, scalar, and block-strided put and get communication. Such type-generic routines are supported for the "standard RMA types" listed in Table 4.

The standard RMA types include the exact-width integer types defined in *stdint.h* by *C99*[5] §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 implementation is not required to provide support for these types.

9.6.1 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);
```

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

C/C++:

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

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 4: Standard RMA Types and Names

```
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 4.

```
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);
```

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*.

IN	*pe*	PE number of the remote PE. *pe* must be of type integer.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

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;
}
```

9.6.2 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 4.

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 4.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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.

If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

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;
}
```

9.6.3 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 4.

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 4.

```
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*.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*.
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*.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

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;
}
```

9.6.4 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 4.

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 4.

```
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);
```

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

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

9.6.5 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 4.

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 4.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

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;
}
```

9.6.6 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 4.

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 4.

```
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*.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*.
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*.
IN	*nelems*	Number of elements in the *dest* and *source* arrays. *nelems* must be of type *size_t* for *C*.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

9.7 Nonblocking Remote Memory Access Routines

9.7.1 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 4.

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 4.

```
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);
```

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer.

API description

The routines return after initiating 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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

9.7.2 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 4.

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 4.

```
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);
```

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*.
IN	*pe*	PE number of the remote PE. *pe* must be of type integer.

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 initiating 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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

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

9.8 Atomic Memory Operations

An 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 9.6, 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 5,
- *shmem_atomic_{fetch, set, swap}* support the "extended AMO types" listed in Table 6, and
- *shmem_atomic_{fetch_and, and, fetch_or, or, fetch_xor, xor}* support the "bitwise AMO types" listed in Table 7.

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.

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 5: 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 6: Extended AMO Types and Names

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 7: Bitwise AMO Types and Names

9.8.1 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 6.

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);
```

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

— 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 6.

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

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.

9.8.2 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 6.

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 6.

— 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 6.

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

9.8.3 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 5.

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 5.

— 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 5.

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

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;
}
```

9.8.4 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 6.

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 6.

— 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 6.

deprecation end

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

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;
}
```

9.8.5 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 5.

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 5.

— 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 5.

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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.

API description

These routines perform a fetch-and-increment operation. The *dest* on PE *pe* is increased by one and the routine returns the previous contents of *dest* as an atomic operation. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

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;
}
```

9.8.6 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 5.

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 5.

— 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 5.

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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.

API description

These routines perform an atomic increment operation on the *dest* data object on PE. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

EXAMPLES

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

```
#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;
}
```

9.8.7 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 5.

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 5.

— 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 5.

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

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:

```c
#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;
}
```

9.8.8 SHMEM_ATOMIC_ADD

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

SYNOPSIS

C11:

```c
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 5.

C/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 5.

— deprecation start —

C11:

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

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

C/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 5.

— deprecation end —

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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.
IN	*pe*	An integer that indicates the PE number upon which *dest* is to be updated.

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. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

EXAMPLES

```
#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;
}
```

9.8.9 SHMEM_ATOMIC_FETCH_AND

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

SYNOPSIS

C11:

```
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 7.

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 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

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

Notes

None.

9.8.10 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 7.

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 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

9.8.11 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 7.

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 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

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

Notes

None.

9.8.12 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 7.

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 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

9.8.13 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 7.

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 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

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

Notes

None.

9.8.14 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 7.

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 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*. If the context handle *ctx* does not correspond to a valid context, the behavior is undefined.

Return Values

None.

Notes

None.

9.9 Nonblocking Atomic Memory Operations

9.9.1 SHMEM_ATOMIC_FETCH_NBI

The nonblocking atomic fetch routine provides a method for atomically fetching the value of a remote data object.

SYNOPSIS

C11:

```
void shmem_atomic_fetch_nbi(TYPE *fetch, const TYPE *source, int pe);
void shmem_atomic_fetch_nbi(shmem_ctx_t ctx, TYPE *fetch, const TYPE *source, int pe);
```

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

C/C++:

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

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

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
OUT	*fetch*	Local data object to be updated.
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

The nonblocking atomic fetch routines perform a nonblocking fetch of a value atomically from a remote data object. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, contents of the *source* data object from PE has been fetched into *fetch* local data object.

Return Values

None.

Notes

None.

9.9.2 SHMEM_ATOMIC_COMPARE_SWAP_NBI

The nonblocking atomic routine provides a method for performing an atomic conditional swap on a remote data object.

SYNOPSIS

C11:

```
void shmem_atomic_compare_swap_nbi(TYPE *fetch, TYPE *dest, TYPE cond, TYPE value, int pe);
void shmem_atomic_compare_swap_nbi(shmem_ctx_t ctx, TYPE *fetch, TYPE *dest, TYPE cond, TYPE
    value, int pe);
```

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

C/C++:

```
void shmem_<TYPENAME>_atomic_compare_swap_nbi(TYPE *fetch, TYPE *dest, TYPE cond, TYPE
    value, int pe);
void shmem_ctx_<TYPENAME>_atomic_compare_swap_nbi(shmem_ctx_t ctx, TYPE *fetch, 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 5.

DESCRIPTION

Arguments

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

API description

The nonblocking conditional swap routines conditionally update a *dest* data object on the specified PE as an atomic operation and fetches the prior contents of the *dest* data object into the *fetch* local data object. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, prior contents of the *dest* data object have been fetched into *fetch* local data object and the contents of *value* have been conditionally updated into *dest* on the remote PE.

Return Values

None.

Notes

None.

9.9.3 SHMEM_ATOMIC_SWAP_NBI

This nonblocking atomic operation performs an atomic swap to a remote data object.

SYNOPSIS

C11:

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

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

C/C++:

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

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

DESCRIPTION

Arguments

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

The nonblocking *shmem_atomic_swap_nbi* routines perform an atomic swap operation. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, it has written *value* into *dest* on PE and fetched the prior contents of *dest* into *fetch* local data object.

Return Values

None.

Notes

None.

9.9.4 SHMEM_ATOMIC_FETCH_INC_NBI

This nonblocking atomic routine performs an atomic fetch-and-increment operation on a remote data object.

SYNOPSIS

C11:

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

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

C/C++:

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

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

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
OUT	*fetch*	Local data object to be updated.
OUT	*dest*	The remotely accessible data object to be updated on the remote PE.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be updated.

API description

The nonblocking *shmem_atomic_fetch_inc_nbi* routines perform an atomic fetch-and-increment operation. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, *dest* on PE *pe* has been increased by one and the previous contents of *dest* fetched into the *fetch* local data object.

Return Values

None.

Notes

None.

9.9.5 SHMEM_ATOMIC_FETCH_ADD_NBI

The nonblocking atomic routine performs an atomic fetch-and-add operation on a remote data object.

SYNOPSIS

C11:

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

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

C/C++:

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

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

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
OUT	*fetch*	Local data object to be updated.
OUT	*dest*	The remotely accessible data object to be updated on the remote PE.
IN	*value*	The value to be atomically added to *dest*.
IN	*pe*	An integer that indicates the PE number on which *dest* is to be updated.

API description

The nonblocking *shmem_atomic_fetch_add_nbi* 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. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, *value* has been added to *dest* on *pe* and the prior contents of *dest* fetched into the *fetch* local data object.

Return Values

None.

Notes

None.

9.9.6 SHMEM_ATOMIC_FETCH_AND_NBI

This nonblocking atomic operation performs an atomic fetching bitwise AND operation on a remote data object.

SYNOPSIS

C11:

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

where *TYPE* is one of the bitwise AMO types specified by Table 7.

C/C++:

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

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
OUT	*fetch*	Local data object to be updated.
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

The nonblocking *shmem_atomic_fetch_and_nbi* routines perform an atomic fetching bitwise AND on the remotely accessible data object pointed by *dest* at PE *pe* with the operand *value*. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, these routines have performed a fetching bitwise AND on *dest* at PE *pe* with the operand *value* and fetched the prior contents of *dest* into the *fetch* local data object.

Return Values

None.

Notes

None.

9.9.7 SHMEM_ATOMIC_FETCH_OR_NBI

This nonblocking atomic operation performs an atomic fetching bitwise OR operation on a remote data object.

SYNOPSIS

C11:

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

where *TYPE* is one of the bitwise AMO types specified by Table 7.

C/C++:

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

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
OUT	*fetch*	Local data object to be updated.
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

The nonblocking *shmem_atomic_fetch_or_nbi* routines perform an atomic fetching bitwise OR on the remotely accessible data object pointed by *dest* at PE *pe* with the operand *value*. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, these routines have performed a fetching bitwise OR on *dest* at PE *pe* with the operand *value* and fetched the prior contents of *dest* into the *fetch* local data object.

Return Values

None.

Notes

None.

9.9.8 SHMEM_ATOMIC_FETCH_XOR_NBI

This nonblocking atomic operation performs an atomic fetching bitwise XOR operation on a remote data object.

SYNOPSIS

C11:

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

where *TYPE* is one of the bitwise AMO types specified by Table 7.

C/C++:

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

where *TYPE* is one of the bitwise AMO types and has a corresponding *TYPENAME* specified by Table 7.

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
OUT	*fetch*	Local data object to be updated.
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

The nonblocking *shmem_atomic_fetch_xor_nbi* routines perform an atomic fetching bitwise XOR on the remotely accessible data object pointed by *dest* at PE *pe* with the operand *value*. This routine returns after initiating the operation. The operation is considered complete after a subsequent call to *shmem_quiet*. At the completion of *shmem_quiet*, these routines have performed a fetching bitwise XOR on *dest* at PE *pe* with the operand *value* and fetched the prior contents of *dest* into the *fetch* local data object.

Return Values

None.

Notes

None.

9.10 Signaling Operations

This section specifies the OpenSHMEM support for *put-with-signal*, nonblocking *put-with-signal*, and *signal-fetch* routines. The put-with-signal routines provide a method for copying data from a contiguous local data object to a data object on a specified PE and subsequently updating a remote flag to signal completion. The signal-fetch routine provides support for fetching a signal update operation.

9.10.1 Atomicity Guarantees for Signaling Operations

All signaling operations put-with-signal, nonblocking put-with-signal, and signal-fetch are performed on a signal data object, a remotely accessible symmetric object of type *uint64_t*. A signal operator in the put-with-signal routine is a OpenSHMEM library constant that determines the type of update to be performed as a signal on the signal data object.

All signaling operations on the signal data object completes as if performed atomically with respect to the following:

- other blocking or nonblocking variant of the put-with-signal routine that updates the signal data object using the same signal update operator;
- signal-fetch routine that fetches the signal data object; and
- any point-to-point synchronization routine that accesses the signal data object.

9.10.2 Available Signal Operators

With the atomicity guarantees as described in Section 9.10.1, the following options can be used as a signal operator.

SHMEM_SIGNAL_SET	An update to signal data object is an atomic set operation. It writes an unsigned 64-bit value as a signal into the signal data object on a remote *PE* as an atomic operation.
SHMEM_SIGNAL_ADD	An update to signal data object is an atomic add operation. It adds an unsigned 64-bit value as a signal into the signal data object on a remote *PE* as an atomic operation.

9.10.3 SHMEM_PUT_SIGNAL

The *put-with-signal* routines provide a method for copying data from a contiguous local data object to a data object on a specified PE and subsequently updating a remote flag to signal completion.

SYNOPSIS

C11:

```
void shmem_put_signal(TYPE *dest, const TYPE *source, size_t nelems, uint64_t *sig_addr,
    uint64_t signal, int sig_op, int pe);
void shmem_put_signal(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t nelems,
    uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

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

C/C++:

```
void shmem_<TYPENAME>_put_signal(TYPE *dest, const TYPE *source, size_t nelems, uint64_t
    *sig_addr, uint64_t signal, int sig_op, int pe);
void shmem_ctx_<TYPENAME>_put_signal(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t
    nelems, uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

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

```
void shmem_put<SIZE>_signal(void *dest, const void *source, size_t nelems, uint64_t
    *sig_addr, uint64_t signal, int sig_op, int pe);
void shmem_ctx_put<SIZE>_signal(shmem_ctx_t ctx, void *dest, const void *source, size_t
    nelems, uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

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

```
void shmem_putmem_signal(void *dest, const void *source, size_t nelems, uint64_t *sig_addr,
    uint64_t signal, int sig_op, int pe);
void shmem_ctx_putmem_signal(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems,
    uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.
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*.
OUT	*sig_addr*	signal data object to be updated on the remote PE as a signal. This signal data object must be remotely accessible.

IN	*signal*	Unsigned 64-bit value that is used for updating the remote *sig_addr* signal data object.
IN	*sig_op*	Signal operator that represents the type of update to be performed on the remote *sig_addr* signal data object.
IN	*pe*	PE number of the remote PE.

API description

The *put-with-signal* routines provide a method for copying data from a contiguous local data object to a data object on a specified PE and subsequently updating a remote flag to signal completion. The routines return after the data has been copied out of the *source* array on the local PE.

The *sig_op* signal operator determines the type of update to be performed on the remote *sig_addr* signal data object. The completion of signal update based on the *sig_op* signal operator using the *signal* flag on the remote PE indicates the delivery of its corresponding *dest* data words into the data object on the remote PE.

An update to the *sig_addr* signal data object through a *put-with-signal* routine completes as if performed atomically as described in Section 9.10.1. The various options as described in Section 9.10.2 can be used as the *sig_op* signal operator.

Return Values

None.

Notes

The *dest* and *sig_addr* data objects must both be remotely accessible. The *sig_addr* and *dest* could be of different kinds, for example, one could be a global/static *C* variable and the other could be allocated on the symmetric heap.

sig_addr and *dest* may not be overlapping in memory.

The completion of signal update using the *signal* flag on the remote PE indicates only the delivery of its corresponding *dest* data words into the data object on the remote PE. Without a memory-ordering operation, there is no implied ordering between the signal update of a *put-with-signal* routine and another data transfer. For example, the completion of the signal update in a sequence consisting of a put routine followed by a *put-with-signal* routine does not imply delivery of the put routine's data.

EXAMPLES

The following example demonstrates the usage of *shmem_put_signal*. It shows the implementation of a broadcast operation from PE 0 to itself and all other PEs in the job as a simple ring-based algorithm using *shmem_put_signal*:

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

int main(void)
{
    int i, err_count = 0;

    shmem_init();

    size_t          size    = 2048;
    int             me      = shmem_my_pe();
    int             n       = shmem_n_pes();
    int             pe      = (me + 1) % n;
    uint64_t *      message = malloc(size * sizeof(uint64_t));
```

```
    static uint64_t sig_addr = 0;

    for (i = 0; i < size; i++) {
        message[i] = me;
    }

    uint64_t *data = shmem_calloc(size, sizeof(uint64_t));

    if (me == 0) {
        shmem_put_signal(data, message, size, &sig_addr, 1, SHMEM_SIGNAL_SET, pe);
    } else {
        shmem_wait_until(&sig_addr, SHMEM_CMP_EQ, 1);
        shmem_put_signal(data, data, size, &sig_addr, 1, SHMEM_SIGNAL_SET, pe);
    }

    free(message);
    shmem_free(data);

    shmem_finalize();
    return 0;
}
```

9.10.4 SHMEM_PUT_SIGNAL_NBI

The nonblocking *put-with-signal* routines provide a method for copying data from a contiguous local data object to a data object on a specified PE and subsequently updating a remote flag to signal completion.

SYNOPSIS

C11:

```
void shmem_put_signal_nbi(TYPE *dest, const TYPE *source, size_t nelems, uint64_t *sig_addr,
    uint64_t signal, int sig_op, int pe);
void shmem_put_signal_nbi(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t nelems,
    uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

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

C/C++:

```
void shmem_<TYPENAME>_put_signal_nbi(TYPE *dest, const TYPE *source, size_t nelems, uint64_t
    *sig_addr, uint64_t signal, int sig_op, int pe);
void shmem_ctx_<TYPENAME>_put_signal_nbi(shmem_ctx_t ctx, TYPE *dest, const TYPE *source,
    size_t nelems, uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

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

```
void shmem_put<SIZE>_signal_nbi(void *dest, const void *source, size_t nelems, uint64_t
    *sig_addr, uint64_t signal, int sig_op, int pe);
void shmem_ctx_put<SIZE>_signal_nbi(shmem_ctx_t ctx, void *dest, const void *source, size_t
    nelems, uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

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

```
void shmem_putmem_signal_nbi(void *dest, const void *source, size_t nelems, uint64_t
    *sig_addr, uint64_t signal, int sig_op, int pe);
void shmem_ctx_putmem_signal_nbi(shmem_ctx_t ctx, void *dest, const void *source, size_t
    nelems, uint64_t *sig_addr, uint64_t signal, int sig_op, int pe);
```

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.

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*.
OUT	*sig_addr*	Data object to be updated on the remote PE as the signal. This signal data object must be remotely accessible.
IN	*signal*	Unsigned 64-bit value that is used for updating the remote *sig_addr* signal data object.
IN	*sig_op*	Signal operator that represents the type of update to be performed on the remote *sig_addr* signal data object.
IN	*pe*	PE number of the remote PE.

API description

The nonblocking *put-with-signal* routines provide a method for copying data from a contiguous local data object to a data object on a specified PE and subsequently updating a remote flag to signal completion.

The routines return after initiating 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 out of the *source* array on the local PE and delivered into the *dest* array on the destination PE.

The delivery of *signal* flag on the remote PE indicates only the delivery of its corresponding *dest* data words into the data object on the remote PE. Furthermore, two successive nonblocking *put-with-signal* routines, or a nonblocking *put-with-signal* routine with another data transfer may deliver data out of order unless a call to *shmem_fence* is introduced between the two calls.

The *sig_op* signal operator determines the type of update to be performed on the remote *sig_addr* signal data object.

An update to the *sig_addr* signal data object through a nonblocking *put-with-signal* routine completes as if performed atomically as described in Section 9.10.1. The various options as described in Section 9.10.2 can be used as the *sig_op* signal operator.

Return Values

None.

Notes

The *dest* and *sig_addr* data objects must both be remotely accessible. The *sig_addr* and *dest* could be of different kinds, for example, one could be a global/static *C* variable and the other could be allocated on the symmetric heap.

sig_addr and *dest* may not be overlapping in memory.

9.10.5 SHMEM_SIGNAL_FETCH

Fetches the signal update on a local data object.

SYNOPSIS

C/C++:

```
uint64_t shmem_signal_fetch(const uint64_t *sig_addr);
```

DESCRIPTION

Arguments

IN	*sig_addr*	A pointer to a remotely accessible variable.

API description

shmem_signal_fetch performs a fetch operation and returns the contents of the *sig_addr* signal data object. Access to *sig_addr* signal object at the calling PE is expected to satisfy the atomicity guarantees as described in Section 9.10.1.

Return Values

Returns the contents of the signal data object, *sig_addr*, at the calling PE.

Notes

None.

9.11 Collective Routines

Collective routines are defined as coordinated communication or synchronization operations performed by a group of PEs.

OpenSHMEM provides three types of collective routines:

1. Collective routines that operate on teams use a team handle parameter to determine which PEs will participate in the routine, and use resources encapsulated by the team object to perform operations. See Section 9.4 for details on team management.

— deprecation start —

2. Collective routines that operate on active sets use a set of parameters to determine which PEs will participate and what resources are used to perform operations.

— deprecation end —

3. Collective routines that accept neither team nor active set parameters, which implicitly operate on the default team and, as required, the default context.

Team-based collectives

The team-based collective routines are performed with respect to a valid OpenSHMEM team, which is specified by a team handle argument. Team-based collective operations require all PEs in the team to call the routine in order for the operation to complete. If an invalid team handle or *SHMEM_TEAM_INVALID* is passed to a team-based collective routine, the behavior is undefined.

Team objects encapsulate the per PE system resources required to complete team-based collective routines. All OpenSHMEM teams-based collective calls are blocking routines which may use those system resources. On completion of a team-based collective call, the PE may immediately call another collective on that same team without any other intervening synchronization across the team.

While OpenSHMEM routines provide thread support according to the thread-support level provided at initialization (see Section 9.2), team-based collective routines may not be called simultaneously by multiple threads on a given team.

Collective operations are matched across a given team based on ordering. So for a given team, collectives must occur in the same order across all PEs in a team.

The team-based collective routines defined in the OpenSHMEM Specification are:

- *shmem_team_sync*
- *shmem_{TYPE_}broadcast{mem}*
- *shmem_{TYPE_}collect{mem}*
- *shmem_{TYPE_}fcollect{mem}*
- Reduction routines for the following operations: AND, OR, XOR, MAX, MIN, SUM, PROD
- *shmem_{TYPE_}alltoall{mem}*
- *shmem_{TYPE_}alltoalls{mem}*

In addition, all team creation functions are collective operations. In addition to the ordering and thread safety requirements described here, there are additional synchronization requirements on team creation operations. See Section 9.4 for more details.

— deprecation start —

Active-set-based collectives

The active-set-based 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.

The active set is defined by the arguments *PE_start*, *logPE_stride*, and *PE_size*. *PE_start* specifies the starting PE number and is the lowest numbered PE in the active set. The stride between successive PEs in the active set is 2^{logPE_stride} and *logPE_stride* must be greater than or equal to zero. *PE_size* specifies the number of PEs in the active set and must be greater than zero. The active set must satisfy the requirement that its last member corresponds to a valid PE number, that is $0 \leq PE\_start + (PE\_size - 1) * 2^{logPE\_stride} < npes$.

All PEs participating in the active-set-based collective routine must provide the same values for these arguments. If any of these requirements are not met, the behavior is undefined.

Another argument important to active-set-based collective routines is *pSync*, which is a symmetric work array. All PEs participating in an active-set-based collective must pass the same *pSync* array. On completion of such 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 active-set-based 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 active-set-based collective routines defined in the OpenSHMEM Specification are:

- *shmem_barrier*
- *shmem_sync*
- *shmem_broadcast{32, 64}*
- *shmem_collect{32, 64}*
- *shmem_fcollect{32, 64}*
- Reduction routines for the following operations: AND, MAX, MIN, SUM, PROD, OR, XOR
- *shmem_alltoall{32, 64}*
- *shmem_alltoalls{32, 64}*

— deprecation end —

Team-implicit collectives

The *shmem_sync_all* routine synchronizes all PEs in the computation through the default team. This routine is equivalent to a call to *shmem_team_sync* on the default team.

The *shmem_barrier_all* routine synchronizes all PEs in the default team and ensures completion of all local and remote memory updates issued via the default context. This routine is equivalent to a call to *shmem_ctx_quiet* on the default context followed by a call to *shmem_team_sync* on the default team.

Error codes returned from collectives

Collective operations involving multiple PEs may return values indicating success while other PEs are still executing the collective operation. Return values indicating success or failure of a collective routine on one PE may not indicate that all PEs involved in the collective operation will return the same value. Some operations, such as team creation, must return identical return codes across multiple PEs.

9.11.1 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);
```

DESCRIPTION

Arguments

None.

API description

The *shmem_barrier_all* routine is a mechanism for synchronizing all PEs in the default team at once. This routine blocks the calling PE until all PEs have called *shmem_barrier_all*. In a multithreaded OpenSHMEM program, only the calling thread is blocked, however, it may not be called concurrently by multiple threads in the same PE.

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 is equivalent to calling *shmem_ctx_quiet* on the default context followed by calling *shmem_team_sync* on the default team.

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;
}
```

9.11.2 SHMEM_BARRIER

— deprecation start —

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);
```

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 of size *SHMEM_BARRIER_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. 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

As of OpenSHMEM 1.5, *shmem_barrier* has been deprecated. No team-based barrier is provided by OpenSHMEM, as a team may have any number of communication contexts associated with the team. Applications seeking such an idiom should call *shmem_ctx_quiet* on the desired communication context, followed by a call to *shmem_team_sync* on the desired team.

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;
}
```

deprecation end

9.11.3 SHMEM_SYNC

Registers the arrival of a PE at a synchronization point and suspends execution until all other PEs in a given OpenSHMEM team or active set arrive at a synchronization point. For multithreaded programs, execution is suspended as specified by the threading model (Section 9.2).

SYNOPSIS

C11:

```
int shmem_sync(shmem_team_t team);
```

C/C++:

```
int shmem_team_sync(shmem_team_t team);
```

deprecation start

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

deprecation end

DESCRIPTION

Arguments

IN	*team*	The team over which to perform the operation.

deprecation start

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.

deprecation end

API description

shmem_sync is a collective synchronization routine over an existing OpenSHMEM team or active set.

The routine registers the arrival of a PE at a synchronization point in the program. This is a fast mechanism for synchronizing all PEs that participate in this collective call. The routine blocks the calling PE until all PEs in the specified team or active set have called *shmem_sync*. In a multithreaded OpenSHMEM program, only the calling thread is blocked.

Team-based sync routines operate over all PEs in the provided team argument. All PEs in the provided team must participate in the sync operation. If an invalid team handle or *SHMEM_TEAM_INVALID* is passed to this routine, the behavior is undefined.

Active-set-based sync routines operate over all PEs in the active set defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

As with all active set-based 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 active set-based 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

Zero on successful local completion. Nonzero otherwise.

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;

   shmem_team_t         twos_team, threes_team;
   shmem_team_config_t *config;

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

   int odd_npes = npes % 2;

   shmem_team_split_strided(SHMEM_TEAM_WORLD, 0, 2, npes / 2, config, 0,
                            &twos_team);

   shmem_team_split_strided(SHMEM_TEAM_WORLD, 0, 3, npes / 3 + odd_npes,
                            config, 0, &threes_team);

   int my_pe_twos = shmem_team_my_pe(twos_team);
   int my_pe_threes = shmem_team_my_pe(threes_team);

   if (twos_team != SHMEM_TEAM_INVALID) {
      /* put the value 2 to the next team member in a circular fashion */
      shmem_p(&x, 2, (me + 2) % npes);
```

```
        shmem_quiet();
        shmem_sync(twos_team);
    }

    if (threes_team != SHMEM_TEAM_INVALID) {
        /* put the value 3 to the next team member in a circular fashion */
        shmem_p(&x, 3, (me + 3) % npes);
        shmem_quiet();
        shmem_sync(threes_team);
    }

    if (me % 3 == 0 && x != 3) {
        shmem_global_exit(3);
    }
    else if (me % 2 == 0 && x != 2) {
        shmem_global_exit(2);
    }
    else if (x != 10101)  {
        shmem_global_exit(1);
    }

    shmem_finalize();
    return 0;
}
```

9.11.4 SHMEM_SYNC_ALL

Registers the arrival of a PE at a synchronization point and suspends execution until all other PEs in the default team arrive at a synchronization point. For multithreaded programs, execution is suspended as specified by the threading model (Section 9.2).

SYNOPSIS

C/C++:

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

DESCRIPTION

Arguments

None.

API description

This routine blocks the calling PE until all PEs in the default team 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 is equivalent to calling *shmem_team_sync* on the default team.

9.11.5 SHMEM_BROADCAST

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

SYNOPSIS

C11:

```
int shmem_broadcast(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nelems, int
    PE_root);
```

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

C/C++:

```
int shmem_<TYPENAME>_broadcast(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nelems, int PE_root);
```

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

```
int shmem_broadcastmem(shmem_team_t team, void *dest, const void *source, size_t nelems, int
    PE_root);
```

— deprecation start —

```
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 end —

DESCRIPTION

Arguments

IN	*team*	The team over which to perform the operation.
OUT	*dest*	A symmetric data object. See the table below in this description for allowable types.
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*. 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 team or active set, from which the data is copied. *PE_root* must be of type *int*. When using *Fortran*, it must be a default integer value.

— deprecation start —

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. *log_PE_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 of size *SHMEM_BCAST_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. 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*.

— deprecation end —

API description

OpenSHMEM broadcast routines are collective routines over an active set or existing OpenSHMEM team. They copy data object *source* on the processor specified by *PE_root* and store the values at *dest* on the other PEs participating in the collective operation. The data is not copied to the *dest* area on the root PE.

The same *dest* and *source* data objects and the same value of *PE_root* must be passed by all PEs participating in the collective operation.

Team-based broadcast routines operate over all PEs in the provided team argument. All PEs in the provided team must participate in the operation. If an invalid team handle or *SHMEM_TEAM_INVALID* is passed to this routine, the behavior is undefined.

As with all team-based OpenSHMEM routines, PE numbering is relative to the team. The specified root PE must be a valid PE number for the team, between *0* and *N-1*, where *N* is the size of the team.

Active-set-based broadcast routines operate over all PEs in the active set defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

As with all active-set-based 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 active-set-based 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 value of *PE_root* must be between *0* and *PE_size*. 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 *dest* array on all PEs participating in the broadcast is ready to accept the broadcast data.
- If using active-set-based routines, the *pSync* array on all PEs in the active set is not still in use from a prior call to a collective OpenSHMEM routine.

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.
- If using active-set-based routines, 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_broadcastmem	*C*: Any data type. nelems is scaled in bytes.
shmem_broadcast64	No *C/C++* structures are allowed.
shmem_broadcast32	No *C/C++* structures are allowed.

Return Values

Zero on successful local completion. Nonzero otherwise.

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.

Team handle error checking and integer return codes are currently undefined. Implementations may define these behaviors as needed, but programs should ensure portability by doing their own checks for invalid team handles and for *SHMEM_TEAM_INVALID*.

EXAMPLES

In the following *C11* example, the call to *shmem_broadcast* copies *source* on PE 0 to *dest* on PEs $1 \ldots npes - 1$.

C/C++ example:

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

int main(void)
{
   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_broadcast(SHMEM_TEAM_WORLD, dest, source, 4, 0);

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

9.11.6 SHMEM_COLLECT, SHMEM_FCOLLECT

Concatenates blocks of data from multiple PEs to an array in every PE participating in the collective routine.

SYNOPSIS

C11:

```
int shmem_collect(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nelems);
int shmem_fcollect(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nelems);
```

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

C/C++:

```
int shmem_<TYPENAME>_collect(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nelems);
int shmem_<TYPENAME>_fcollect(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nelems);
```

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

```
int shmem_collectmem(shmem_team_t team, void *dest, const void *source, size_t nelems);
int shmem_fcollectmem(shmem_team_t team, void *dest, const void *source, size_t nelems);
```

— deprecation start —

```
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 end

DESCRIPTION

Arguments

IN	*team*	A valid OpenSHMEM team handle.
OUT	*dest*	A symmetric array large enough to accept the concatenation of the *source* arrays on all participating PEs. See table below in this description for allowable data types.
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*.

deprecation start

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 of size *SHMEM_COLLECT_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. 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*.

deprecation end

API description

OpenSHMEM *collect* and *fcollect* routines perform a collective operation to concatenate *nelems* data items from the *source* array into the *dest* array, over an OpenSHMEM team or active set in processor number order. The resultant *dest* array contains the contribution from PEs as follows:

- For an active set, the data from PE *PE_start* is first, then the contribution from PE *PE_start* + *PE_stride* second, and so on.
- For a team, the data from PE number *0* in the team is first, then the contribution from PE *1* in the team, and so on.

The collected result is written to the *dest* array for all PEs that participate in the operation. The same *dest* and *source* arrays must be passed by all PEs that participate in the operation.

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.

Team-based collect routines operate over all PEs in the provided team argument. All PEs in the provided team must participate in the operation.

Active-set-based collective routines operate over all PEs in the active set defined by the *PE_start*, *logPE_stride*, *PE_size* triplet. As with all active-set-based 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 *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.
- For active-set-based collective routines, 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_collectmem, *shmem_fcollectmem*	*C*: Any data type. *nelems* is scaled in bytes.
shmem_collect64, *shmem_fcollect64*	Any noncharacter type that has an element size of *64* bits. No *Fortran* derived types nor *C/C++* structures are allowed.
shmem_collect32, *shmem_fcollect32*	Any noncharacter type that has an element size of *32* bits. No *Fortran* derived types nor *C/C++* structures are allowed.

Return Values

Zero on successful local completion. Nonzero otherwise.

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;
```

```
    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;

    /* Wait for all PEs to initialize source/dest: */
    shmem_team_sync(SHMEM_TEAM_WORLD);

    shmem_int_collect(SHMEM_TEAM_WORLD, dest, source, my_nelem);

    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;
}
```

9.11.7 SHMEM_REDUCTIONS

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

SYNOPSIS

TYPE	*TYPENAME*	Operations Supporting *TYPE*		
unsigned char	uchar	AND, OR, XOR		
short	short	AND, OR, XOR	MAX, MIN	SUM, PROD
unsigned short	ushort	AND, OR, XOR	MAX, MIN	SUM, PROD
int	int	AND, OR, XOR	MAX, MIN	SUM, PROD
unsigned int	uint	AND, OR, XOR	MAX, MIN	SUM, PROD
long	long	AND, OR, XOR	MAX, MIN	SUM, PROD
unsigned long	ulong	AND, OR, XOR	MAX, MIN	SUM, PROD
long long	longlong	AND, OR, XOR	MAX, MIN	SUM, PROD
unsigned long long	ulonglong	AND, OR, XOR	MAX, MIN	SUM, PROD
float	float		MAX, MIN	SUM, PROD
double	double		MAX, MIN	SUM, PROD
long double	longdouble		MAX, MIN	SUM, PROD
double _Complex	complexd			SUM, PROD
float _Complex	complexf			SUM, PROD

Table 8: Reduction Types, Names and Supporting Operations

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

C11:

```
int shmem_and_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nreduce);
```

where *TYPE* is one of the integer types supported for the AND operation as specified by Table 8.

C/C++:

```
int shmem_<TYPENAME>_and_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nreduce);
```

— deprecation start —

```
void shmem_<TYPENAME>_and_to_all(TYPE *dest, const TYPE *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

— deprecation end —

where *TYPE* is one of the integer types supported for the AND operation and has a corresponding *TYPENAME* as specified by Table 8.

9.11.7.2 OR

Performs a bitwise OR reduction across a set of PEs.

C11:

```
int shmem_or_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nreduce);
```

where *TYPE* is one of the integer types supported for the OR operation as specified by Table 8.

C/C++:

```
int shmem_<TYPENAME>_or_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nreduce);
```

— deprecation start —

```
void shmem_<TYPENAME>_or_to_all(TYPE *dest, const TYPE *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

— deprecation end —

where *TYPE* is one of the integer types supported for the OR operation and has a corresponding *TYPENAME* as specified by Table 8.

9.11.7.3 XOR

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

C11:

```
int shmem_xor_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nreduce);
```

where *TYPE* is one of the integer types supported for the XOR operation as specified by Table 8.

C/C++:

```
int shmem_<TYPENAME>_xor_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nreduce);
```

— deprecation start —

```
void shmem_<TYPENAME>_xor_to_all(TYPE *dest, const TYPE *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

— deprecation end —

where *TYPE* is one of the integer types supported for the XOR operation and has a corresponding *TYPENAME* as specified by Table 8.

9.11.7.4 MAX

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

C11:

```
int shmem_max_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nreduce);
```

where *TYPE* is one of the integer or real types supported for the MAX operation as specified by Table 8.

C/C++:

```
int shmem_<TYPENAME>_max_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nreduce);
```

— deprecation start —

```
void shmem_<TYPENAME>_max_to_all(TYPE *dest, const TYPE *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

— deprecation end —

where *TYPE* is one of the integer or real types supported for the MAX operation and has a corresponding *TYPENAME* as specified by Table 8.

9.11.7.5 MIN

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

C11:

```
int shmem_min_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nreduce);
```

where *TYPE* is one of the integer or real types supported for the MIN operation as specified by Table 8.

C/C++:

```
int shmem_<TYPENAME>_min_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nreduce);
```

— deprecation start —

```
void shmem_<TYPENAME>_min_to_all(TYPE *dest, const TYPE *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

— deprecation end —

where *TYPE* is one of the integer or real types supported for the MIN operation and has a corresponding *TYPENAME* as specified by Table 8.

9.11.7.6 SUM

Performs a sum reduction across a set of PEs.

C11:

```
int shmem_sum_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nreduce);
```

where *TYPE* is one of the integer, real, or complex types supported for the SUM operation as specified by Table 8.

C/C++:

```
int shmem_<TYPENAME>_sum_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nreduce);
```

— deprecation start —

```
void shmem_<TYPENAME>_sum_to_all(TYPE *dest, const TYPE *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

— deprecation end —

where *TYPE* is one of the integer, real, or complex types supported for the SUM operation and has a corresponding *TYPENAME* as specified by Table 8.

9.11.7.7 PROD

Performs a product reduction across a set of PEs.

C11:

```
int shmem_prod_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nreduce);
```

where *TYPE* is one of the integer, real, or complex types supported for the PROD operation as specified by Table 8.

C/C++:

```
int shmem_<TYPENAME>_prod_reduce(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nreduce);
```

— deprecation start —

```
void shmem_<TYPENAME>_prod_to_all(TYPE *dest, const TYPE *source, int nreduce, int PE_start,
    int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

— deprecation end —

where *TYPE* is one of the integer, real, or complex types supported for the PROD operation and has a corresponding *TYPENAME* as specified by Table 8.

DESCRIPTION

Arguments

IN	*team*	The team over which to perform the operation.
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. In teams based API calls, *nreduce* must be of type size_t. In deprecated active-set based API calls, *nreduce* must be of type integer.

— deprecation start —

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	*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*. 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.

— deprecation end —

API description

OpenSHMEM reduction routines are collective routines over an active set or existing OpenSHMEM team that 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 participating in the reduction provides one element for each reduction. The results of the reductions are placed in the *dest* array on all PEs participating in the reduction.

The *source* and *dest* arrays may be the same array, but they may not be overlapping arrays. The same *dest* and *source* arrays must be passed to all PEs participating in the reduction.

Team-based reduction routines operate over all PEs in the provided team argument. All PEs in the provided team must participate in the reduction. If an invalid team handle or *SHMEM_TEAM_INVALID* is passed to this routine, the behavior is undefined.

Active-set-based sync routines operate over all PEs in the active set defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

As with all active set-based 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 active set-based 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 *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 *dest* array on all PEs participating in the reduction is ready to accept the results of the *reduction*.
- If using active-set-based routines, 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.

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.
- If using active-set-based routines, 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 [6], an OpenSHMEM implementation is not required to provide support for these complex-typed interfaces.

Return Values

Zero on successful local completion. Nonzero otherwise.

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 *C/C++* reduction example gets integers from an external source (random genererator in this example), tests to see if the PE got a valid value, and outputs the sum of values for which all PEs got a valid value.

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

/* As if we receive some value from external source */
long recv_a_value(unsigned seed, int npes) {
```

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

```
  srand(seed);
  return rand() % npes;
}

/* Validate the value we recieved */
unsigned char is_valid(long value, int npes) {
  if (value == (npes-1))
    return 0;
  return 1;
}

int main(void)
{

  shmem_init();
  int me = shmem_my_pe();
  int npes = shmem_n_pes();
  size_t num = 32;

  long *values = shmem_malloc(num * sizeof(long));
  long *sums = shmem_malloc(num * sizeof(long));

  unsigned char *valid_me = shmem_malloc(num * sizeof(unsigned char));
  unsigned char *valid_all = shmem_malloc(num * sizeof(unsigned char));

  values[0] = recv_a_value((unsigned)me, npes);
  valid_me[0] = is_valid(values[0], npes);

  for (int i=1; i < num; i++) {
    values[i] = recv_a_value((unsigned)values[i-1], npes);
    valid_me[i] = is_valid(values[i], npes);
  }

  /* Wait for all PEs to initialize reductions arrays */
  shmem_sync(SHMEM_TEAM_WORLD);

  shmem_and_reduce(SHMEM_TEAM_WORLD, valid_all, valid_me, num);
  shmem_sum_reduce(SHMEM_TEAM_WORLD, sums, values, num);

  for (int i=0; i < num; i++) {
    if (valid_all[i]) {
      printf ("[%d] = %ld\n", i, sums[i]);
    }
    else {
      printf ("[%d] = invalid on one or more pe\n", i);
    }
  }

  shmem_finalize();
  return 0;
}
```

9.11.8 SHMEM_ALLTOALL

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

SYNOPSIS

C11:

```
int shmem_alltoall(shmem_team_t team, TYPE *dest, const TYPE *source, size_t nelems);
```

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

C/C++:

```
int shmem_<TYPENAME>_alltoall(shmem_team_t team, TYPE *dest, const TYPE *source, size_t
    nelems);
```

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

```
int shmem_alltoallmem(shmem_team_t team, void *dest, const void *source, size_t nelems);
```

— deprecation start —

```
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 end —

DESCRIPTION

Arguments

IN	*team*	A valid OpenSHMEM team handle to a team.
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++*.

— deprecation start —

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 of size *SHMEM_ALLTOALL_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. 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.

— deprecation end —

API description

The *shmem_alltoall* routines are collective routines. Each PE participating in the operation exchanges *nelems* data elements with all other PEs participating in the operation. The size of a data element is:

- 32 bits for *shmem_alltoall32*
- 64 bits for *shmem_alltoall64*
- 8 bits for *shmem_alltoallmem*
- *sizeof*(*TYPE*) for alltoall routines taking typed *source* and *dest*

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 times *N*, where *N* equals the number of PEs participating in the operation. The *source* object contains *N* blocks of data (where the size of each block is defined by *nelems*) and each block of data is sent to a different PE.

The same *dest* and *source* arrays, and same value for nelems must be passed by all PEs that participate in the collective.

Given a PE *i* that is the k^{th}PE participating in the operation and a PE *j* that is the l^{th}PE participating in the operation,

PE *i* sends the l^{th}block of its *source* object to the k^{th}block of the *dest* object of PE *j*.

Team-based collect routines operate over all PEs in the provided team argument. All PEs in the provided team must participate in the collective.

Active-set-based collective routines operate over all PEs in the active set defined by the *PE_start*, *logPE_stride*, *PE_size* triplet.

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

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.
- For active-set-based routines, 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

Zero on successful local completion. Nonzero otherwise.

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 *C/C++* example shows a *shmem_int64_alltoall* on two 64-bit integers among all PEs.

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

int main(void)
{
   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 initialize source/dest */
   shmem_team_sync(SHMEM_TEAM_WORLD);

   /* alltoall on all PES */
   shmem_int64_alltoall(SHMEM_TEAM_WORLD, dest, source, count);

   /* 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;
}
```

9.11.9 SHMEM_ALLTOALLS

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

SYNOPSIS

C11:

```c
int shmem_alltoalls(shmem_team_t team, TYPE *dest, const TYPE *source, ptrdiff_t dst,
    ptrdiff_t sst, size_t nelems);
```

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

C/C++:

```c
int shmem_<TYPENAME>_alltoalls(shmem_team_t team, TYPE *dest, const TYPE *source, ptrdiff_t
    dst, ptrdiff_t sst, size_t nelems);
```

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

```c
int shmem_alltoallsmem(shmem_team_t team, void *dest, const void *source, ptrdiff_t dst,
    ptrdiff_t sst, size_t nelems);
```

— deprecation start —

```
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 end

DESCRIPTION

Arguments

IN	*team*	A valid OpenSHMEM team handle.
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*.
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 *ptrdiff_t*.

deprecation start

IN	*nelems*	The number of elements to exchange for each PE. *nelems* must be of type size_t for *C/C++*.
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 of size *SHMEM_ALLTOALLS_SYNC_SIZE*. In *C/C++*, *pSync* must be an array of elements of type *long*. 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.

deprecation end

API description

The *shmem_alltoalls* routines are collective routines. These routines are equivalent in functionality to the corresponding *shmem_alltoall* routines except that they add explicit stride values for accessing the source and destination data arrays, whereas the array access in *shmem_alltoall* is always with a stride of *1*.

Each PE participating in the operation exchanges *nelems* strided data elements with all other PEs participating in the operation. Both strides, *dst* and *sst*, must be greater than or equal to *1*.

The same *dest* and *source* arrays and same values for values of arguments *dst*, *sst*, *nelems* must be passed by all PEs that participate in the collective.

Given a PE *i* that is the k^{th}PE participating in the operation and a PE *j* that is the l^{th}PE participating in the operation 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*.

See the description of *shmem_alltoall* in Section 9.11.8 for:

- Data element sizes for the different sized and typed *shmem_alltoalls* variants.
- Rules for PE participation in the collective routine.
- The pre- and post-conditions for symmetric objects.
- Typing constraints for *dest* and *source* data objects.

Return Values

Zero on successful local completion. Nonzero otherwise.

Notes

See notes for *shmem_alltoall* in Section 9.11.8.

EXAMPLES

This *C/C++* example shows a *shmem_int64_alltoalls* on two 64-bit integers among all PEs.

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

int main(void)
{
   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 initialize source/dest */
   shmem_team_sync(SHMEM_TEAM_WORLD);

   /* alltoalls on all PES */
   shmem_int64_alltoalls(SHMEM_TEAM_WORLD, dest, source, dst, sst, count);

   /* 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;
}
```

9.12 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 *put-with-signal*, *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 9.

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. The *shmem_test_any* and *shmem_wait_until_any* routines require the *SIZE_MAX* macro defined in *stdint.h* by *C99* §7.18.3 and *C11* §7.20.3.

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 9: 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 10.

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 10: Point-to-Point Comparison Constants

9.12.1 SHMEM_WAIT_UNTIL

Wait for a variable on the local PE to change.

SYNOPSIS

C11:

```
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 9.

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 9.

— 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 9.

— deprecation end —

DESCRIPTION

Arguments

IN	*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 *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.

API description

The *shmem_wait* and *shmem_wait_until* operations block until the value contained in the symmetric data object, *ivar*, at the calling PE satisfies the wait condition. The *ivar* object at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE.

These routines can be used to implement point-to-point synchronization between PEs or between threads within the same PE. A call to *shmem_wait* blocks until the value of *ivar* at the calling PE is not equal to *cmp_value*. A call to *shmem_wait_until* blocks until the value of *ivar* at the calling PE satisfies the wait condition specified by the comparison operator, *cmp*, and comparison value, *cmp_value*.

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.

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

Some platforms may allow wait operations to efficiently poll or block on an update to *ivar*. On others, an

atomic read operation may be needed to observe updates to *ivar*. On platforms where atomic read operations incur high overhead, implementations may be able to reduce the number of atomic reads performed by using non-atomic reads of *ivar* to wait for a change to occur, followed by an atomic read operation to fetch the updated value.

9.12.2 SHMEM_WAIT_UNTIL_ALL

Wait on an array of variables on the local PE until all variables meet the specified wait condition.

SYNOPSIS

C11:

```
void shmem_wait_until_all(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
void shmem_<TYPENAME>_wait_until_all(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the wait set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with *cmp_value*.
IN	*cmp_value*	The value to be compared with the objects pointed to by *ivars*.

API description

The *shmem_wait_until_all* routine waits until all entries in the wait set specified by *ivars* and *status* have satisfied the wait condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. If *nelems* is 0, the wait set is empty and this routine returns immediately. This routine compares each element of the *ivars* array in the wait set with the value *cmp_value* according to the comparison operator *cmp* at the calling PE. This routine is semantically similar to *shmem_wait_until* in Section 9.12.1, but adds support for point-to-point synchronization involving an array of symmetric data objects.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the wait set. Elements of *status* set to 0 will be included in the wait set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the wait set is empty and this routine returns immediately. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the wait set. The *ivars* and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_wait_until_all* does not return before the update of the memory indicated by *ivars* is fully complete.

Return Values

None.

Notes

None.

EXAMPLES

The following *C11* example demonstrates the use of *shmem_wait_until_all* to implement a simple linear barrier synchronization.

```
#include <shmem.h>

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

  int *flags = shmem_calloc(npes, sizeof(int));
  int *status = NULL;

  for (int i = 0; i < npes; i++)
      shmem_atomic_set(&flags[mype], 1, i);

  shmem_wait_until_all(flags, npes, status, SHMEM_CMP_EQ, 1);

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

9.12.3 SHMEM_WAIT_UNTIL_ANY

Wait on an array of variables on the local PE until any one variable meets the specified wait condition.

SYNOPSIS

C11:

```
size_t shmem_wait_until_any(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_wait_until_any(TYPE *ivars, size_t nelems, const int *status,
    int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.

IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the wait set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with *cmp_value*.
IN	*cmp_value*	The value to be compared with the objects pointed to by *ivars*.

API description

The *shmem_wait_until_any* routine waits until any one entry in the wait set specified by *ivars* and *status* satisfies the wait condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine compares each element of the *ivars* array in the wait set with the value *cmp_value* according to the comparison operator *cmp* at the calling PE. The order in which these elements are waited upon is unspecified. If an entry *i* in *ivars* within the wait set satisfies the wait condition, a series of calls to *shmem_wait_until_any* must eventually return *i*.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the wait set. Elements of *status* set to 0 will be included in the wait set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the wait set is empty and this routine returns *SIZE_MAX*. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the wait set. The *ivars* and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_wait_until_any* does not return before the update of the memory indicated by *ivars* is fully complete.

Return Values

shmem_wait_until_any returns the index of an element in the *ivars* array that satisfies the wait condition. If the wait set is empty, this routine returns *SIZE_MAX*.

Notes

None.

EXAMPLES

The following *C11* example demonstrates the use of *shmem_wait_until_any* to process a simple all-to-all transfer of *N* data elements via a sum reduction.

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

#define N 100

int main(void)
{
  int total_sum = 0;

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

  int *my_data = malloc(N * sizeof(int));
  int *all_data = shmem_malloc(N * npes * sizeof(int));

  int *flags = shmem_calloc(npes, sizeof(int));
  int *status = calloc(npes, sizeof(int));
```

```
  for (int i = 0; i < N; i++)
      my_data[i] = mype*N + i;

  for (int i = 0; i < npes; i++)
      shmem_put_nbi(&all_data[mype*N], my_data, N, i);

  shmem_fence();

  for (int i = 0; i < npes; i++)
      shmem_atomic_set(&flags[mype], 1, i);

  for (int i = 0; i < npes; i++) {
      size_t completed_idx = shmem_wait_until_any(flags, npes, status, SHMEM_CMP_NE, 0);
      for (int j = 0; j < N; j++) {
          total_sum += all_data[completed_idx * N + j];
      }
      status[completed_idx] = 1;
  }

  /* check the result */
  int M = N * npes - 1;
  if (total_sum != M * (M + 1) / 2) {
      shmem_global_exit(1);
  }

  shmem_finalize();
  return 0;
}
```

9.12.4 SHMEM_WAIT_UNTIL_SOME

Wait on an array of variables on the local PE until at least one variable meets the specified wait condition.

SYNOPSIS

C11:

```
size_t shmem_wait_until_some(TYPE *ivars, size_t nelems, size_t *indices, const int *status,
    int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_wait_until_some(TYPE *ivars, size_t nelems, size_t *indices,
    const int *status, int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
OUT	*indices*	An array of indices of length at least *nelems* into *ivars* that satisfied the wait condition.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the wait set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with *cmp_value*.

IN	*cmp_value*	The value to be compared with the objects pointed to by *ivars*.

API description

The *shmem_wait_until_some* routine waits until at least one entry in the wait set specified by *ivars* and *status* satisfies the wait condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine compares each element of the *ivars* array in the wait set with the value *cmp_value* according to the comparison operator *cmp* at the calling PE. This routine tests all elements of *ivars* in the wait set at least once, and the order in which the elements are waited upon is unspecified.

Upon return, the *indices* array contains the indices of at least one element in the wait set that satisfied the wait condition during the call to *shmem_wait_until_some*. The return value of *shmem_wait_until_some* is equal to the total number of these satisfied elements. For a given return value *N*, the first *N* elements of the *indices* array contain those unique indices that satisfied the wait condition. These first *N* elements of *indices* may be unordered with respect to the corresponding indices of *ivars*. The array pointed to by *indices* must be at least *nelems* long. If an entry *i* in *ivars* within the wait set satisfies the wait condition, a series of calls to *shmem_wait_until_some* must eventually include *i* in the *indices* array.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the wait set. Elements of *status* set to 0 will be included in the wait set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the wait set is empty and this routine returns 0. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the wait set. The *ivars*, *indices*, and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_wait_until_some* does not return before the update of the memory indicated by *ivars* is fully complete.

Return Values

shmem_wait_until_some returns the number of indices returned in the *indices* array. If the wait set is empty, this routine returns 0.

Notes

None.

EXAMPLES

The following *C11* example demonstrates the use of *shmem_wait_until_some* to process a simple all-to-all transfer of *N* data elements via a sum reduction. This pattern is similar to the *shmem_wait_until_any* example above, but may reduce the number of iterations in the while loop.

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

#define N 100

int main(void)
{
  int total_sum = 0;

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

  int *my_data = malloc(N * sizeof(int));
  int *all_data = shmem_malloc(N * npes * sizeof(int));
```

```
    int *flags = shmem_calloc(npes, sizeof(int));
    size_t *indices = malloc(npes * sizeof(size_t));
    int *status = calloc(npes, sizeof(int));

    for (int i = 0; i < N; i++)
        my_data[i] = mype*N + i;

    for (int i = 0; i < npes; i++)
        shmem_put_nbi(&all_data[mype*N], my_data, N, i);

    shmem_fence();

    for (int i = 0; i < npes; i++)
        shmem_atomic_set(&flags[mype], 1, i);

    size_t ncompleted;
    while ((ncompleted = shmem_wait_until_some(flags, npes, indices,
                                               status, SHMEM_CMP_NE, 0))) {
        for (size_t i = 0; i < ncompleted; i++) {
            for (size_t j = 0; j < N; j++) {
                total_sum += all_data[indices[i]*N + j];
            }
            status[indices[i]] = 1;
        }
    }

    /* check the result */
    int M = N * npes - 1;
    if (total_sum != M * (M + 1) / 2) {
        shmem_global_exit(1);
    }

    shmem_finalize();
    return 0;
}
```

9.12.5 SHMEM_WAIT_UNTIL_ALL_VECTOR

Wait on an array of variables on the local PE until all variables meet the specified wait conditions.

SYNOPSIS

C11:

```
void shmem_wait_until_all_vector(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
void shmem_<TYPENAME>_wait_until_all_vector(TYPE *ivars, size_t nelems, const int *status,
    int cmp, TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.

IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the wait set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with elements of *cmp_values*.
IN	*cmp_values*	An array of length *nelems* containing values to be compared with the respective objects in *ivars*.

API description

The *shmem_wait_until_all_vector* routine waits until all entries in the wait set specified by *ivars* and *status* have satisfied the wait conditions at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. If *nelems* is 0, the wait set is empty and this routine returns immediately. This routine compares each element of the *ivars* array in the wait set with each respective value in *cmp_values* according to the comparison operator *cmp* at the calling PE.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the wait set. Elements of *status* set to 0 will be included in the wait set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the wait set is empty and this routine returns immediately. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the wait set. The *ivars* and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_wait_until_all_vector* does not return before the update of the memory indicated by *ivars* is fully complete.

Return Values

None.

Notes

None.

9.12.6 SHMEM_WAIT_UNTIL_ANY_VECTOR

Wait on an array of variables on the local PE until any one variable meets its specified wait condition.

SYNOPSIS

C11:

```
size_t shmem_wait_until_any_vector(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_wait_until_any_vector(TYPE *ivars, size_t nelems, const int *status,
    int cmp, TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the wait set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with elements of *cmp_values*.
IN	*cmp_values*	An array of length *nelems* containing values to be compared with the respective objects in *ivars*

.

API description

The *shmem_wait_until_any_vector* routine waits until any one entry in the wait set specified by *ivars* and *status* satisfies the wait condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine compares each element of the *ivars* array in the wait set with each respective value in *cmp_values* according to the comparison operator *cmp* at the calling PE. The order in which these elements are waited upon is unspecified. If an entry *i* in *ivars* within the wait set satisfies the wait condition, a series of calls to *shmem_wait_until_any_vector* must eventually return *i*.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the wait set. Elements of *status* set to 0 will be included in the wait set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the wait set is empty and this routine returns *SIZE_MAX*. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the wait set. The *ivars* and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_wait_until_any_vector* does not return before the update of the memory indicated by *ivars* is fully complete.

Return Values

shmem_wait_until_any_vector returns the index of an element in the *ivars* array that satisfies the wait condition. If the wait set is empty, this routine returns *SIZE_MAX*.

Notes

None.

EXAMPLES

The following *C11* example demonstrates the use of *shmem_wait_until_any_vector* to wait on values that differ between even PEs and odd PEs.

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

#define N 100

int main(void)
{
  int total_sum = 0;

  shmem_init();
  int mype = shmem_my_pe();
  int npes = shmem_n_pes();
```

```
    int *ivars = shmem_calloc(npes, sizeof(int));
    int *status = calloc(npes, sizeof(int));
    int *cmp_values = malloc(npes * sizeof(int));

    /* All odd PEs put 2 and all even PEs put 1 */
    for (int i = 0; i < npes; i++) {
        shmem_atomic_set(&ivars[mype], mype % 2 + 1, i);

        /* Set cmp_values to the expected values coming from each PE */
        cmp_values[i] = i % 2 + 1;
    }

    for (int i = 0; i < npes; i++) {
        size_t completed_idx = shmem_wait_until_any_vector(ivars, npes, status,
                                                           SHMEM_CMP_EQ, cmp_values);
        status[completed_idx] = 1;
        total_sum += ivars[completed_idx];
    }

    /* check the result */
    int correct_result = npes + npes / 2;

    if (total_sum != correct_result) {
        shmem_global_exit(1);
    }

    shmem_finalize();
    return 0;
}
```

9.12.7 SHMEM_WAIT_UNTIL_SOME_VECTOR

Wait on an array of variables on the local PE until at least one variable meets the its specified wait condition.

SYNOPSIS

C11:

```
size_t shmem_wait_until_some_vector(TYPE *ivars, size_t nelems, size_t *indices,
    const int *status, int cmp, TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_wait_until_some_vector(TYPE *ivars, size_t nelems, size_t *indices,
    const int *status, int cmp, TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
OUT	*indices*	An array of indices of length at least *nelems* into *ivars* that satisfied the wait condition.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the wait set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with elements of *cmp_values*.

IN	*cmp_values*	An array of length *nelems* containing values to be compared with the respective objects in *ivars*.

API description

The *shmem_wait_until_some_vector* routine waits until at least one entry in the wait set specified by *ivars* and *status* satisfies the wait condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine compares each element of the *ivars* array in the wait set with each respective value in *cmp_values* according to the comparison operator *cmp* at the calling PE. This routine tests all elements of *ivars* in the wait set at least once, and the order in which the elements are waited upon is unspecified.

Upon return, the *indices* array contains the indices of at least one element in the wait set that satisfied the wait condition during the call to *shmem_wait_until_some_vector*. The return value of *shmem_wait_until_some_vector* is equal to the total number of these satisfied elements. For a given return value *N*, the first *N* elements of the *indices* array contain those unique indices that satisfied the wait condition. These first *N* elements of *indices* may be unordered with respect to the corresponding indices of *ivars*. The array pointed to by *indices* must be at least *nelems* long. If an entry *i* in *ivars* within the wait set satisfies the wait condition, a series of calls to *shmem_wait_until_some_vector* must eventually include *i* in the *indices* array.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the wait set. Elements of *status* set to 0 will be included in the wait set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the wait set is empty and this routine returns 0. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the wait set. The *ivars*, *indices*, and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_wait_until_some_vector* does not return before the update of the memory indicated by *ivars* is fully complete.

Return Values

shmem_wait_until_some_vector returns the number of indices returned in the *indices* array. If the wait set is empty, this routine returns 0.

Notes

None.

9.12.8 SHMEM_TEST

Indicate whether a variable on the local PE meets the specified condition.

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 9.

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 9.

DESCRIPTION

Arguments

IN	*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*. The *ivar* object at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE.

Implementations must ensure that *shmem_test* does not return 1 before the update of the memory indicated by *ivar* is fully complete.

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* evaluates 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_atomic_set(&wait_vars[mype], mype, 0);

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

9.12.9 SHMEM_TEST_ALL

Indicate whether all variables within an array of variables on the local PE meet a specified test condition.

SYNOPSIS

C11:

```
int shmem_test_all(TYPE *ivars, size_t nelems, const int *status, int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
int shmem_<TYPENAME>_test_all(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the test set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with *cmp_value*.
IN	*cmp_value*	The value to be compared with the objects pointed to by *ivars*.

API description

The *shmem_test_all* routine indicates whether all entries in the test set specified by *ivars* and *status* have satisfied the test condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine does not block and returns zero if not all entries in *ivars* satisfied the test condition. This routine compares each element of the *ivars* array in the test set with the value *cmp_value* according to the comparison operator *cmp* at the calling PE.

If *nelems* is 0, the test set is empty and this routine returns 1.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the test set. Elements of *status* set to 0 will be included in the test set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the test set is empty and this routine returns 0. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the test set. The *ivars*, *indices*, and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_test_all* does not return 1 before the update of the memory indicated by *ivars* is fully complete.

Return Values

shmem_test_all returns 1 if all variables in *ivars* satisfy the test condition or if *nelems* is 0, otherwise this routine returns 0.

Notes

None.

9.12.10 SHMEM_TEST_ANY

Indicate whether any one variable within an array of variables on the local PE meets a specified test condition.

SYNOPSIS

C11:

```
size_t shmem_test_any(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_test_any(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the test set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with *cmp_value*.
IN	*cmp_value*	The value to be compared with the objects pointed to by *ivars*.

API description

The *shmem_test_any* routine indicates whether any entry in the test set specified by *ivars* and *status* has satisfied the test condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine does not block and returns *SIZE_MAX* if no entries in *ivars* satisfied the test condition. This routine compares each element of the *ivars* array in the test set with the value *cmp_value* according to the comparison operator *cmp* at the calling PE. The order in which these elements are tested is unspecified. If an entry *i* in *ivars* within the test set satisfies the test condition, a series of calls to *shmem_test_any* must eventually return *i*.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the test set. Elements of *status* set to 0 will be included in the test set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the test set is empty and this routine returns *SIZE_MAX*. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the test set. The *ivars* and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_test_any* does not return an index before the update of the memory indicated by the corresponding *ivars* element is fully complete.

Return Values

shmem_test_any returns the index of an element in the *ivars* array that satisfies the test condition. If the test set is empty or no conditions in the test set are satisfied, this routine returns *SIZE_MAX*.

Notes

None.

EXAMPLES

The following *C11* example demonstrates the use of *shmem_test_any* to implement a simple linear barrier synchronization while potentially overlapping communication with computation.

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

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

  int *flags = shmem_calloc(npes, sizeof(int));
  int *status = calloc(npes, sizeof(int));

  for (int i = 0; i < npes; i++)
      shmem_atomic_set(&flags[mype], 1, i);

  int ncompleted = 0;
  size_t completed_idx;

  while (ncompleted < npes) {
      completed_idx = shmem_test_any(flags, npes, status, SHMEM_CMP_EQ, 1);
      if (completed_idx != SIZE_MAX) {
          ncompleted++;
          status[completed_idx] = 1;
      } else {
          /* Overlap some computation here */
      }
  }

  free(status);
  shmem_free(flags);
  shmem_finalize();
  return 0;
}
```

9.12.11 SHMEM_TEST_SOME

Indicate whether at least one variable within an array of variables on the local PE meets a specified test condition.

SYNOPSIS

C11:

```
size_t shmem_test_some(TYPE *ivars, size_t nelems, size_t *indices, const int *status,
    int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_test_some(TYPE *ivars, size_t nelems, size_t *indices,
    const int *status, int cmp, TYPE cmp_value);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
OUT	*indices*	An array of indices of length at least *nelems* into *ivars* that satisfied the test condition.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the test set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with *cmp_value*.
IN	*cmp_value*	The value to be compared with the objects pointed to by *ivars*.

API description

The *shmem_test_some* routine indicates whether at least one entry in the test set specified by *ivars* and *status* satisfies the test condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine does not block and returns zero if no entries in *ivars* satisfied the test condition. This routine compares each element of the *ivars* array in the test set with the value *cmp_value* according to the comparison operator *cmp* at the calling PE. This routine tests all elements of *ivars* in the test set at least once, and the order in which the elements are tested is unspecified. If an entry *i* in *ivars* within the test set satisfies the test condition, a series of calls to *shmem_test_some* must eventually return *i*.

Upon return, the *indices* array contains the indices of the elements in the test set that satisfied the test condition during the call to *shmem_test_some*. The return value of *shmem_test_some* is equal to the total number of these satisfied elements. If the return value is *N*, then the first *N* elements of the *indices* array contain those unique indices that satisfied the test condition. These first *N* elements of *indices* may be unordered with respect to the corresponding indices of *ivars*. The array pointed to by *indices* must be at least *nelems* long. If an entry *i* in *ivars* within the test set satisfies the test condition, a series of calls to *shmem_test_some* must eventually include *i* in the *indices* array.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the test set. Elements of *status* set to 0 will be included in the test set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the test set is empty and this routine returns 0. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the test set. The *ivars*, *indices*, and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_test_some* does not return indices before the updates of the memory indicated by the corresponding *ivars* elements are fully complete.

Return Values

shmem_test_some returns the number of indices returned in the *indices* array. If the test set is empty, this routine returns 0.

Notes

None.

EXAMPLES

The following *C11* example demonstrates the use of *shmem_test_some* to process a simple all-to-all transfer of N data elements via a sum reduction, while potentially overlapping communication with computation. This pattern is similar to the *shmem_test_any* example above, but each while loop iteration may process more than one data item.

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

#define N 100

int main(void)
{
  int total_sum = 0;

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

  int *my_data = malloc(N * sizeof(int));
  int *all_data = shmem_malloc(N * npes * sizeof(int));

  int *flags = shmem_calloc(npes, sizeof(int));
  size_t *indices = calloc(npes, sizeof(size_t));
  int *status = calloc(npes, sizeof(int));

  for (int i = 0; i < N; i++)
      my_data[i] = mype*N + i;

  for (int i = 0; i < npes; i++)
      shmem_put_nbi(&all_data[mype*N], my_data, N, i);

  shmem_fence();

  for (int i = 0; i < npes; i++)
      shmem_atomic_set(&flags[mype], 1, i);

  int ncompleted = 0;

  while (ncompleted < npes) {
      int ntested = shmem_test_some(flags, npes, indices, status, SHMEM_CMP_NE, 0);
      if (ntested > 0) {
          for (int i = 0; i < ntested; i++) {
              for (int j = 0; j < N; j++) {
                  total_sum += all_data[indices[i]*N + j];
              }
              status[indices[i]] = 1;
          }
          ncompleted += ntested;
      } else {
          /* Overlap some computation here */
      }
  }

  /* check the result */
  int M = N * npes - 1;
  if (total_sum != M * (M + 1) / 2) {
      shmem_global_exit(1);
  }

  shmem_finalize();
  return 0;
}
```

9.12.12 SHMEM_TEST_ALL_VECTOR

Indicate whether all variables within an array of variables on the local PE meet the specified test conditions.

SYNOPSIS

C11:

```
int shmem_test_all_vector(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
int shmem_<TYPENAME>_test_all_vector(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the test set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with elements of *cmp_values*.
IN	*cmp_values*	An array of length *nelems* containing values to be compared with the respective objects in *ivars*.

API description

The *shmem_test_all_vector* routine indicates whether all entries in the test set specified by *ivars* and *status* have satisfied the test condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine does not block and returns zero if not all entries in *ivars* satisfied the test conditions. This routine compares each element of the *ivars* array in the test set with each respective value in *cmp_values* according to the comparison operator *cmp* at the calling PE. If *nelems* is 0, the test set is empty and this routine returns 1.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the test set. Elements of *status* set to 0 will be included in the test set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the test set is empty and this routine returns 0. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the test set. The *ivars*, *indices*, and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_test_all_vector* does not return 1 before the update of the memory indicated by *ivars* is fully complete.

Return Values

shmem_test_all_vector returns 1 if all variables in *ivars* satisfy the test conditions or if *nelems* is 0, otherwise this routine returns 0.

Notes

None.

9.12.13 SHMEM_TEST_ANY_VECTOR

Indicate whether any one variable within an array of variables on the local PE meets its specified test condition.

SYNOPSIS

C11:

```
size_t shmem_test_any_vector(TYPE *ivars, size_t nelems, const int *status, int cmp,
    TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_test_any_vector(TYPE *ivars, size_t nelems, const int *status,
    int cmp, TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the test set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with elements of *cmp_values*.
IN	*cmp_values*	An array of length *nelems* containing values to be compared with the respective objects in *ivars*.

API description

The *shmem_test_any_vector* routine indicates whether any entry in the test set specified by *ivars* and *status* has satisfied the test condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine does not block and returns *SIZE_MAX* if no entries in *ivars* satisfied the test condition. This routine compares each element of the *ivars* array in the test set with each respective value in *cmp_values* according to the comparison operator *cmp* at the calling PE. The order in which these elements are tested is unspecified. If an entry *i* in *ivars* within the test set satisfies the test condition, a series of calls to *shmem_test_any_vector* must eventually return *i*.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the test set. Elements of *status* set to 0 will be included in the test set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the test set is empty and this routine returns *SIZE_MAX*. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the test set. The *ivars* and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_test_any_vector* does not return an index before the update of the memory indicated by the corresponding *ivars* element is fully complete.

Return Values

shmem_test_any_vector returns the index of an element in the *ivars* array that satisfies the test condition. If the test set is empty or no conditions in the test set are satisfied, this routine returns *SIZE_MAX*.

Notes

None.

9.12.14 SHMEM_TEST_SOME_VECTOR

Indicate whether at least one variable within an array of variables on the local PE meets its specified test condition.

SYNOPSIS

C11:

```
size_t shmem_test_some_vector(TYPE *ivars, size_t nelems, size_t *indices, const int *status,
    int cmp, TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types specified by Table 9.

C/C++:

```
size_t shmem_<TYPENAME>_test_some_vector(TYPE *ivars, size_t nelems, size_t *indices,
    const int *status, int cmp, TYPE *cmp_values);
```

where *TYPE* is one of the point-to-point synchronization types and has a corresponding *TYPENAME* specified by Table 9.

DESCRIPTION

Arguments

IN	*ivars*	A pointer to an array of remotely accessible data objects.
IN	*nelems*	The number of elements in the *ivars* array.
OUT	*indices*	An array of indices of length at least *nelems* into *ivars* that satisfied the test condition.
IN	*status*	An optional mask array of length *nelems* that indicates which elements in *ivars* are excluded from the test set.
IN	*cmp*	A comparison operator from Table 10 that compares elements of *ivars* with elements of *cmp_values*.
IN	*cmp_values*	An array of length *nelems* containing values to be compared with the respective objects in *ivars*.

API description

The *shmem_test_some_vector* routine indicates whether at least one entry in the test set specified by *ivars* and *status* satisfies the test condition at the calling PE. The *ivars* objects at the calling PE may be updated by an AMO performed by a thread located within the calling PE or within another PE. This routine does not block and returns zero if no entries in *ivars* satisfied the test condition. This routine compares each element of the *ivars* array in the test set with each respective value in *cmp_values* according to the comparison operator *cmp* at the calling PE. This routine tests all elements of *ivars* in the test set at least once, and the order in which the elements are tested is unspecified.

Upon return, the *indices* array contains the indices of the elements in the test set that satisfied the test condition during the call to *shmem_test_some_vector*. The return value of *shmem_test_some_vector* is equal to the total number of these satisfied elements. If the return value is *N*, then the first *N* elements of the *indices* array contain those unique indices that satisfied the test condition. These first *N* elements of *indices* may be unordered with respect to the corresponding indices of *ivars*. The array pointed to by *indices* must be at least *nelems* long. If an entry *i* in *ivars* within the test set satisfies the test condition, a series of calls to *shmem_test_some_vector* must eventually include *i* in the *indices* array.

The optional *status* is a mask array of length *nelems* where each element corresponds to the respective element in *ivars* and indicates whether the element is excluded from the test set. Elements of *status* set to 0 will be included in the test set, and elements set to 1 will be ignored. If all elements in *status* are set to 1 or *nelems* is 0, the test set is empty and this routine returns 0. If *status* is a null pointer, it is ignored and all elements in *ivars* are included in the test set. The *ivars*, *indices*, and *status* arrays must not overlap in memory.

Implementations must ensure that *shmem_test_some_vector* does not return indices before the updates of the memory indicated by the corresponding *ivars* elements are fully complete.

Return Values

shmem_test_some_vector returns the number of indices returned in the *indices* array. If the test set is empty, this routine returns 0.

Notes

None.

9.12.15 SHMEM_SIGNAL_WAIT_UNTIL

Wait for a variable on the local PE to change from a signaling operation.

SYNOPSIS

C/C++:

```
uint64_t shmem_signal_wait_until(uint64_t *sig_addr, int cmp, uint64_t cmp_value);
```

DESCRIPTION

Arguments

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

API description

shmem_signal_wait_until operation blocks until the value contained in the signal data object, *sig_addr*, at the calling PE satisfies the wait condition. In an OpenSHMEM program with single-threaded or multi-threaded PEs, the *sig_addr* object at the calling PE is expected only to be updated as a signal, through the signaling operations available in Section 9.10.3 and Section 9.10.4.

This routine can be used to implement point-to-point synchronization between PEs or between threads within the same PE. A call to this routine blocks until the value of *sig_addr* at the calling PE satisfies the wait condition specified by the comparison operator, *cmp*, and comparison value, *cmp_value*.

Return Values

Return the contents of the signal data object, *sig_addr*, at the calling PE that satisfies the wait condition.

Notes

None.

Note to implementors

Implementations must ensure that *shmem_signal_wait_until* do not return before the update of the memory indicated by *sig_addr* is fully complete. Partial updates to the memory must not cause *shmem_signal_wait_until* to return.

9.13 Memory Ordering Routines

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

9.13.1 SHMEM_FENCE

Assures ordering of delivery of memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, and AMO routines to symmetric data objects.

SYNOPSIS

C/C++:

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

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.

API description

This routine assures ordering of delivery of memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, and AMO routines to symmetric data objects. All memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, and AMO 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 memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, and AMO 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* or values fetched by nonblocking AMO routines. If *ctx* has the value *SHMEM_CTX_INVALID*, no operation is performed.

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 memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, and AMO routines to symmetric data objects which makes the updates visible to all other PEs.

The *shmem_quiet* routine should be called if completion of memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, and AMO 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, *put-with-signal*, 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*.

9.13.2 SHMEM_QUIET

Waits for completion of all outstanding memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, *Get*, and AMO routines to symmetric data objects issued by a PE.

SYNOPSIS

C/C++:

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

DESCRIPTION

Arguments

IN	*ctx*	A context handle specifying the context on which to perform the operation. When this argument is not provided, the operation is performed on the default context.

API description

The *shmem_quiet* routine ensures completion of memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*,*put-with-signal*, *Get*, and AMO routines on symmetric data objects issued by the calling PE on the given context. All memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, *Get*, and AMO routines to symmetric data objects are guaranteed to be completed and visible to all PEs when *shmem_quiet* returns. If *ctx* has the value *SHMEM_CTX_INVALID*, no operation is performed.

Return Values

None.

Notes

shmem_quiet is most useful as a way of ensuring completion of several memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, *Get*, and AMO 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 (memory store, blocking *Put*, AMO, and *put-with-signal*, as well as nonblocking *Put*, *put-with-signal*, *Get*, and AMO 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, *put-with-signal*, 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

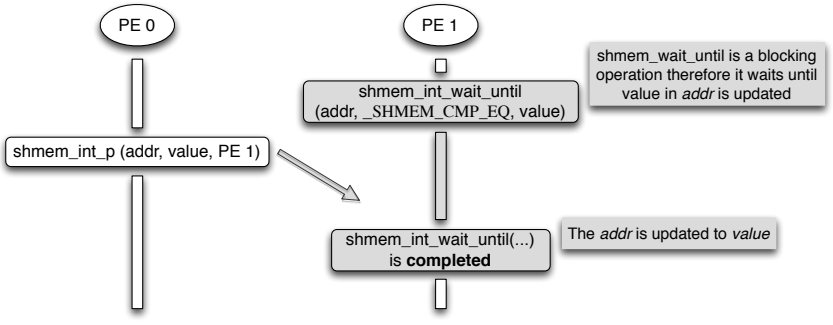
The following example uses *shmem_quiet* in a *C11* program:

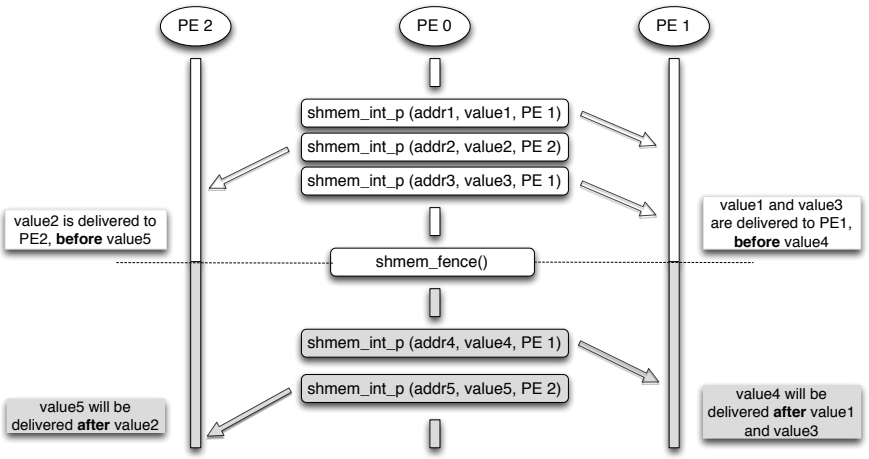
```
#include <stdio.h>
#include <shmem.h>

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;
}
```

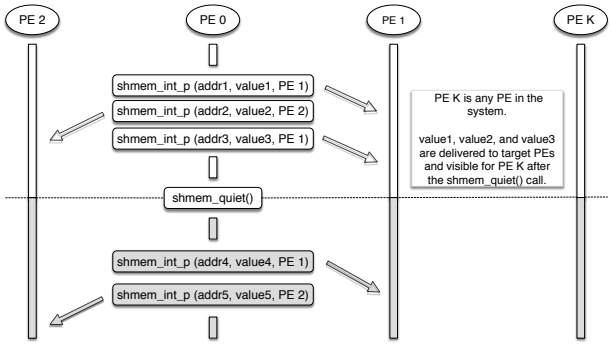
Put1 and *put2* will be completed and visible before *put3* and *put4*.

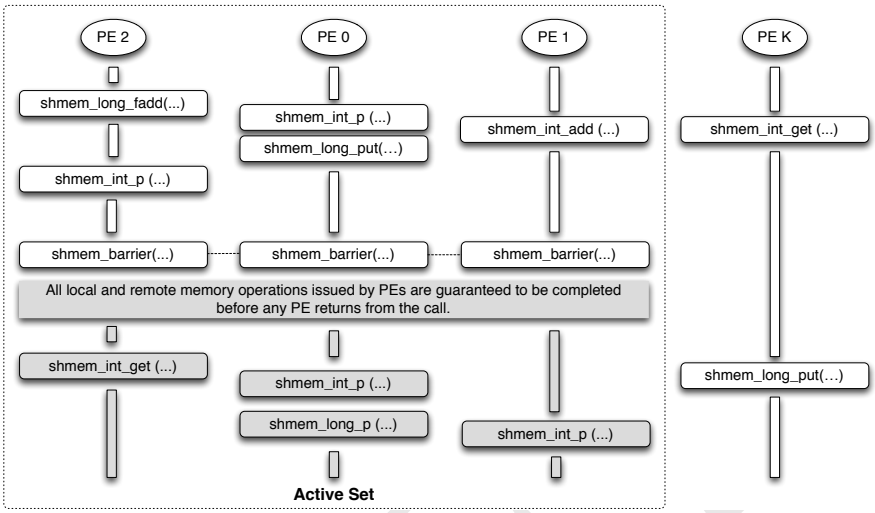
9.13.3 Synchronization and Communication Ordering in OpenSHMEM

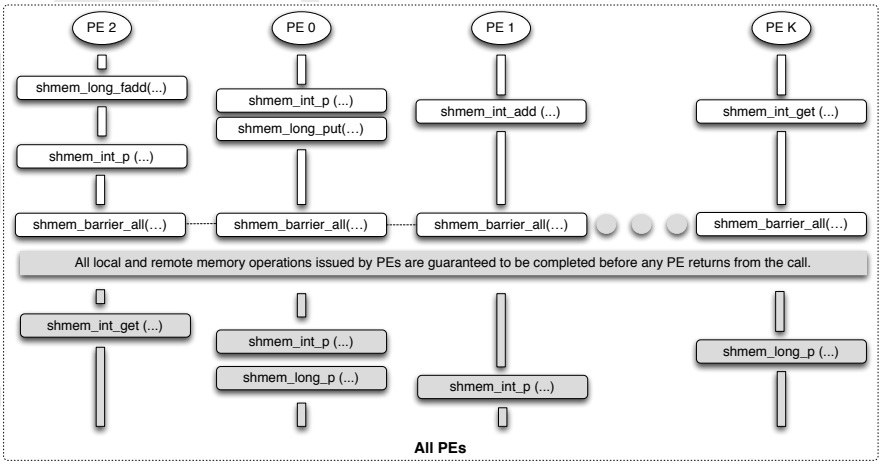
When using the OpenSHMEM API, synchronization, ordering, and completion of communication become critical. The updates via *Put* routines, AMOs, stores, and nonblocking *Put* and *Get* routines on symmetric data cannot be guaranteed until some form of synchronization or ordering is introduced in the user's 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.

9.14 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*.

9.14.1 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);
```

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*.

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_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. The *shmem_clear_lock* routine releases a lock previously set by *shmem_set_lock* or *shmem_test_lock* after performing a quiet operation on the default context to ensure that all symmetric memory accesses that occurred during the critical region are complete. These routines are appropriate for protecting a critical region from simultaneous update by multiple PEs.

The OpenSHMEM lock API provides a non-reentrant mutex. Thus, a call to *shmem_set_lock* or *shmem_test_lock* when the calling PE already holds the given lock will result in undefined behavior. In a multithreaded OpenSHMEM program, the user must ensure that such calls do not occur.

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 must be initialized to zero before any PE performs an OpenSHMEM lock operation on the given variable. Accessing an in-use lock variable using any method other than the OpenSHMEM lock API, e.g. using local load/store, RMA, or AMO operations, results in undefined behavior.

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

EXAMPLES

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

```c
#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_clear_lock(&lock); /* ensures count update has completed before clearing the lock */
   shmem_finalize();
   return 0;
}
```

9.15 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.

9.15.1 SHMEM_CACHE

Controls data cache utilities.

SYNOPSIS

deprecation start

C/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

DESCRIPTION

Arguments

IN	*dest*	A data object that is local to the PE.

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.

10 OpenSHMEM Profiling Interface

The objective of the OpenSHMEM profiling interface is to ensure an easy and flexible usage model for profiling (and other similar) tool developers to interface their codes into OpenSHMEM implementations on different platforms. Since OpenSHMEM is a machine-independent standard with different implementations, it is unreasonable to expect that the authors and developers of profiling tools for OpenSHMEM will have access to the source code that implements OpenSHMEM on any particular machine. It is, therefore, necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

The OpenSHMEM profiling interface places the following requirements on implementations.

1. An OpenSHMEM implementation must provide a mechanism through which all of the OpenSHMEM defined functions may be accessible with a name shift. This requires an alternate entry point name, with the prefix *pshmem_* for each OpenSHMEM function. For OpenSHMEM inlined functions (e.g. macros), it is also required that the *pshmem_* version is supplied although it is not possible to replace the *shmem_* version with a user-defined version at link time.
2. It must be ensured that the OpenSHMEM functions that are not replaced as above, may still be linked into an executable image without causing name clashes.
3. Documentation of the implementation of different language bindings of the OpenSHMEM interface must indicate if they are layered on top of each other. Using this documentation, developers can determine whether they need to implement the profile interface for each binding or not. For example, it must be noted that the OpenSHMEM *C11* type-generic interfaces for different RMA and AMO operations cannot have any equivalent *pshmem_* interfaces because the *C11* type-generic interfaces are implemented as macros.
4. In the case where the implementation of different API feature sets is implemented through a layered approach using "wrapper" functions, the wrapper functions must be kept separate from the rest of the library. This requirement allows the developers to extract these functions from the original OpenSHMEM library and add them into the profiling library without bringing along any other code.
5. A no-op routine, *shmem_pcontrol*, must be provided in the OpenSHMEM library.
6. It must be ensured that any OpenSHMEM types or constants that are needed by the *pshmem_* interfaces are defined in *pshmem.h*.

Provided that an OpenSHMEM implementation meets these requirements, it is possible for the implementor of the profiling system to intercept the OpenSHMEM calls that are made by the user program. The information required can be collected before and after calling the underlying OpenSHMEM implementation through the name shifted entry points.

10.1 Control of Profiling

Any user code must be able to control the profiler dynamically during runtime. Generally, this capability is used for the purposes of

- Enabling and disabling of profiling based on the current state of the execution and calculation,
- Flushing of the trace buffers at non-critical execution regions,
- Adding user events to a trace file.

These functionalities can be achieved through the usage of *shmem_pcontrol*.

10.1.1 SHMEM_PCONTROL

Allows the user to control profiling.

SYNOPSIS

C/C++:

```
void shmem_pcontrol(const int level, ...);
```

DESCRIPTION

Arguments

IN	*level*	The profiling level.

API description

shmem_pcontrol sets the profiling level and any other library defined effects through additional arguments. OpenSHMEM libraries make no use of this routine and simply return immediately to the user code.

Return Values

None.

Notes

Since OpenSHMEM has no control of the implementation of the profiling code, it is impossible to precisely specify the semantics that will be provided by calls to *shmem_pcontrol*. This vagueness extends to the number of arguments to the function and their datatypes. However, to provide some level of portability of user codes to different profiling libraries, the following *level* values are recommended.

- `level <= 0` Profiling is disabled.
- `level == 1` Profiling is enabled at the default level of detail.
- `level == 2` Profiling is enabled and profile buffers are flushed if available.
- `level >= 2` Profiling is enabled with profile library defined effects and additional arguments.

The default state after *shmem_init* is recommended to have profiling enabled at the default level of detail (`level == 1`). This allows users to link with a profiling library and to obtain profile output without having to modify the user-level source code.

10.2 Example Implementations

10.2.1 Profiler

The following example illustrates how a profiler can measure the total and average time spent by the *shmem_long_put* function in the profiling library that intercepts the OpenSHMEM function calls from the user application.

```
#include <stdio.h>
#include <sys/time.h>
#include <pshmem.h>

static double total_put_time = 0.0;
static double avg_put_time = 0.0;
static long put_count = 0;

static inline double get_wtime(void) {
    double wtime = 0.0;
    struct timeval tv;
    gettimeofday(&tv, NULL);
    wtime = tv.tv_sec;
    wtime += (double)tv.tv_usec / 1.0e6;
    return wtime;
}

void shmem_long_put(long *dest, const long *source, size_t nelems, int pe)
{
    double t_start = get_wtime();                                   /* Start timer */
    pshmem_long_put(dest, source, nelems, pe);                  /* Name shifted call to put */
    total_put_time += get_wtime() - t_start;                 /* Calculate total time elapsed */
    put_count += 1;                                                 /* Increment put counts */
    avg_put_time = total_put_time / (double) put_count;     /* Calculate average put latency */

    return;
}
```

10.2.2 OpenSHMEM Library

To implement the name-shift versions of the OpenSHMEM functions, there are various options available. The following two examples present two such options that can be implemented in C on a Unix system. These two options are dependent on whether the linker and compiler support weak symbols.

If the compiler and linker support weak external symbols, then only a single library is required. The following two examples show how the name-shifted requirement can be achieved on such platforms.

Example 1

```
#pragma weak shmem_example = pshmem_example

void pshmem_example(/* appropriate arguments */)
{
    /* function body */
}
```

The effect of the *#pragma* directive is to define the external symbol *shmem_example* as a weak definition that aliases the *pshmem_example* function. This means that the linker will allow another definition of the symbol (e.g. the profiling library may contain an alternate definition). The weak definition is used in the case where no other definition for the same function exists.

Example 2

```
void pshmem_example(/* appropriate arguments */)
{
    /* function body */
}

void shmem_example(/* appropriate arguments */) __attribute__ ((weak, alias("pshmem_example")));
```

In this example, the keyword *__attribute__* is used to declare the *shmem_example* function as an alias for the original function, *pshmem_example*.

In the absence of weak symbols, one possible solution would be to use the C macro preprocessor as shown in the following example.

```
#ifdef BUILD_PSHMEM_INTERFACES
#    define SHFN(fn) p##fn
#else
#    define SHFN(fn) fn
#endif
```

Each of the user-defined functions in the profiling library would then be declared in the following manner.

```
void SHFN(shmem_example)(/* appropriate arguments */)
{
    /* function body */
}
```

The same source file can then be compiled to produce both versions of the library, depending on the state of the *BUILD_PSHMEM_INTERFACES* macro symbol.

10.3 Limitations

10.3.1 Multiple Counting

Since some functions in OpenSHMEM library may be implemented using more basic OpenSHMEM functions, it is possible for these basic profiling functions to be called from within an OpenSHMEM function that was originally called from a profiling routine. For example, OpenSHMEM collective operations can be implemented using basic point-to-point operations. Thus, profiling such a collective operation may lead to counting a profiling function for a point-to-point operation more than once after being called from the collective function. It is the developer's responsibility to ensure the profiling application does not count a function more than once if that effect is not intended. For a single-threaded profiler, this can be achieved through a static variable counting the number of times a function has been profiled. In a multi-threaded environment, additional synchronizations are needed to manage updates to this counter and thus, it becomes more complex to accurately profile the OpenSHMEM functions.

10.3.2 Separate Build and Link

To build the profiling tool with both the default OpenSHMEM functions as well as the OpenSHMEM functions to be intercepted, developers must build the multiple instances of the OpenSHMEM functions separately and link them to provide all the definitions. This is necessary so that the developers of the profiling library need only to define those OpenSHMEM functions that they wish to intercept; references to any other functions will be fulfilled by the default OpenSHMEM library. The link step can be summarized as follows.

```
% cc ...  -lmyprof -lpsma -lsma
```

Here, `libmyprof.a` contains the profiler functions that intercept the OpenSHMEM functions to be profiled, `libpsma.a` contains the name-shifted OpenSHMEM function definitions, and `libsma.a` contains the default OpenSHMEM function definitions.

10.3.3 *C11* Type-Generic Interfaces

OpenSHMEM provides type-generic interfaces through *C11* generic selection. These interfaces are defined as macros and are mapped to *C* interface bindings. As a result, the *C11* type-generic interfaces cannot be intercepted and name-shifted *pshmem_* routines are not provided for these bindings. Furthermore, because no two associations in a *C11* *_Generic* selection expression can contain compatible types, the type name of the *C* operation that is invoked may not be identical to the type name of the original call's arguments (e.g. *int32_t* may map to *int*).

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*.
- 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
```

The example in Listing A.3 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.3: Example program with symmetric data objects

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

```
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++**.

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

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, Silicon Graphics International (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 E

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)
C/C++: shmem_barrier	1.5	Current	*shmem_quiet*; *shmem_sync*
C/C++: Active set based *shmem_sync*	1.5	Current	Team based *shmem_sync*
C/C++: shmem_broadcast[32,64]	1.5	Current	*shmem_broadcast*
C/C++: shmem_collect[32,64]	1.5	Current	*shmem_collect*
C/C++: shmem_fcollect[32,64]	1.5	Current	*shmem_fcollect*
*C/C++: shmem*_TYPENAME_*OP_to_all*	1.5	Current	*shmem*_TYPENAME_*OP_reduce*
C/C++: shmem_alltoall[32,64]	1.5	Current	*shmem_alltoall*
C/C++: shmem_alltoalls[32,64]	1.5	Current	*shmem_alltoalls*

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 in OpenSHMEM 1.4 and removed in OpenSHMEM 1.5 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].

2.14 Active-set-based collective routines

With the addition of OpenSHMEM teams, the previous methods for performing collective operations has been superseded by a more readable, flexible method for organizing and communicating between groups of PEs. All collective routines which previously indicated subgroups of PEs with a list of parameters to describe the subgroup composition should be phased out in favor of using collective operations with a team parameter.

When moving from active set routines to teams based routines, the fixed-size versions of the routines, e.g. *shmem_broadcast32*, were not carried forward. Instead, all teams based collective routines use standard *C* types with the option to use generic *C11* functions for more portable and maintainable implementations.

2.15 *C/C++*: *shmem_barrier*

Each OpenSHMEM team might be associated with some number of communication contexts. The *shmem_barrier* functions imply that the default context is quiesced after synchronizing some set of PEs. Since teams may have some

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

number of contexts associated with the team, it becomes less clear which context would be the "default" context for that particular team. Rather than continue to support *shmem_barrier* for active-sets or teams, programs should use a call to *shmem_quiet* followed by a call to *shmem_sync* in order to explicitly indicate which context to quiesce.

Annex F

Changes to this Document

1 Version 1.5

Major changes in OpenSHMEM 1.5 include . . .

The following list describes the specific changes in OpenSHMEM 1.5:

- Added support for nonblocking AMO functions.
  See Section 9.9.
- Added support for blocking *put-with-signal* functions.
  See Section 9.10.3.
- Added support for nonblocking *put-with-signal* functions.
  See Section 9.10.4.
- Clarified that point-to-point synchronization routines preserve the atomicity of OpenSHMEM AMOs.
  See Section 3.1.
- Clarified that symmetric variables used as *ivar* arguments to point-to-point synchronization routines must be updated using OpenSHMEM AMOs.
  See Section 9.12.
- Removed the entire OpenSHMEM *Fortran* API.
- Added support for multipliers in *SHMEM_SYMMETRIC_SIZE* environment variables.
  See Section 8.
- Added support for a multiple-element point-to-point synchronization API with the functions: *shmem_wait_until_all*, *shmem_wait_until_any*, *shmem_wait_until_some*, *shmem_test_all*, *shmem_test_any*, and *shmem_test_some*.
  See Sections 9.12.2, 9.12.3, 9.12.4, 9.12.9, 9.12.10, and 9.12.11.
- Added support for vectorized comparison values in the multiple-element point-to-point synchronization API with the functions: *shmem_wait_until_all_vector*, *shmem_wait_until_any_vector*, *shmem_wait_until_some_vector*, *shmem_test_all_vector*, *shmem_test_any_vector*, and *shmem_test_some_vector*.
  See Sections 9.12.5, 9.12.6, 9.12.7, 9.12.12, 9.12.13, and 9.12.14.
- Added OpenSHMEM profiling interface.
  See Section 10.
- Specified the validity of communication contexts, added the constant *SHMEM_CTX_INVALID*, and clarified the behavior of *shmem_ctx_** routines on invalid contexts.
  See Section 9.5.

- Clarified PE active set requirements.
  See Section 9.11.

- Clarified that when the *size* argument is zero, symmetric heap allocation routines perform no action and return a null pointer; that symmetric heap management routines that perform no action do not perform a barrier; and that the *alignment* argument to *shmem_align* must be power of two multiple of *sizeof(void*)*.
  See Section 9.3.1.

- Clarified that the OpenSHMEM lock API provides a non-reentrant mutex and that *shmem_clear_lock* performs a quiet operation on the default context.
  See Section 9.14.1

- Clarified the atomicity guarantees of the OpenSHMEM memory model.
  See Section 3.1.

2 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 9.5.

- New API *shmem_sync_all* and *shmem_sync* to provide PE synchronization without completing pending communication operations.
  See Sections 9.11.4 and 9.11.3.

- 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 9.6.4 and 9.7.2.

- 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 9.12.1.

- 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 9.12.1 and 9.14.1.

- 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 E.
- Deprecated header directory *mpp*.
  See Section E.
- Deprecated the *shmem_wait* functions and the *long*-typed *C/C++ shmem_wait_until* function.
  See Section 9.12.
- Added the *shmem_test* functions.
  See Section 9.12.
- 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 9.13.2
- Revised the description of *shmem_barrier_all* for a multithreaded OpenSHMEM PE.
  See Section 9.11.1
- Revised the description of *shmem_wait* for a multithreaded OpenSHMEM PE.
  See Section 9.12.1
- 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 4, 5, 6, and 9
- Renamed AMO operations to use *shmem_atomic_** prefix and deprecated old AMO routines.
  See Section 9.8.
- Added fetching and non-fetching bitwise AND, OR, and XOR atomic operations.
  See Section 9.8.
- 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 9.11.7.
- Clarified that complex-typed reductions in C are optionally supported.
  See Section 9.11.7.

3 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 9.6, 9.7 and 9.8.
- New nonblocking variants of remote memory access, *SHMEM_PUT_NBI* and *SHMEM_GET_NBI*.
  See Sections 9.7.1 and 9.7.2.
- New atomic elemental read and write operations, *SHMEM_FETCH* and *SHMEM_SET*.
  See Sections 9.8.1 and 9.8.2
- New alltoall data exchange operations, *SHMEM_ALLTOALL* and *SHMEM_ALLTOALLS*.
  See Sections 9.11.8 and 9.11.9.
- Added *volatile* to remotely accessible pointer argument in *SHMEM_WAIT* and *SHMEM_LOCK*.
  See Sections 9.12.1 and 9.14.1.
- Deprecation of *SHMEM_CACHE*.
  See Section 9.15.1.

4 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 9.6.1.

- Clarification related to *shmem_wait*.
  See Section 9.12.1.

- 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 E.

5 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 9.13.

- Clarification of the completion semantics of memory load and store operations in context of *shmem_barrier_all* and *shmem_barrier* routines.
  See Section 9.11.1 and 9.11.2.

- Clarification of the completion and ordering semantics of *shmem_quiet* and *shmem_fence*.
  See Section 9.13.2 and 9.13.1.

- Clarifications of the completion semantics of RMA and AMO routines.
  See Sections 9.6 and 9.8

- 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 9.12.1.
- Added API calls: *shmem_char_p*, *shmem_char_g*.
  See Sections 9.6.2 and 9.6.5.
- Removed API calls: *shmem_char_put*, *shmem_char_get*.
  See Sections 9.6.1 and 9.6.4.
- The usage of *ptrdiff_t*, *size_t*, and *int* in the interface signature was made consistent with the description.
  See Sections 9.11, 9.6.3, and 9.6.6.
- Revised *shmem_barrier* example.
  See Section 9.11.2.
- Clarification of the initial value of *pSync* work arrays for *shmem_barrier*.
  See Section 9.11.2.
- 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 9.8.6.
- 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 9.8.8, 9.8.3, 9.8.4, 9.8.5, 9.8.6, and 9.8.7.
- 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 9.11.7 and 9.12.1.

- Clarifications of the reuse *pSync* and *pWork* across collectives.
  See Sections 9.11, 9.11.5, 9.11.6 and 9.11.7.

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

Index