Sponsored by

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

Authors and Collaborators

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

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

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

1. The OpenSHMEM Effort ................................................................. 1
2. Programming Model Overview ...................................................... 1
3. Memory Model ........................................................................ 3
4. Execution Model ..................................................................... 4
   4.1 Progress of OpenSHMEM Operations ....................................... 4
   4.2 Atomicity Guarantees ............................................................ 4
5. Language Bindings and Conformance ............................................ 5
6. Library Constants .................................................................... 5
7. Environment Variables .............................................................. 6
8. OpenSHMEM Library API ............................................................ 7
   8.1 Library Setup, Exit, and Query Routines ................................. 7
   8.1.1 SHMEM_INIT ................................................................. 7
   8.1.2 SHMEM_MY_PE ............................................................ 8
   8.1.3 SHMEM_N_PES ............................................................ 9
   8.1.4 SHMEM_FINALIZE ....................................................... 10
   8.1.5 SHMEM_GLOBAL_EXIT ............................................... 11
   8.1.6 SHMEM_PE_ACCESSIBLE .................. .............................. 12
   8.1.7 SHMEM_ADDR_ACCESSIBLE ....................................... 13
   8.1.8 SHMEM_PTR ............................................................... 14
   8.1.9 SHMEM_INFO_GET_VERSION ....................................... 15
   8.1.10 SHMEM_INFO_GET_NAME ......................................... 16
   8.1.11 START_PES ............................................................... 17
   8.2 Memory Management Routines .............................................. 18
   8.2.1 SHMEM_MALLOC, SHMEM_FREE, SHMEM_REALLOC, SHMEM_ALIGN 18
   8.2.2 SHPALLOC ................................................................. 19
   8.2.3 SHPCLMOVE ............................................................. 20
   8.2.4 SHPDEALLOC ............................................................. 21
   8.3 Remote Memory Access Routines ............................................ 22
   8.3.1 SHMEM_PUT ............................................................... 22
   8.3.2 SHMEM_P ................................................................. 24
   8.3.3 SHMEM_IPUT ............................................................. 25
   8.3.4 SHMEM_GET ............................................................. 27
   8.3.5 SHMEM_G ................................................................. 29
   8.3.6 SHMEM_IGET ............................................................ 30
   8.4 Non-blocking Remote Memory Access Routines ....................... 32
   8.4.1 SHMEM.PUT.NBI ....................................................... 32
   8.4.2 SHMEM.GET.NBI ...................................................... 34
   8.5 Atomic Memory Operations .................................................... 35
   8.5.1 SHMEM_ADD ............................................................. 36
   8.5.2 SHMEM_CSWAP ........................................................ 37
   8.5.3 SHMEM_SWAP .......................................................... 39
   8.5.4 SHMEM_FINC ............................................................ 40
   8.5.5 SHMEM_FDEC ........................................................... 42
A Writing OpenSHMEM Programs

B Compiling and Running Programs
  1 Compilation
  2 Running Programs

C Undefined Behavior in OpenSHMEM

D Interoperability with other Programming Models
  1 Message Passing Interface (MPI) Interoperability

E History of OpenSHMEM

F OpenSHMEM Specification and Deprecated API

G Changes to this Document
  1 Version 1.3
  2 Version 1.2
  3 Version 1.1
1. The OpenSHMEM Effort

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

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

2. Programming Model Overview

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

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

An overview of the OpenSHMEM routines is described below:

1. Library Setup and Query

   (a) **Initialization**: The OpenSHMEM library environment is initialized.

   (b) **Query**: The local PE may get 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 SGI.
3. MEMORY MODEL

(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. 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.

4. 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) **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.

(e) **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.

(f) **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.

5. 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 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.

6. 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.

7. 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.

8. Data Cache Control (deprecated)

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

An OpenSHMEM program consists of data objects that are private to each PE and data objects that are remotely accessible by all PEs. Private data objects are stored in the local memory of each PE and can only be accessed by the PE itself; these data objects cannot be accessed by other PEs via OpenSHMEM routines. Private data objects follow the memory model of C or Fortran. Remotely accessible objects, however, can be accessed by remote PEs using OpenSHMEM routines. Remotely accessible data objects are called Symmetric Data Objects. Each symmetric data object has a corresponding object with the same name, type, and size on all PEs where that object is accessible via the OpenSHMEM API\(^2\). (For the definition of what is accessible, see the descriptions for shmem_pe_accessible and shmem_addr_accessible in sections 8.1.6 and 8.1.7.) Symmetric data objects accessed via typed OpenSHMEM interfaces are required to be natural aligned based on their type requirements and underlying architecture. In OpenSHMEM the following kinds of data objects are symmetric:

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

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

\(^2\)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 8.2.1
4 Execution Model

An OpenSHMEM program consists of a set of OpenSHMEM processes called PEs that execute in a SPMD-like model where each PE can take a different execution path. For example, a PE can be implemented using an OS process. 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`\(^3\) before using any of the other OpenSHMEM library routines. An OpenSHMEM program finishes execution by returning from the main routine or when any PE calls `shmem_global_exit`. When returning from main, OpenSHMEM must complete all pending communication and release all the resources associated to the library using an implicit collective synchronization across PEs. The user has the option to call `shmem_finalize` (before returning from main) to complete all pending communication and release all the OpenSHMEM library resources without terminating the program. Calling any OpenSHMEM routine after `shmem_finalize` leads to undefined behavior.

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

4.1 Progress of OpenSHMEM Operations

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

Note to implementors:

- An OpenSHMEM implementation for hardware that does not provide asynchronous communication capabilities may require a software progress thread in order to process remotely-issued communication requests without explicit program calls to the OpenSHMEM library.

- High performance implementations of OpenSHMEM are expected to leverage hardware offload capabilities and provide asynchronous one-sided communication without software assistance.

- Implementations should avoid deferring the execution of one-sided operations until a synchronization point where data is known to be available. High-quality implementations should attempt asynchronous delivery whenever possible, for performance reasons. Additionally, the OpenSHMEM community discourages releasing OpenSHMEM implementations that do not provide asynchronous one-sided operations, as these have very limited performance value for OpenSHMEM programs.

4.2 Atomicity Guarantees

OpenSHMEM contains a number of routines that operate on symmetric data atomically (Section 8.5). These routines guarantee that accesses by OpenSHMEM’s atomic operations with the same datatype will be exclusive, but do not guarantee exclusivity in combination with other routines, either inside OpenSHMEM or outside.

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

\(^3\)\texttt{start\_pes} has been deprecated as of Specification 1.2
5 Language Bindings and Conformance

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

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

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

6 Library Constants

The constants that start with SHMEM_* are for both Fortran and C/C++, and they are compile-time constants. All constants that start with _SHMEM_* are deprecated and provided for backwards compatibility.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++/Fortran: SHMEM_BCAST_SYNC_SIZE</td>
<td>Length of the pSync arrays needed for broadcast routines. The value of this constant is implementation specific. Refer to the Broadcast Routines section under Library Routines for more information about the usage of this constant.</td>
</tr>
<tr>
<td>C/C++/Fortran: SHMEM_SYNC_VALUE</td>
<td>The value used to initialize the elements of pSync arrays. The value of this constant is implementation specific.</td>
</tr>
<tr>
<td>C/C++/Fortran: SHMEM_REDUCE_SYNC_SIZE</td>
<td>Length of the work arrays needed for reduction routines. The value of this constant is implementation specific. Refer to the Reduction Routines section under Library Routines for more information about the usage of this constant.</td>
</tr>
<tr>
<td>C/C++/Fortran: SHMEM_BARRIER_SYNC_SIZE</td>
<td>Length of the work array needed for barrier routines. The value of this constant is implementation specific. Refer to the Barrier Synchronization Routines section under Library Routines for more information about the usage of this constant.</td>
</tr>
<tr>
<td>C/C++/Fortran: SHMEM_COLLECT_SYNC_SIZE</td>
<td>Length of the work array needed for collect routines. The value of this constant is implementation specific. Refer to the Collect Routines section under Library Routines for more information about the usage of this constant.</td>
</tr>
<tr>
<td>C/C++/Fortran: SHMEM_ALLTOALL_SYNC_SIZE</td>
<td>Length of the work array needed for shmem_alltoall routines. The value of this constant is implementation specific. Refer to the Alltoall routines sections under Library Routines for more information about the usage of this constant.</td>
</tr>
</tbody>
</table>
7. ENVIRONMENT VARIABLES

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMA_VERSION</td>
<td>any</td>
<td>print the library version at start-up</td>
</tr>
<tr>
<td>SMA_INFO</td>
<td>any</td>
<td>print helpful text about all these environment variables</td>
</tr>
<tr>
<td>SMA_SYMMETRIC_SIZE</td>
<td>non-negative integer</td>
<td>number of bytes to allocate for symmetric heap</td>
</tr>
<tr>
<td>SMA_DEBUG</td>
<td>any</td>
<td>enable debugging messages</td>
</tr>
</tbody>
</table>

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.
8 OpenSHMEM Library API

8.1 Library Setup, Exit, and Query Routines

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

8.1.1 SHMEM_INIT

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

SYNOPSIS

C/C++:

    void shmem_init(void);

FORTRAN:

    CALL SHMEM_INIT()

DESCRIPTION

Arguments
None.

API description

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

Return Values

None.

Notes

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

EXAMPLES

This is a simple program that calls `shmem_init`:

```
PROGRAM PUT
    INCLUDE "shmem.fh"

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

    IF (SHMEM_MY_PE() .EQ. 0) THEN
```
SRC = 33
CALL SHMEM_INTEGER_PUT(TARG, SRC, 1, RECEIVER)
ENDIF
CALL SHMEM_BARRIER_ALL ! SYNCHRONIZES SENDER AND RECEIVER
IF (SHMEM_MY_PE() .EQ. RECEIVER) THEN
  PRINT*, 'PE', SHMEM_MY_PE(), ' TARG=',TARG,' (expect 33)'
ENDIF
CALL SHMEM_FINALIZE()
END

8.1.2 SHMEM_MY_PE

Returns the number of the calling PE.

SYNOPSIS

C/C++:
int shmem_my_pe(void);

FORTRAN:
INTEGER SHMEM_MY_PE, ME
ME = SHMEM_MY_PE()

DESCRIPTION

Arguments
None.

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

Return Values
Integer - Between 0 and npes - 1

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

EXAMPLES

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

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

int main(void)
{
  int me;
```
8. OPENSHMEM LIBRARY API

```c
shmem_init();
me = shmem_my_pe();
printf("My PE id is: \d\n", me);

return 0;
```

8.1.3 SHMEM_N_PES

Returns the number of PEs running in a program.

SYNOPSIS

**C/C++:**

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

**FORTRAN:**

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

DESCRIPTION

Arguments

None.

API description

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

Return Values

Integer - Number of PEs running in the OpenSHMEM program.

Notes

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

EXAMPLES

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

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

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

    return 0;
}
```
8.1.4 SHMEM_FINALIZE

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

SYNOPSIS

C/C++:

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

FORTRAN:

```fortran
CALL SHMEM_FINALIZE()
```

DESCRIPTION

Arguments

None.

API description

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

Return Values

None.

Notes

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

EXAMPLES

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

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

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

8.1.5 SHMEM_GLOBAL_EXIT

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

SYNOPSIS

C/C++:
void shmem_global_exit(int status);

FORTRAN:
INTEGER STATUS
CALL SHMEM_GLOBAL_EXIT(status)

DESCRIPTION

Arguments

IN status The exit status from the main program.

API description

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

Return Values

None.

Notes

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

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

int main(void)
{
    int me, npes;
    shmem_init();
    me = shmem_my_pe();
    npes = shmem_n_pes();
    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);
    }
    return 0;
}
```

### 8.1.6 SHMEM\_PE\_ACCESSIBLE

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

#### SYNOPSIS

**C/C++:**

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

**FORTRAN:**

```fortran
LOGICAL LOG, SHMEM\_PE\_ACCESSIBLE
INTEGER pe
LOG = SHMEM\_PE\_ACCESSIBLE(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 `TRUE` only if the remote PE is a process running from the same executable file as the local PE, indicating that full OpenSHMEM support for symmetric data objects (that reside in the static memory and symmetric heap) is available, otherwise it returns `FALSE`. This routine may be particularly useful for hybrid programming with other communication libraries (such as a MPI) or parallel languages. For example, on SGI Altix series systems, OpenSHMEM is supported across multiple partitioned hosts and InfiniBand connected hosts. When running multiple executable MPI programs using OpenSHMEM on an Altix, full OpenSHMEM support is available between processes running from the same executable file. However, OpenSHMEM support between processes of
different executable files is supported only for data objects on the symmetric heap, since static data objects are not symmetric between different executable files.

**Return Values**

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

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

**Notes**

None.

### 8.1.7 SHMEM_ADDR_ACCESSIBLE

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

**SYNOPSIS**

**C/C++**:  
```c
int shmem_addr_accessible(const void *addr, int pe);
```

**FORTRAN**:  
```fortran
LOGICAL LOG, SHMEM_ADDR_ACCESSIBLE
INTEGER pe
LOG = SHMEM_ADDR_ACCESSIBLE(addr, pe)
```

**DESCRIPTION**

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>addr</td>
<td>Data object on the local PE.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>Integer id of a remote PE.</td>
</tr>
</tbody>
</table>

**API description**

`shmem_addr_accessible` is a query routine that indicates whether a local address is accessible via OpenSHMEM routines from the specified remote PE.

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

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

**Return Values**

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

Notes
None.

8.1.8 SHMEM_PTR

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

SYNOPSIS

C/C++:
void *shmem_ptr(const void *dest, int pe);

FORTRAN:
POINTER (PTR, POINTEE)
INTEGER pe
PTR = SHMEM_PTR(dest, 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. If you are using Fortran, it must be a default integer value.

API description
shmem_ptr returns an address that may be used to directly reference dest on the specified PE. This address can be assigned to a pointer. After that, ordinary loads and stores to this remote address may be performed. When a sequence of loads (gets) and stores (puts) to a data object on a remote PE does not match the access pattern provided in an OpenSHMEM data transfer routine like shmem_put32 or shmem_real_iget, the shmem_ptr routine can provide an efficient means to accomplish the communication.

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

Notes
When calling shmem_ptr, dest is the address of the referenced symmetric data object on the calling PE.

EXAMPLES

This Fortran program calls shmem_ptr and then PE 0 writes to the BIGD array on PE 1:

```fortran
PROGRAM REMOTWRITE
INCLUDE "shmem.fh"

INTEGER BIGD(100)
SAVE BIGD

INTEGER POINTEE(*)
POINTER (PTR, POINTEE)
```
CALL SHMEM_INIT()

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

CALL SHMEM_BARRIER_ALL

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

END

This is the equivalent program written in C:

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

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

    shmem_init();

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

    shmem_barrier_all();

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

### 8.1.9 SHMEM_INFO_GET_VERSION

Returns the major and minor version of the library implementation.

**SYNOPSIS**

**C/C++:**

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

**FORTRAN:**
8. OPENSHMEM LIBRARY API

DESCRIPTION

Arguments

OUT major The major version of the OpenSHMEM standard in use.
OUT minor The minor version of the OpenSHMEM standard in use.

API description

This routine returns the major and minor version of the OpenSHMEM standard in use. For a given library implementation, the major and minor version returned by these calls is consistent with the compile-time constants, SHMEM_MAJOR_VERSION and SHMEM_MINOR_VERSION, defined in its shmem.h.

Return Values

None.

Notes

None.

8.1.10 SHMEM_INFO_GET_NAME

This routine returns the vendor defined character string.

SYNOPSIS

C/C++:

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

FORTRAN:

```fortran
CHARACTER (*) NAME
SHMEM_INFO_GET_NAME(NAME)
```

DESCRIPTION

Arguments

OUT name The vendor defined string.

API description

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

Return Values
None.

Notes
None.

8.1.11 START_PES

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

SYNOPSIS

C/C++:
```c
void start_pes(int npes);
```

FORTRAN:
```fortran
CALL START_PES(npes)
```

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` before calling any other OpenSHMEM routine.

Return Values

None.

Notes

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

As of OpenSHMEM Specification 1.2 the use of `start_pes` has been deprecated. Although OpenSHMEM libraries are required to support the call, program users are encouraged to use `shmem_init` instead.

EXAMPLES

This is a simple program that calls `start_pes`:
```fortran
PROGRAM PUT
INCLUDE "shmem.fh"

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

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

CALL SHMEM_BARRIER_ALL  ! SYNCHRONIZES SENDER AND RECEIVER

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

8.2 Memory Management Routines

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

8.2.1 SHMEM_MALLOC, SHMEM_FREE, SHMEM_REALLOC, SHMEM_ALIGN

Symmetric heap memory management routines.

SYNOPSIS

C/C++:

```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

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>The size, in bytes, of a block to be allocated from the symmetric heap. This argument is of type size_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>ptr</td>
<td>Points to a block within the symmetric heap.</td>
</tr>
<tr>
<td>IN</td>
<td>alignment</td>
<td>Byte alignment of the block allocated from the symmetric heap.</td>
</tr>
</tbody>
</table>

API description

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

The `shmem_align` routine allocates a block in the symmetric heap that has a byte alignment specified by the alignment argument.

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

The `shmem_realloc` routine changes the size of the block to which `ptr` points to the size (in bytes) specified by `size`. The contents of the block are unchanged up to the lesser of the new and old sizes. If the new size is larger, the 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_free`, and `shmem_realloc` routines are provided so that multiple PEs in a program can allocate symmetric, remotely accessible memory blocks. These memory blocks can then be used with OpenSHMEM communication routines. Each of these routines call the `shmem_barrier_all` routine before returning; this ensures that all PEs participate in the memory allocation, and that the memory on
other PEs can be used as soon as the local PE returns. The user is responsible for calling these routines with identical argument(s) on all PEs; if differing size arguments are used, the behavior of the call and any subsequent OpenSHMEM calls becomes undefined.

Return Values
The \texttt{shmem\_malloc} routine returns a pointer to the allocated space; otherwise, it returns a \texttt{NULL} pointer. The \texttt{shmem\_free} routine returns no value. The \texttt{shmem\_realloc} routine returns a pointer to the allocated space (which may have moved); otherwise, it returns a null pointer. The \texttt{shmem\_align} routine returns an aligned pointer to the allocated space; otherwise, it returns a \texttt{NULL} pointer.

Notes
As of Specification 1.2 the use of \texttt{shmalloc}, \texttt{shmemalign}, \texttt{shfree}, and \texttt{shrealloc} has been deprecated. Although OpenSHMEM libraries are required to support the calls, program users are encouraged to use \texttt{shmem\_malloc}, \texttt{shmem\_align}, \texttt{shmem\_free}, and \texttt{shmem\_realloc} instead. The behavior and signature of the routines remains unchanged from the deprecated versions. The total size of the symmetric heap is determined at job startup. One can adjust the size of the heap using the \texttt{SMA\_SYMMETRIC\_SIZE} environment variable (where available). The \texttt{shmem\_malloc}, \texttt{shmem\_free}, and \texttt{shmem\_realloc} routines differ from the private heap allocation routines in that all PEs in a program must call them (a barrier is used to ensure this).

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

8.2.2 SHPALLOC
Allocates a block of memory from the symmetric heap.

SYNOPSIS

\begin{verbatim}
FORTRAN: 
  POINTER (addr, A(1))
  INTEGER length, errcode, abort
  CALL SHPALLOC(addr, length, errcode, abort)
\end{verbatim}

DESCRIPTION

Arguments

\begin{tabular}{ll}
  \textbf{OUT} & \textit{addr} & First word address of the allocated block. \\
  \textbf{IN} & \textit{length} & Number of words of memory requested. One word is 32 bits. \\
  \textbf{OUT} & \textit{errcode} & Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error. \\
  \textbf{IN} & \textit{abort} & Abort code; nonzero requests abort on error; 0 requests an error code.
\end{tabular}
API description

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

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

Return Values

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Length is not an integer greater than 0</td>
</tr>
<tr>
<td>-2</td>
<td>No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).</td>
</tr>
</tbody>
</table>

Notes

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

Note to implementors

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

8.2.3 SHPCLMOVE

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

SYNOPSIS

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

DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>INOUT</th>
<th>addr</th>
<th>On entry, first word address of the block to change; on exit, the new address of the block if it was moved.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>length</td>
<td>Requested new total length in words. One word is 32 bits.</td>
</tr>
<tr>
<td>OUT</td>
<td>status</td>
<td>Status is 0 if the block was extended in place, 1 if it was moved, and a negative integer for the type of error detected.</td>
</tr>
<tr>
<td>IN</td>
<td>abort</td>
<td>Abort code. Nonzero requests abort on error; 0 requests an error code.</td>
</tr>
</tbody>
</table>
API description
The SHPCLMOVE routine either extends a symmetric heap block if the block is followed by a large enough free block or copies the contents of the existing block to a larger block and returns a status code indicating that the block was moved. This routine also can reduce the size of a block if the new length is less than the old length. All PEs in a program must call SHPCLMOVE with the same value of addr to maintain symmetric heap consistency; if any PEs are missing, the program hangs.

Return Values

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

Notes
None.

8.2.4 SHPDEALLOC

Returns a memory block to the symmetric heap.

SYNOPSIS

FORTRAN:

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

DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>IN</td>
<td>First word address of the block to deallocate.</td>
</tr>
<tr>
<td>errcode</td>
<td>OUT</td>
<td>Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error.</td>
</tr>
<tr>
<td>abort</td>
<td>IN</td>
<td>Abort code. Nonzero requests abort on error; 0 requests an error code.</td>
</tr>
</tbody>
</table>

API description

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

Return Values

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Length is not an integer greater than 0</td>
</tr>
</tbody>
</table>
No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the symmetric heap).

Address is outside the bounds of the symmetric heap.

Block is already free.

Address is not at the beginning of a block.

**Notes**

None.

### 8.3 Remote Memory Access Routines

The *Remote Memory Access* (RMA) routines described in this section are one-sided communication mechanisms of the OpenSHMEM API. While using these mechanisms, the user is required to provide parameters only on the calling side. A characteristic of one-sided communication is that it decouples communication from the synchronization. One-sided communication mechanisms transfer the data but do not synchronize the sender of the data with the receiver of the data.

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

Where appropriate compiler support is available, OpenSHMEM provides type-generic one-sided communication interfaces via C11 generic selection (C11 §6.5.1.1) for block, scalar, and block-strided put and get communication. Such type-generic routines are supported for the “standard RMA types” identified in Table 1.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>TYPENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>long double</td>
<td>longdouble</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>long long</td>
<td>longlong</td>
</tr>
</tbody>
</table>

Table 1: Standard RMA Types and Names

#### 8.3.1 SHMEM_PUT

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

**SYNOPSIS**

**C11:**

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

where `TYPE` is one of the standard RMA types specified by Table 1.

**C/C++:**

Formally, the C11 specification is ISO/IEC 9899:2011(E).
### 8. OPENSHMEM LIBRARY API

**void shmem_<TYPENAME>_{put}(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 1.

**void shmem_put<SIZE>(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);**

**FORTRAN:**

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

### DESCRIPTION

**Arguments**

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

**API description**

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of <code>dest</code> and <code>source</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_putmem</td>
<td><em>Fortran</em>: Any noncharacter type. C: Any data type. <code>nelems</code> is scaled in bytes.</td>
</tr>
<tr>
<td>shmem_put4, shmem_put32</td>
<td>Any noncharacter type that has a storage size equal to 32 bits. C: Any noncharacter type that has a storage size equal to 8 bits. <em>Fortran</em>: Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_put8</td>
<td><em>Fortran</em>: Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_put64</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_put128</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>SHMEM_CHARACTER_PUT</td>
<td>Elements of type character. <code>nelems</code> is the number of characters to transfer. The actual character lengths of the <code>source</code> and <code>dest</code> variables are ignored.</td>
</tr>
</tbody>
</table>
8. OPENSHMEM LIBRARY API

SHMEM_COMPLEX_PUT  Elements of type complex of default size.
SHMEM_DOUBLE_PUT   Elements of type double precision.
SHMEM_INTEGER_PUT  Elements of type integer.
SHMEM_LOGICAL_PUT  Elements of type logical.
SHMEM_REAL_PUT     Elements of type real.

Return Values
None.

Notes
If you are using Fortran, data types must be of default size. For example, a real variable must be declared as REAL, REAL*4, or REAL(KIND=KIND(1.0)). The Fortran API routine SHMEM_PUT has been deprecated, and either SHMEM_PUT8 or SHMEM_PUT64 should be used in its place.

EXAMPLES

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

```c
#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();
    if (shmem_my_pe() == 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", shmem_my_pe(), dest[0]);
    return 0;
}
```

8.3.2 SHMEM_P

Copies one data item to a remote PE.

SYNOPSIS

C11:
```c
void shmem_p(TYPE *dest, TYPE value, int pe);
```
where TYPE is one of the standard RMA types specified by Table 1.

C/C++:
```c
void shmem_<TYPENAME>_p(TYPE *dest, TYPE value, int pe);
```
where TYPE is one of the standard RMA types and has a corresponding TYPENAME specified by Table 1.

DESCRIPTION

Arguments

```
IN     addr    The remotely accessible array element or scalar data object which will receive the data on the remote PE.
```
8. OPENSHMEM LIBRARY API

These routines provide a very low latency put capability for single elements of most basic types.
As with `shmem_put`, these routines start the remote transfer and may return before the data is delivered to the remote PE. Use `shmem_quiet` to force completion of all remote Put transfers.

**Return Values**
None.

**Notes**
None.

**EXAMPLES**

The following example uses `shmem_p` in a C program.

```c
#include <stdio.h>
#include <math.h>
#include <shmem.h>
static const double e = 2.71828182;
static const double epsilon = 0.00000001;

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

    shmem_init();
    me = shmem_my_pe();
    f = (double *) shmem_malloc(sizeof(*f));
    *f = 3.1415927;
    shmem_barrier_all();

    if (me == 0)
        shmem_p(f, e, 1);
    shmem_barrier_all();
    if (me == 1)
        printf("%s\n", (fabs(*f - e) < epsilon) ? "OK" : "FAIL");

    return 0;
}
```

8.3.3 SHMEM_IPUT

Copies strided data to a specified PE.

**SYNOPSIS**

C11:

```c
void shmem_iput(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 1.

C/C++:

```c
```
void shmem_<TYPENAME>_iput(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 1.

void shmem_iput<SIZE>(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.

FORTRAN:

INTEGER dst, sst, nelems, pe
CALL SHMEM_COMPLEX_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_DOUBLE_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_INTEGER_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT4(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT8(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT32(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT64(dest, source, dst, sst, nelems, pe)
CALL SHMEM_IPUT128(dest, source, dst, sst, nelems, pe)
CALL SHMEM_LOGICAL_IPUT(dest, source, dst, sst, nelems, pe)
CALL SHMEM_REAL_IPUT(dest, source, dst, sst, nelems, pe)

DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>OUT</th>
<th>dest</th>
<th>Array to be updated on the remote PE. This data object must be remotely accessible.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>source</td>
<td>Array containing the data to be copied.</td>
</tr>
<tr>
<td>IN</td>
<td>dst</td>
<td>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. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td>sst</td>
<td>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. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td>nelems</td>
<td>Number of elements in the dest and source arrays. nelems must be of type size_t for C. If you are using Fortran, it must be a constant, variable, or array element of default integer type.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>PE number of the remote PE. pe must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.</td>
</tr>
</tbody>
</table>

API description

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of dest and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_iput4, shmem_iput32</td>
<td>Any noncharacter type that has a storage size equal to 32 bits.</td>
</tr>
</tbody>
</table>
OPENSHMEM LIBRARY API

8. SHMEM/API

8. OPENSHMEM LIBRARY API

shmem_iput8  C: Any noncharacter type that has a storage size equal to 8 bits.

Fortran: Any noncharacter type that has a storage size equal to 64 bits.

shmem_iput64  Any noncharacter type that has a storage size equal to 64 bits.

shmem_iput128 Any noncharacter type that has a storage size equal to 128 bits.

SHMEM_COMPLEX_IPUT  Elements of type complex of default size.

SHMEM_DOUBLE_IPUT  Elements of type double precision.

SHMEM_INTEGER_IPUT  Elements of type integer.

SHMEM_LOGICAL_IPUT  Elements of type logical.

SHMEM_REAL_IPUT  Elements of type real.

Return Values

None.

Notes

If you are using Fortran, data types must be of default size. For example, a real variable must be declared as REAL, REAL*4 or REAL(KIND=KIND(1.0)). See Section 3 for a definition of the term remotely accessible.

EXAMPLES

Consider the following shmem_iput example for C/C++ programs.

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

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

8.3.4 SHMEM_GET

Copies data from a specified PE.

SYNOPSIS

C11:

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

where TYPE is one of the standard RMA types specified by Table 1.

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of dest and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_getmem</td>
<td>Fortran: Any noncharacter type. C: Any data type. nelems is scaled in bytes.</td>
</tr>
<tr>
<td>shmem_get4, shmem_get32</td>
<td>Any noncharacter type that has a storage size equal to 32 bits. C: Any noncharacter type that has a storage size equal to 8 bits. Fortran: Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_get8</td>
<td>Any noncharacter type that has a storage size equal to 8 bits. Fortran: Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>shmem_get64</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_get128</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>SHMEM_CHARACTER_GET</td>
<td>Elements of type character. nelems is the number of characters to transfer. The actual character lengths of the source and dest variables are ignored.</td>
</tr>
</tbody>
</table>
SHMEM_COMPLEX_GET  Elements of type complex of default size.
SHMEM_DOUBLE_GET   Fortran: Elements of type double precision.
SHMEM_INTEGER_GET  Elements of type integer.
SHMEM_LOGICAL_GET  Elements of type logical.
SHMEM_REAL_GET     Elements of type real.

Return Values
None.

Notes
See Section 3 for a definition of the term remotely accessible. If you are using Fortran, data types must be of default size. For example, a real variable must be declared as REAL, REAL*4, or REAL(KIND=KIND(1.0)).

EXAMPLES

Consider this example for Fortran.

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

8.3.5 SHMEM_G

Copies one data item from a remote PE

SYNOPSIS

C11:

```c
TYPE shmem_g(const TYPE *addr, int pe);
```

where `TYPE` is one of the standard RMA types specified by Table 1.

C/C++:

```c
TYPE shmem_<typename>_g(const TYPE *addr, int pe);
```

where `TYPE` is one of the standard RMA types and has a corresponding `typename` specified by Table 1.

DESCRIPTION

Arguments

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>addr</td>
<td>The remotely accessible array element or scalar data object.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>The number of the remote PE on which <code>addr</code> resides.</td>
</tr>
</tbody>
</table>
API description
These routines provide a very low latency get capability for single elements of most basic types.

Return Values
Returns a single element of type specified in the synopsis.

Notes
None.

EXAMPLES
The following shmem_g example is for C/C++ programs:

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

long x = 10101;

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

8.3.6 SHMEM_IGET
Copies strided data from a specified PE.

SYNOPSIS

C11:
```c
void shmem_iget(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 1.

C/C++:
```c
void shmem_<TYPENAME>_iget(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 1.

```c
void shmem_iget<8>(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.

FORTRAN:
8. OPENSHEM LIBRARY API

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

**DESCRIPTION**

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OUT</strong></td>
<td>dest</td>
</tr>
<tr>
<td><strong>IN</strong></td>
<td>source</td>
</tr>
<tr>
<td><strong>IN</strong></td>
<td>dst</td>
</tr>
<tr>
<td><strong>IN</strong></td>
<td>sst</td>
</tr>
<tr>
<td><strong>IN</strong></td>
<td>nelems</td>
</tr>
<tr>
<td><strong>IN</strong></td>
<td>pe</td>
</tr>
</tbody>
</table>

**API description**

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of dest and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_iget4, shmem_iget32</td>
<td>Any noncharacter type that has a storage size equal to 32 bits.</td>
</tr>
<tr>
<td>shmem_iget8</td>
<td>Any noncharacter type that has a storage size equal to 8 bits.</td>
</tr>
<tr>
<td>Fortran: Any noncharacter type that has a storage size equal to 64 bits.</td>
<td></td>
</tr>
<tr>
<td>shmem_iget64</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_iget128</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>SHMEM_COMPLEX_IGET</td>
<td>Elements of type complex of default size.</td>
</tr>
<tr>
<td>SHMEM_DOUBLE_IGET</td>
<td>Fortran: Elements of type double precision.</td>
</tr>
<tr>
<td>SHMEM_INTEGER_IGET</td>
<td>Elements of type integer.</td>
</tr>
<tr>
<td>SHMEM_LOGICAL_IGET</td>
<td>Elements of type logical.</td>
</tr>
<tr>
<td>SHMEM_REAL_IGET</td>
<td>Elements of type real.</td>
</tr>
</tbody>
</table>
Return Values
None.

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

EXAMPLES

The following example uses shmem_logical_iget in a Fortran program.

```fortran
PROGRAM STRIDELOGICAL
INCLUDE "shmem.fh"

LOGICAL SOURCE(10), DEST(5)
SAVE SOURCE ! SAVE MAKES IT REMOTELY ACCESSIBLE
DATA DEST / 5*.F. /
CALL SHMEM_INIT()
IF (SHMEM_MY_PE() .EQ. 0) THEN
  CALL SHMEM_LOGICAL_IGET(DEST, SOURCE, 1, 2, 5, 1)
  PRINT*, 'DEST AFTER SHMEM_LOGICAL_IGET:', DEST
ENDIF
CALL SHMEM_BARRIER_ALL
```

8.4 Non-blocking Remote Memory Access Routines

8.4.1 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:
```c
void shmem_put_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe);
```
where TYPE is one of the standard RMA types specified by Table 1.

C/C++:
```c
void shmem_<TYPENAME>_put_nbi(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 1.

```c
void shmem_put<SIZE>_nbi(void *dest, const void *source, size_t nelems, int pe);
```
where SIZE is one of 8, 16, 32, 64, 128.

```c
void shmem_putmem_nbi(void *dest, const void *source, size_t nelems, int pe);
```

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

DESCRIPTION

Arguments

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. If you are using Fortran, it must be a constant, variable, or array element of default integer type.

IN pe PE number of the remote PE. pe must be of type integer. If you are using Fortran, it must be a constant, variable, or array element of default integer type.

API description

The routines return after posting the operation. The operation is considered complete after a subsequent call to shmem_quiet. At the completion of shmem_quiet, the data has been copied into the dest array on the destination PE. The delivery of data words into the data object on the destination PE may occur in any order. Furthermore, two successive put routines may deliver data out of order unless a call to shmem_fence is introduced between the two calls.

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of dest and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_putmem_nbi</td>
<td>Fortran: Any noncharacter type. C: Any data type. nelems is scaled in bytes.</td>
</tr>
<tr>
<td>shmem_put4_nbi,</td>
<td>Any noncharacter type that has a storage size equal to 32 bits.</td>
</tr>
<tr>
<td>shmem_put32_nbi</td>
<td>C: Any noncharacter type that has a storage size equal to 8 bits. Fortran: Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_put8_nbi</td>
<td>Any noncharacter type that has a storage size equal to 64 bits.</td>
</tr>
<tr>
<td>shmem_put64_nbi</td>
<td>Any noncharacter type that has a storage size equal to 128 bits.</td>
</tr>
<tr>
<td>SHMEM_CHARACTER_PUT_NBI</td>
<td>Elements of type character. nelems is the number of characters to transfer. The actual character lengths of the source and dest variables are ignored.</td>
</tr>
<tr>
<td>SHMEM_COMPLEX_PUT_NBI</td>
<td>Elements of type complex of default size.</td>
</tr>
<tr>
<td>SHMEM_DOUBLE_PUT_NBI</td>
<td>Elements of type double precision.</td>
</tr>
<tr>
<td>SHMEM_INTEGER_PUT_NBI</td>
<td>Elements of type integer.</td>
</tr>
<tr>
<td>SHMEM_LOGICAL_PUT_NBI</td>
<td>Elements of type logical.</td>
</tr>
<tr>
<td>SHMEM_REAL.PUT_NBI</td>
<td>Elements of type real.</td>
</tr>
</tbody>
</table>

Return Values
None.

Notes
None.
8.4.2 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:**
```c
void shmem_get_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe);
```
where `TYPE` is one of the standard RMA types specified by Table 1.

**C/C++:**
```c
void shmem_<TYPENAME>_get_nbi(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 1.

```c
void shmem_get<SIZE>_nbi(void *dest, const void *source, size_t nelems, int pe);
```
where `SIZE` is one of 8, 16, 32, 64, 128.

**FORTRAN:**
```fortran
INTEGER nelems, pe
CALL SHMEM_CHARACTER_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_COMPLEX_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_DOUBLE_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_GET4_NBI(dest, source, nelems, pe)
CALL SHMEM_GET8_NBI(dest, source, nelems, pe)
CALL SHMEM_GET32_NBI(dest, source, nelems, pe)
CALL SHMEM_GET128_NBI(dest, source, nelems, pe)
CALL SHMEM_INTEGER_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_LOGICAL_GET_NBI(dest, source, nelems, pe)
CALL SHMEM_REAL_GET_NBI(dest, source, nelems, pe)
```

DESCRIPTION

**Arguments**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td><code>dest</code></td>
</tr>
<tr>
<td>IN</td>
<td><code>source</code></td>
</tr>
<tr>
<td>IN</td>
<td><code>nelems</code></td>
</tr>
<tr>
<td>IN</td>
<td><code>pe</code></td>
</tr>
</tbody>
</table>

**API description**

The get routines provide a method for copying a contiguous symmetric data object from a different PE to a contiguous data object on the local PE. The routines return after posting the operation. The operation is considered complete after a subsequent call to `shmem_quiet`. At the completion of `shmem_quiet`, the data has been delivered to the `dest` array on the local PE.

The `dest` and `source` data objects must conform to typing constraints, which are as follows:
8. OPENSHMEM LIBRARY API

Routine | Data type of dest and source
--- | ---
shmem_getmem_nbi | Fortran: Any noncharacter type. C: Any data type. nelems is scaled in bytes.
shmem_get4_nbi, shmem_get32_nbi | Any noncharacter type that has a storage size equal to 32 bits.
shmem_get8_nbi | C: Any noncharacter type that has a storage size equal to 8 bits. Fortran: Any noncharacter type that has a storage size equal to 64 bits.
shmem_get64_nbi, shmem_get128_nbi | Any noncharacter type that has a storage size equal to 64 bits.
SHMEM_CHARACTER_GET_NBI | Elements of type character. nelems is the number of characters to transfer. The actual character lengths of the source and dest variables are ignored.
SHMEM_COMPLEX_GET_NBI | Elements of type complex of default size.
SHMEM_DOUBLE_GET_NBI | Fortran: Elements of type double precision.
SHMEM_INTEGER_GET_NBI | Elements of type integer.
SHMEM_LOGICAL_GET_NBI | Elements of type logical.
SHMEM_REAL_GET_NBI | Elements of type real.

Return Values
None.

Notes
See Section 3 for a definition of the term remotely accessible. If you are using Fortran, data types must be of default size. For example, a real variable must be declared as REAL, REAL*4, or REAL(KIND=KIND(1.0)).

8.5 Atomic Memory Operations

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

- The 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 and delivered to the local PE.
  The fetching operations include: SHMEM_FETCH, SHMEM_CSWAP, SHMEM_SWAP, SHMEM_FINC, and SHMEM_FADD.
- The non-fetching atomic routines update the remote memory in a single atomic operation. A non-fetching atomic routine starts the atomic operation and may return before the operation execution on the remote PE. To force completion for these non-fetching atomic routines, shmem_quiet, shmem_barrier, or shmem_barrier_all can be used by an OpenSHMEM program.
  The non-fetching operations include: SHMEM_SET, SHMEM_INC and SHMEM_ADD.

Where appropriate compiler support is available, OpenSHMEM provides type-generic atomic memory operation interfaces via C11 generic selection. The type-generic AMO routines each support the “standard AMO types” listed in Table 2, except for shmem_fetch, shmem_set, and shmem_swap, which supports the “extended AMO types” listed in Table 3.
8.6.1 SHMEM_ADD

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

SYNOPSIS

C11:
```c
void shmem_add(TYPE *dest, TYPE value, int pe);
```

where `TYPE` is one of the standard AMO types specified by Table 2.

C/C++:
```c
void shmem_<TYPENAME>_add(TYPE *dest, TYPE value, int pe);
```

where `TYPE` is one of the standard AMO types and has a corresponding `TYPENAME` specified by Table 2.

FORTRAN:
```fortran
INTEGER pe
INTEGER*4 value_i4
CALL SHMEM_INT4_ADD(dest, value_i4, pe)
INTEGER*8 value_i8
CALL SHMEM_INT8_ADD(dest, value_i8, pe)
```

DESCRIPTION

Arguments

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

API description

The `shmem_add` routine performs an atomic add operation. It adds `value` to `dest` on PE `pe` and atomically updates the `dest` without returning the value.
If you are using Fortran, dest must be of the following type:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_ADD</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_ADD</td>
<td>8-byte integer</td>
</tr>
</tbody>
</table>

**Return Values**

None.

**Notes**

The term remotely accessible is defined in Section 3.

**EXAMPLES**

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

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

**8.5.2 SHMEM_CSWAP**

Performs an atomic conditional swap on a remote data object.

**SYNOPSIS**

**C11:**

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

where *TYPE* is one of the standard AMO types specified by Table 2.

**C/C++:**

```c
TYPE shmem_<TYPENAME>_cswap(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 2.

**FORTRAN:**

```fortran
shmem_cswap(dest, cond, value, pe);
```
**DESCRIPTION**

**Arguments**

<table>
<thead>
<tr>
<th>OUT</th>
<th>dest</th>
<th>The remotely accessible integer data object to be updated on the remote PE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>cond</td>
<td>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.</td>
</tr>
<tr>
<td>IN</td>
<td>value</td>
<td>The value to be atomically written to the remote PE. value must be the same data type as dest.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>An integer that indicates the PE number upon which dest is to be updated. If you are using Fortran, it must be a default integer value.</td>
</tr>
</tbody>
</table>

**API description**

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of dest and value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_CSWAP</td>
<td>4-byte integer.</td>
</tr>
<tr>
<td>SHMEM_INT8_CSWAP</td>
<td>8-byte integer.</td>
</tr>
</tbody>
</table>

**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.

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

int main(void)
{
  static int race_winner = -1;
  int oldval;
  shmem_init();
  oldval = shmem_cswap(&race_winner, -1, shmem_my_pe(), 0);
```
8. OPENSHMEM LIBRARY API

if (oldval == -1) printf("pe %d was first\n",shmem_my_pe());
    return 1;
}

8.5.3 SHMEM_SWAP

Performs an atomic swap to a remote data object.

SYNOPSIS

C11:

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

where `TYPE` is one of the extended AMO types specified by Table 3.

C/C++:

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

where `TYPE` is one of the extended AMO types and has a corresponding `TYPENAME` specified by Table 3.

FORTRAN:

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

DESCRIPTION

Arguments

| OUT | dest | The remotely accessible integer data object to be updated on the remote PE. If you are 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. If you are using Fortran, it must be a default integer value. |

API description

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of <code>dest</code> and <code>source</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_SWAP</td>
<td>Integer of default kind</td>
</tr>
<tr>
<td>SHMEM_INT4_SWAP</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_SWAP</td>
<td>8-byte integer</td>
</tr>
<tr>
<td>SHMEM_REAL4_SWAP</td>
<td>4-byte real</td>
</tr>
</tbody>
</table>
SHMEM_REAL8_SWAP  8-byte real

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

Notes
None.

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

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

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

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

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

SYNOPSIS

C11:
```
TYPE shmem_finc(TYPE *dest, int pe);
```
where TYPE is one of the standard AMO types specified by Table 2.

C/C++:
```
TYPE shmem_<TYPENAME>_finc(TYPE *dest, int pe);
```
where TYPE is one of the standard AMO types and has a corresponding TYPENAME specified by Table 2.

FORTRAN:
```
INTEGER pe
INTEGER*4 SHMEM_INT4_FINC, ires_i4
ires_i4 = SHMEM_INT4_FINC(dest, pe)
INTEGER*8 SHMEM_INT8_FINC, ires_i8
ires_i8 = SHMEM_INT8_FINC(dest, pe)
```
DESCRIPTION

Arguments

IN \textit{dest} \hspace{1cm} The remotely accessible integer data object to be updated on the remote PE. The type of \textit{dest} should match that implied in the SYNOPSIS section.

IN \textit{pe} \hspace{1cm} An integer that indicates the PE number on which \textit{dest} is to be updated. If you are using Fortran, it must be a default integer value.

API description

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

If you are using Fortran, \textit{dest} must be of the following type:

\begin{tabular}{|l|l|}
\hline
Routine & Data type of \textit{dest} and source \\
\hline
SHMEM\_INT4\_FINC & 4-byte integer \\
SHMEM\_INT8\_FINC & 8-byte integer \\
\hline
\end{tabular}

Return Values

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

Notes

None.

EXAMPLES

The following \texttt{shmem\_finc} example is for C/C++ programs:

\begin{verbatim}
#include <stdio.h>
#include <shmem.h>

int dst;

int main(void)
{
    int me;
    int old;
    shmem_init();
    me = shmem_my_pe();
    old = -1;
    dst = 22;
    shmem_barrier_all();

    if (me == 0)
    {
        old = shmem_finc(&dst, 1);
    }
    shmem_barrier_all();

    printf("Return Value: \%d\n", old);
    return 0;
}
\end{verbatim}
8.5.5 SHMEM_INC

Performs an atomic increment operation on a remote data object.

SYNOPSIS

C11:

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

where *TYPE* is one of the standard AMO types specified by Table 2.

C/C++:

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

where *TYPE* is one of the standard AMO types and has a corresponding *TYPENAME* specified by Table 2.

FORTRAN:

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

DESCRIPTION

Arguments

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

API description

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of <em>dest</em> and <em>source</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_INC</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_INC</td>
<td>8-byte integer</td>
</tr>
</tbody>
</table>

Return Values

None.

Notes

The term remotely accessible is defined in Section 3.
8. EXAMPLES

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

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

int dst;

int main(void)
{
    int me;

    shmem_init();
    me = shmem_my_pe();
    dst = 74;
    shmem_barrier_all();
    if (me == 0)
        shmem_inc(&dst, 1);
    shmem_barrier_all();
    printf("%d: dst = %d\n", me, dst);
    return 0;
}
```

8.5.6 SHMEM_FADD

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

SYNOPSIS

C11:

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

where `TYPE` is one of the standard AMO types specified by Table 2.

C/C++:

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

where `TYPE` is one of the standard AMO types and has a corresponding `TYPENAME` specified by Table 2.

FORTRAN:

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

DESCRIPTION

Arguments

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

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of <code>dest</code> and <code>source</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_INT4_FADD</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>SHMEM_INT8_FADD</td>
<td>8-byte integer</td>
</tr>
</tbody>
</table>

Return Values

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

Notes

None.

EXAMPLES

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

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

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

    shmem_init();
    me = shmem_my_pe();
    old = -1;
    dst = 22;
    shmem_barrier_all();

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

8.5.7 SHMEM_FETCH

Atomically fetches the value of a remote data object.

SYNOPSIS

C11:
8. OPENSHMEM LIBRARY API

```c
TYPE shmem_fetch(const TYPE *dest, int pe);
```

where \( TYPE \) is one of the extended AMO types specified by Table 3.

C/C++:

```c
TYPE shmem<TYPENAME>_fetch(const TYPE *dest, int pe);
```

where \( TYPE \) is one of the extended AMO types and has a corresponding \( TYPENAME \) specified by Table 3.

FORTRAN:

```fortran
INTEGER pe
INTEGER*4 SHMEM_INT4_FETCH, ires_i4
ires_i4 = SHMEM_INT4_FETCH(dest, pe)
INTEGER*8 SHMEM_INT8_FETCH, ires_i8
ires_i8 = SHMEM_INT8_FETCH(dest, pe)
REAL*4 SHMEM_REAL4_FETCH, res_r4
res_r4 = SHMEM_REAL4_FETCH(dest, pe)
REAL*8 SHMEM_REAL8_FETCH, res_r8
res_r8 = SHMEM_REAL8_FETCH(dest, pe)
```

DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>IN</th>
<th>dest</th>
<th>The remotely accessible data object to be fetched from the remote PE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>pe</td>
<td>An integer that indicates the PE number from which ( dest ) is to be fetched.</td>
</tr>
</tbody>
</table>

API description

\( shmem\_fetch \) performs an atomic fetch operation. It returns the contents of the \( dest \) as an atomic operation.

Return Values

The contents at the \( dest \) address on the remote PE. The data type of the return value is the same as the the type of the remote data object.

Notes

None.

8.5.8 SHMEM_SET

Atomically sets the value of a remote data object.

SYNOPSIS

C11:

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

where \( TYPE \) is one of the extended AMO types specified by Table 3.

C/C++:

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

where \( TYPE \) is one of the extended AMO types and has a corresponding \( TYPENAME \) specified by Table 3.

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

DESCRIPTION

Arguments

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>dest</td>
<td>The remotely accessible data object to be set on the remote PE.</td>
</tr>
<tr>
<td>IN</td>
<td>value</td>
<td>The value to be atomically written to the remote PE.</td>
</tr>
<tr>
<td>IN</td>
<td>pe</td>
<td>An integer that indicates the PE number on which dest is to be updated.</td>
</tr>
</tbody>
</table>

API description

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

Return Values

None.

Notes

None.

8.6 Collective Routines

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

The Active set is defined by the arguments PE_start, logPE_stride, and PE_size. PE_start is the starting PE number, a log (base 2) of logPE_stride is the stride between PEs, and PE_size is the number of PEs participating in the Active set. All PEs participating in the collective routines provide the same values for these arguments.

Another argument important to collective routines is pSync, which is a symmetric work array. All PEs participating in a collective must pass the same pSync array. On completion of a collective call, the pSync is restored to its original contents. The user is permitted to reuse a pSync array if all previous collective routines using the pSync array have been completed by all participating PEs. One can use a synchronization collective routine such as SHMEM_BARRIER to ensure completion of previous collective routines. The shmem_barrier routine allows the same pSync array to be used on consecutive calls as long as the PE Active set does not change.

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

SHMEM_BROADCAST

SHMEM_BARRIER
SHMEM_BARRIER_ALL

SHMEM_COLLECT

SHMEM_FCOLLECT

Reduction Operations

SHMEM_ALLTOALL

SHMEM_ALLTOALLS

8.6.1 SHMEM_BARRIER_ALL

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

SYNOPSIS

C/C++:

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

FORTRAN:

```fortran
CALL SHMEM_BARRIER_ALL
```

DESCRIPTION

Arguments

None.

API description

The `shmem_barrier_all` routine registers the arrival of a PE at a barrier. Barriers are a fast mechanism for synchronizing all PEs at once. This routine causes a PE to suspend execution until all PEs have called `shmem_barrier_all`. This routine must be used with PEs started by `shmem_init`.

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

Return Values

None.

Notes

None.

EXAMPLES

The following `shmem_barrier_all` example is for C/C++ programs:
```c
#include <stdio.h>
#include <shmem.h>

int x=1010;

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

    /*put to next PE in a circular fashion*/
    shmem_int_p(&x, 4, (me+1)%npes);
    /*synchronize all PEs*/
    shmem_barrier_all();
    printf("%d: x = %d\n", me, x);
    return 0;
}
```

### 8.6.2 SHMEM\_BARRIER

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

#### SYNOPSIS

**C/C++:**

```c
void shmem_barrier(int PE\_start, int logPE\_stride, int PE\_size, long \*pSync);
```

**FORTRAN:**

```fortran
INTEGER PE\_start, logPE\_stride, PE\_size
INTEGER pSync(SHMEM\_BARRIER\_SYNC\_SIZE)
CALL SHMEM\_BARRIER(PE\_start, logPE\_stride, PE\_size, pSync)
```

#### DESCRIPTION

**Arguments**

<table>
<thead>
<tr>
<th>Arg</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td><code>PE\_start</code></td>
<td>The lowest PE number of the <code>Active set</code> of PEs. <code>PE\_start</code> must be of type integer. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td><code>logPE\_stride</code></td>
<td>The log (base 2) of the stride between consecutive PE numbers in the <code>Active set</code>. <code>logPE\_stride</code> must be of type integer. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td><code>PE\_size</code></td>
<td>The number of PEs in the <code>Active set</code>. <code>PE\_size</code> must be of type integer. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td><code>pSync</code></td>
<td>A symmetric work array. In C/C++, <code>pSync</code> must be of type long and size <code>SHMEM\_BARRIER\_SYNC\_SIZE</code>. In Fortran, <code>pSync</code> must be of type integer and size <code>SHMEM\_BARRIER\_SYNC\_SIZE</code>. If you are using Fortran, it must be a default integer type. Every element of this array must be initialized to <code>SHMEM\_SYNC\_VALUE</code> before any of the PEs in the <code>Active set</code> enter <code>shmem\_barrier</code> the first time.</td>
</tr>
</tbody>
</table>

#### API description

`shmem\_barrier` is a collective synchronization routine over an `Active set`. Control returns from `shmem\_barrier` after all PEs in the `Active set` (specified by `PE\_start`, `logPE\_stride`, and `PE\_size`) have called `shmem\_barrier`. 
As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the *Active set* call the routine. If a PE not in the *Active set* calls an OpenSHMEM collective routine, undefined behavior results.

The values of arguments *PE_start*, *logPE_stride*, and *PE_size* must be equal on all PEs in the *Active set*. The same work array must be passed in *pSync* to all PEs in the *Active set*.

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

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

**Return Values**

None.

**Notes**

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

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

**EXAMPLES**

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

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

long pSync[SHMEM_BARRIER_SYNC_SIZE];
int x = 10101;

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

    for (i = 0; i < SHMEM_BARRIER_SYNC_SIZE; i += 1)
    {
        pSync[i] = SHMEM_SYNC_VALUE;
    }

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

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

8.6.3 SHMEM_BROADCAST

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

**SYNOPSIS**

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

FORTRAN:
INTEGER nelems, PE_root, PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_BCAST_SYNC_SIZE)
CALL SHMEM_BROADCAST4(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST8(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST32(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_BROADCAST64(dest, source, nelems, PE_root, PE_start, logPE_stride, PE_size, pSync)

DESCRIPTION
Arguments

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

API description
OpenSHMEM broadcast routines are collective routines. They copy data object source on the processor specified by PE_root and store the values at dest on the other PEs specified by the triplet PE_start, logPE_stride, PE_size. The data is not copied to the dest area on the root PE.
As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the Active set call the routine. If a PE not in the Active set calls an OpenSHMEM collective routine, undefined behavior results.
The values of arguments PE_root, PE_start, logPE_stride, and PE_size must be equal on all PEs in the Active set. The same dest and source data objects and the same pSync work array must be passed to all PEs in the Active set.
Before any PE calls a broadcast routine, you must ensure that the following conditions exist (synchronization via a barrier or some other method is often needed to ensure this): The pSync array on all PEs in the Active set is not still in use from a prior call to a broadcast routine. The dest array on all PEs in the Active set is ready to accept the broadcast data.

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

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

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

Return Values
None.

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

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

EXAMPLES

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

C/C++ example:

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

#define NUM_ELEMS 4

long pSync[SHMEM_BCAST_SYNC_SIZE];
long source[NUM_ELEMS], dest[NUM_ELEMS];

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

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

  if (me == 0)
    for (i = 0; i < NUM_ELEMS; i++)
      source[i] = i;
```
for (i=0; i < SHMEM_BCAST_SYNC_SIZE; i++) {
    pSync[i] = SHMEM_SYNC_VALUE;
}

shmem_barrier_all(); /* Wait for all PEs to initialize pSync */

shmem_broadcast64(dest, source, NUM_ELEMS, 0, 0, npes, pSync);
printf("%d: %ld", me, dest[0]);
for (i = 1; i < NUM_ELEMS; i++)
    printf("%ld", dest[i]);
printf("\n");
return 0;

Fortran example:

INCLUDE "shmem.fh"

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

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

8.6.4 SHMEM_COLLECT, SHMEM_FCOLLECT

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

SYNOPSIS

C/C++:

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

FORTRAN:

INTEGER nelems
INTEGER PE_start, logPE_stride, PE_size
INTEGER pSync(SHMEM_COLLECT_SYNC_SIZE)

CALL SHMEM_COLLECT4(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT8(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT32(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)
CALL SHMEM_COLLECT64(dest, source, nelems, PE_start, logPE_stride, PE_size, pSync)

DESCRIPTION

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

IN  source  A symmetric data object that can be of any type permissible for the dest argument.

IN  nelems  The number of elements in the source array. nelems must be of type size_t for C. If you are using Fortran, it must be a default integer value. The values in the source array are restored to the original values.

IN  PE_start  The lowest PE number of the Active set of PEs. PE_start must be of type integer. If you are using Fortran, it must be a default integer value. As with all OpenSHMEM collective routines, each of these routines assumes that only PEs in the Active set call the routine. If a PE not in the Active set and calls this collective routine, the behavior is undefined.

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. If you are using Fortran, it must be a default integer value.

IN  PE_size  The number of PEs in the Active set. PE_size must be of type integer. If you are using Fortran, it must be a default integer value.

IN  pSync  A symmetric work array. In C/C++, pSync must be of type long and size SHMEM_COLLECT_SYNC_SIZE. In Fortran, pSync must be of type integer and size SHMEM_COLLECT_SYNC_SIZE. If you are using Fortran, it must be a default integer value. Every element of this array must be initialized with the value SHMEM_SYNC_VALUE in C/C++ or SHMEM_SYNC_VALUE in Fortran before any of the PEs in the Active set enter shmem_collect or shmem_fcollect.

API description
OpenSHMEM collect and fcollect routines concatenate nelems 64-bit or 32-bit data items from the source array into the dest array, over the set of PEs defined by PE_start, log2PE_stride, and PE_size, in processor number order. The resultant dest array contains the contribution from PE PE_start first, then the contribution from PE PE_start + PE_stride second, and so on. The collected result is written to the dest array for all PEs in the Active set.

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

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

The values of arguments PE_start, logPE_stride, and PE_size must be equal on all PEs in the Active set. The same dest and source arrays and the same pSync work array must be passed to all PEs in the Active set.

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

Return Values
None.

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

You must ensure that the pSync array is not being updated on any PE in the Active set while any of the PEs participate in processing of an OpenSHMEM collective routine. Be careful to avoid these situations: If the
pSync array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the working set have initialized pSync before any of them enter an OpenSHMEM routine called with the pSync synchronization array. A pSync array can be reused on a subsequent OpenSHMEM collective routine only if none of the PEs in the Active set are still processing a prior OpenSHMEM collective routine call that used the same pSync array. In general, this may be ensured only by doing some type of synchronization. The collective routines operate on active PE sets that have a non-power-of-two PE_size with some performance degradation. They operate with no performance degradation when nelems is a non-power-of-two value.

**EXAMPLES**

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

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

long pSync[SHMEM_COLLECT_SYNC_SIZE];
int source[2];

int main(void)
{
    int i, me, npes;
    int *dest;
    shmem_init();
    me = shmem_my_pe();
    npes = shmem_n_pes();
    source[0] = me * 2;
    source[1] = me * 2 + 1;
    for (i=0; i < SHMEM_COLLECT_SYNC_SIZE; i++) {
        pSync[i] = SHMEM_SYNC_VALUE;
    }
    shmem_barrier_all(); /* Wait for all PEs to initialize pSync */
    shmem_collect32(dest, source, 2, 0, 0, npes, pSync);
    printf("%d: %d", me, dest[0]);
    for (i = 1; i < npes * 2; i++)
        printf(" , %d", dest[i]);
    printf("\n");
    return 0;
}
```

The following `SHMEM_COLLECT` example is for Fortran programs:

```fortran
INCLUDE "shmem.fh"

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

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

### 8.6.5 SHMEM_REDUCTIONS

Performs arithmetic and logical operations across a set of PEs.

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

C/C++:
```c
void shmem_int_and_to_all(int *dest, const int *source, int nreduce, int PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_and_to_all(long *dest, const long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longlong_and_to_all(long long *dest, const long long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_and_to_all(short *dest, const short *source, int nreduce, int PE_start, int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

FORTRAN:
```fortran
CALL SHMEM_INT4_AND_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_AND_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
```

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

C/C++:
```c
void shmem_double_max_to_all(double *dest, const double *source, int nreduce, int PE_start, int logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_max_to_all(float *dest, const float *source, int nreduce, int PE_start, int logPE_stride, int PE_size, float *pWrk, long *pSync);
void shmem_int_max_to_all(int *dest, const int *source, int nreduce, int PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_max_to_all(long *dest, const long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_max_to_all(long double *dest, const long double *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_max_to_all(long long *dest, const long long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_max_to_all(short *dest, const short *source, int nreduce, int PE_start, int logPE_stride, int PE_size, short *pWrk, long *pSync);
```

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

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

C/C++:
```c
void shmem_double_min_to_all(double *dest, const double *source, int nreduce, int PE_start, int logPE_stride, int PE_size, double *pWrk, long *pSync);
void shmem_float_min_to_all(float *dest, const float *source, int nreduce, int PE_start, int logPE_stride, int PE_size, float *pWrk, long *pSync);
```
void shmem_int_min_to_all(int *dest, const int *source, int nreduce, int PE_start, int logPE_stride, int PE_size, int *pWrk, long *pSync);
void shmem_long_min_to_all(long *dest, const long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long *pWrk, long *pSync);
void shmem_longdouble_min_to_all(long double *dest, const long double *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long double *pWrk, long *pSync);
void shmem_longlong_min_to_all(long long *dest, const long long *source, int nreduce, int PE_start, int logPE_stride, int PE_size, long long *pWrk, long *pSync);
void shmem_short_min_to_all(short *dest, const short *source, int nreduce, int PE_start, int logPE_stride, int PE_size, short *pWrk, long *pSync);

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

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

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

FORTRAN:
CALL SHMEM_COMP4_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_COMP8_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT4_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
CALL SHMEM_INT8_SUM_TO_ALL(dest, source, nreduce, PE_start, logPE_stride, PE_size, pWrk, pSync)
PROD
Performs a product reduction across a set of processing elements (PEs).

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

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

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

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

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

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

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

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

DESCRIPTION

Arguments

IN dest A symmetric array, of length nreduce elements, to receive the result of the reduction routines. The data type of dest varies with the version of the reduction routine being called. When calling from C/C++, refer to the SYNOPSIS section for data type information.

IN source A symmetric array, of length nreduce elements, that contains one element for each separate reduction routine. The source argument must have the same data type as dest.

IN nreduce The number of elements in the dest and source arrays. nreduce must be of type integer. If you are using Fortran, it must be a default integer value.

IN PE_start The lowest PE number of the Active set of PEs. PE_start must be of type integer. If you are using Fortran, it must be a default integer value.

IN logPE_stride The log (base 2) of the stride between consecutive PE numbers in the Active set. logPE_stride must be of type integer. If you are using Fortran, it must be a default integer value.

IN PE_size The number of PEs in the Active set. PE_size must be of type integer. If you are using Fortran, it must be a default integer value.

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

API description

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

The nreduce argument determines the number of separate reductions to perform. The source array on all PEs in the Active set provides one element for each reduction. The results of the reductions are placed in the dest array on all PEs in the Active set. The Active set is defined by the PE_start, logPE_stride, PE_size triplet.

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

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

The values of arguments nreduce, PE_start, logPE_stride, and PE_size must be equal on all PEs in the Active set. The same dest and source arrays, and the same pWrk and pSync work arrays, must be passed to all PEs in the Active set.

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

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_int8_and_to_all</td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td>shmem_int4_and_to_all</td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td>shmem_comp8_max_to_all</td>
<td>Complex, with an element size equal to two 8-byte real values.</td>
</tr>
<tr>
<td>shmem_int4_max_to_all</td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td>shmem_int8_max_to_all</td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td>shmem_real4_max_to_all</td>
<td>Real, with an element size of 4 bytes.</td>
</tr>
<tr>
<td>shmem_real16_max_to_all</td>
<td>Real, with an element size of 16 bytes.</td>
</tr>
<tr>
<td>shmem_int4_min_to_all</td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td>shmem_int8_min_to_all</td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td>shmem_real4_min_to_all</td>
<td>Real, with an element size of 4 bytes.</td>
</tr>
<tr>
<td>shmem_real8_min_to_all</td>
<td>Real, with an element size of 8 bytes.</td>
</tr>
<tr>
<td>shmem_real16_min_to_all</td>
<td>Real, with an element size of 16 bytes.</td>
</tr>
<tr>
<td>shmem_comp4_sum_to_all</td>
<td>Complex, with an element size equal to two 4-byte real values.</td>
</tr>
<tr>
<td>shmem_comp8_sum_to_all</td>
<td>Complex, with an element size equal to two 8-byte real values.</td>
</tr>
<tr>
<td>shmem_int4_sum_to_all</td>
<td>Integer, with an element size of 4 bytes.</td>
</tr>
<tr>
<td>shmem_int8_sum_to_all</td>
<td>Integer, with an element size of 8 bytes.</td>
</tr>
<tr>
<td>shmem_real4_sum_to_all</td>
<td>Real, with an element size of 4 bytes.</td>
</tr>
<tr>
<td>shmem_real8_sum_to_all</td>
<td>Real, with an element size of 8 bytes.</td>
</tr>
<tr>
<td>shmem_real16_sum_to_all</td>
<td>Real, with an element size of 16 bytes.</td>
</tr>
</tbody>
</table>
shmem_comp4_prod_to_all Complex, with an element size equal to two 4-byte real values.
shmem_comp8_prod_to_all Complex, with an element size equal to two 8-byte real values.
shmem_int4_prod_to_all Integer, with an element size of 4 bytes.
shmem_int8_prod_to_all Integer, with an element size of 8 bytes.
shmem_real4_prod_to_all Real, with an element size of 4 bytes.
shmem_real8_prod_to_all Real, with an element size of 8 bytes.
shmem_real16_prod_to_all Real, with an element size of 16 bytes.
shmem_int8_or_to_all Integer, with an element size of 8 bytes.
shmem_int4_or_to_all Integer, with an element size of 4 bytes.
shmem_int8_xor_to_all Integer, with an element size of 8 bytes.
shmem_int4_xor_to_all Integer, with an element size of 4 bytes.

Return Values
None.

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

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

```
INCLUDE "shmem.fh"

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

FOO = SHMEM_MY_PE()
IF ( MOD(SHMEM_MY_PE(), 2) .EQ. 0 ) THEN
  IF ( MOD(SHMEM_N_PES(), 2) .EQ. 0 ) THEN
    CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2, &
                              PWRK, PSYNC)
  ELSE
    CALL SHMEM_INT8_AND_TO_ALL(FOOAND, FOO, NR, 0, 1, NPES/2+1, &
                              PWRK, PSYNC)
  ENDIF
ENDIF
PRINT *, 'Result on PE ',SHMEM_MY_PE(),' is ', FOOAND
```

This Fortran example statically initializes the $pSync$ array and finds the maximum value of real variable $FOO$ across all even PEs.
INCLUDE "shmem.fh"

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

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

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

INCLUDE "shmem.fh"

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

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

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

INCLUDE "shmem.fh"

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

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

This Fortran example statically initializes the pSync array and finds the product of the real variable FOO across all the even PEs.
This Fortran example statically initializes the \textit{pSync} array and finds the logical OR of the integer variable \textit{FOO} across all even PEs.

```fortran
INCLUDE "shmem.fh"

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

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

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

```fortran
INCLUDE "shmem.fh"

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

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

### 8.6.6 SHMEM\_ALLTOALL

\texttt{shmem\_alltoall} is a collective routine where each PE exchanges a fixed amount of data with all other PEs in the \textit{Active set}.

**SYNOPSIS**

\textbf{C++:}

```c++
void shmem_alltoall32(void *dest, const void *source, size_t nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
void shmem_alltoall64(void *dest, const void *source, size_t nelems, int PE_start, int logPE_stride, int PE_size, long *pSync);
```

\textbf{FORTRAN:}

```fortran
INTEGER pSync(SHMEM\_ALLTOALL\_SYNC\_SIZE)
INTEGER PE_start, logPE_stride, PE_size, nelems
CALL SHMEM\_ALLTOALL32(dest, source, nelems, PE_start, logPE\_stride, PE\_size, pSync)
CALL SHMEM\_ALLTOALL64(dest, source, nelems, PE_start, logPE\_stride, PE\_size, pSync)
```

**DESCRIPTION**

Arguments
OUT  dest  A symmetric data object large enough to receive the combined total of nelems elements from each PE in the Active set.

IN  source  A symmetric data object that contains nelems elements of data for each PE in the Active set, ordered according to destination PE.

IN  nelems  The number of elements to exchange for each PE. nelems must be of type size_t for C/C++. If you are using Fortran, it must be a default integer value.

IN  PE_start  The lowest PE number of the Active set of PEs. PE_start must be of type integer. If you are using Fortran, it must be a default integer value.

IN  logPE_stride  The log (base 2) of the stride between consecutive PE numbers in the Active set. logPE_stride must be of type integer. If you are using Fortran, it must be a default integer value.

IN  PE_size  The number of PEs in the Active set. PE_size must be of type integer. If you are using Fortran, it must be a default integer value.

IN  pSync  A symmetric work array. In C/C++, pSync must be of type long and size SHMEM_ALLTOALL_SYNC_SIZE. In Fortran, pSync must be of type integer and size SHMEM_ALLTOALL_SYNC_SIZE. If you are using Fortran, it must be a default integer value. Every element of this array must be initialized with the value SHMEM_SYNC_VALUE before any of the PEs in the Active set enter the routine.

API description

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

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

The values of arguments nelems, PE_start, logPE_stride, and PE_size must be equal on all PEs in the Active set. The same dest and source data objects, and the same pSync work array must be passed to all PEs in the Active set.

Before any PE calls a shmem_alltoall routine, the following conditions must exist (synchronization via a barrier or some other method is often needed to ensure this): The pSync array on all PEs in the Active set is not still in use from a prior call to a shmem_alltoall/s routine. The dest data object on all PEs in the Active set is ready to accept the shmem_alltoall data.

Upon return from a shmem_alltoall routine, the following is true for the local PE: Its dest symmetric data object is completely updated and the data has been copied out of the source data object. The values in the pSync array are restored to the original values.

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of dest and source</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_alltoall64</td>
<td>64 bits aligned.</td>
</tr>
<tr>
<td>shmem_alltoall32</td>
<td>32 bits aligned.</td>
</tr>
</tbody>
</table>
8. OPENSHMEM LIBRARY API

Return Values
None.

Notes
This routine restores pSync to its original contents. Multiple calls to OpenSHMEM routines that use the same pSync array do not require that pSync be reinitialized after the first call. You must ensure the 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 that used the same pSync array. In general, this can be ensured only by doing some type of synchronization.

EXAMPLES

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

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

long pSync[SHMEM_ALLTOALL_SYNC_SIZE];

int main(void)
{
    int64_t *source, *dest;
    int i, count, pe;

    shmem_init();

    count = 2;
    dest = (int64_t*) shmem_malloc(count * shmem_n_pes() * sizeof(int64_t));
    source = (int64_t*) shmem_malloc(count * shmem_n_pes() * sizeof(int64_t));

    /* assign source values */
    for (pe=0; pe <shmem_n_pes(); pe++){
        for (i=0; i<count; i++){
            source[(pe*count)+i] = shmem_my_pe() + pe;
            dest[(pe*count)+i] = 9999;
        }
    }

    for (i=0; i< SHMEM_ALLTOALL_SYNC_SIZE; i++) {
        pSync[i] = SHMEM_SYNC_VALUE;
    }
    /* wait for all PEs to initialize pSync */
    shmem_barrier_all();

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

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

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

8.6.7 SHMEM_ALLTOALLS

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

SYNOPSIS

C/C++:

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

FORTRAN:

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

DESCRIPTION

Arguments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>dest</td>
<td>A symmetric data object large enough to receive the combined total of nelems elements from each PE in the Active set.</td>
</tr>
<tr>
<td>IN</td>
<td>source</td>
<td>A symmetric data object that contains nelems elements of data for each PE in the Active set, ordered according to destination PE.</td>
</tr>
<tr>
<td>IN</td>
<td>dst</td>
<td>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. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td>sst</td>
<td>The stride between consecutive elements of the source data object. The stride is scaled by the element size. A value of 1 indicates contiguous data. sst must be of type ptrdiff_t. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td>nelems</td>
<td>The number of elements to exchange for each PE. nelems must be of type size_t for C/C++. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td>PE_start</td>
<td>The lowest PE number of the Active set of PEs. PE_start must be of type integer. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td>logPE_stride</td>
<td>The log (base 2) of the stride between consecutive PE numbers in the Active set. logPE_stride must be of type integer. If you are using Fortran, it must be a default integer value.</td>
</tr>
<tr>
<td>IN</td>
<td>PE_size</td>
<td>The number of PEs in the Active set. PE_size must be of type integer. If you are using Fortran, it must be a default integer value.</td>
</tr>
</tbody>
</table>
IN \[ pSync \] 

A symmetric work array. In C/C++, \( pSync \) must be of type long and size \( SHMEM\_ALLTOALLS\_SYNC\_SIZE \). In Fortran, \( pSync \) must be of type integer and size \( SHMEM\_ALLTOALLS\_SYNC\_SIZE \). If you are using Fortran, it must be a default integer value. Every element of this array must be initialized with the value \( SHMEM\_SYNC\_VALUE \) before any of the PEs in the \( Active\ set \) enter the routine.

API description

The \( shmem\_alltoalls \) routines are collective routines. Each PE in the \( Active\ set \) exchanges \( nelems \) strided data elements of size 32 bits (for \( shmem\_alltoalls32 \)) or 64 bits (for \( shmem\_alltoalls64 \)) with all other PEs in the set. Both strides, \( dst \) and \( sst \), must be greater than or equal to 1. The \( sst\_j \) block sent from PE \( i \) to PE \( j \) is placed in the \( dst\_i \) block of the \( dest \) data object on PE \( j \).

As with all OpenSHMEM collective routines, these routines assume that only PEs in the \( Active\ set \) call the routine. If a PE not in the \( Active\ set \) calls an OpenSHMEM collective routine, undefined behavior results.

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

Before any PE calls to a \( shmem\_alltoalls \) routine, the following conditions must exist (synchronization via a barrier or some other method is often needed to ensure this): The \( pSync \) array on all PEs in the \( Active\ set \) is not still in use from a prior call to a \( shmem\_alltoalls \) routine. The \( dest \) data object on all PEs in the \( Active\ set \) is ready to accept the \( shmem\_alltoalls \) data.

Upon return from a \( shmem\_alltoalls \) routine, the following is true for the local PE: Its \( dest \) symmetric data object is completely updated and the data has been copied out of the \( source \) data object. The values in the \( pSync \) array are restored to the original values.

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type of ( dest ) and ( source )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( shmem_alltoalls64 )</td>
<td>64 bits aligned.</td>
</tr>
<tr>
<td>( shmem_alltoalls32 )</td>
<td>32 bits aligned.</td>
</tr>
</tbody>
</table>

Return Values

None.

Notes

This routine restores \( pSync \) to its original contents. Multiple calls to OpenSHMEM routines that use the same \( pSync \) array do not require that \( pSync \) be reinitialized after the first call. You must ensure that the \( pSync \) array is not being updated by any PE in the \( Active\ set \) while any of the PEs participates in processing of an OpenSHMEM \( shmem\_alltoalls \) routine. Be careful to avoid these situations: If the \( pSync \) array is initialized at run time, some type of synchronization is needed to ensure that all PEs in the \( Active\ set \) have initialized \( pSync \) before any of them enter an OpenSHMEM routine called with the \( pSync \) synchronization array. A \( pSync \) array may be reused on a subsequent OpenSHMEM \( shmem\_alltoalls \) routine only if none of the PEs in the \( Active\ set \) are still processing a prior OpenSHMEM \( shmem\_alltoalls \) routine call that used the same \( pSync \) array. In general, this can be ensured only by doing some type of synchronization.

EXAMPLES

This example shows a \( shmem\_alltoalls64 \) on two long elements among all PEs.
#include <shmem.h>
#include <stdio.h>

long pSync[SHMEM_ALLTOALLS_SYNC_SIZE];

int main(void)
{
    int64_t *source, *dest;
    int i, count, pe;
    shmem_init();

    count = 2;
    dest = (int64_t*) shmem_malloc(count * shmem_n_pes() * sizeof(int64_t));
    source = (int64_t*) shmem_malloc(count * shmem_n_pes() * sizeof(int64_t));

    /* assign source values */
    for (pe=0; pe<shmem_n_pes(); pe++)
        for (i=0; i<count; i++)
            source[(pe*count)+i] = shmem_my_pe() + pe;
    for (i=0; i<count; i++)
        dest[(pe*count)+i] = 9999;

    for (i=0; i< SHMEM_ALLTOALLS_SYNC_SIZE; i++) {
        pSync[i] = SHMEM_SYNC_VALUE;
    }
    /* wait for all PEs to initialize pSync */
    shmem_barrier_all();

    /* alltoalls on all PES */
    shmem_alltoalls64(dest, source, 1, 1, count, 0, 0, shmem_n_pes(), pSync);

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

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

8.7 Point-To-Point Synchronization Routines

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

8.7.1 SHMEM_WAIT

Wait for a variable on the local PE to change.

SYNOPSIS

C/C++:
void shmem_int_wait(volatile int *ivar, int cmp_value);
void shmem_int_wait_until(volatile int *ivar, int cmp, int cmp_value);
void shmem_long_wait(volatile long *ivar, long cmp_value);
void shmem_long_wait_until(volatile long *ivar, int cmp, long cmp_value);
void shmem_longlong_wait(volatile long long *ivar, long long cmp_value);
void shmem_longlong_wait_until(volatile long long *ivar, int cmp, long long cmp_value);
void shmem_short_wait(volatile short *ivar, short cmp_value);
void shmem_short_wait_until(volatile short *ivar, int cmp, short cmp_value);
void shmem_wait(volatile long *ivar, long cmp_value);
void shmem_wait_until(volatile long *ivar, int cmp, long cmp_value);

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

DESCRIPTION

Arguments

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

API description

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>shmem_wait, shmem_wait_until</td>
<td>default INTEGER</td>
</tr>
<tr>
<td>shmem_int4_wait, shmem_int4_wait_until</td>
<td>INTEGER*4</td>
</tr>
</tbody>
</table>
The following `cmp` values are supported:

<table>
<thead>
<tr>
<th>CMP Value</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_CMP_EQ</td>
<td>Equal</td>
</tr>
<tr>
<td>SHMEM_CMP_NE</td>
<td>Not equal</td>
</tr>
<tr>
<td>SHMEM_CMP_GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>SHMEM_CMP_LE</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>SHMEM_CMP_LT</td>
<td>Less than</td>
</tr>
<tr>
<td>SHMEM_CMP_GE</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

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

```c
#include "shmem.fh"

integer*8 ivar

call shmem_int8_wait(ivar, integer*8(100))
```

The following call to `SHMEM_INT8_WAIT_UNTIL` is equivalent to the call to `SHMEM_INT8_WAIT` in example 1:

```c
#include "shmem.fh"

integer*8 ivar

call shmem_int8_wait_until(ivar, shmem_cmp_ne, integer*8(100))
```

The following `C/C++` call waits until the sign bit in `ivar` is set by a transfer from a remote PE:
#include <stdio.h>
#include <shmem.h>

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

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

INCLUDE "shmem.fh"

SUBROUTINE EXAMPLE()
  INTEGER FLAG_VAR
  COMMON/FLAG/FLAG_VAR

  ..
  FLAG_VAR = FLAG_VALUE  ! initialize the event variable
  ..
  IF (FLAG_VAR .EQ. FLAG_VALUE) THEN
    CALL SHMEM_WAIT(FLAG_VAR, FLAG_VALUE)
  ENDIF
  FLAG_VAR = FLAG_VALUE  ! reset the event variable for next time
  ..
END

8.8 Memory Ordering Routines

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

8.8.1 SHMEM_FENCE

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

SYNOPSIS

C/C++:
void shmem_fence(void);

FORTRAN:
CALL SHMEM_FENCE

DESCRIPTION

Arguments
None.

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

Return Values
None.
Notes

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

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

EXAMPLES

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

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

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

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

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

8.8.2 SHMEM_QUIET

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

SYNOPSIS

C/C++:
```c
void shmem_quiet(void);
```

FORTRAN:
```fortran
CALL SHMEM_QUIET
```

DESCRIPTION

Arguments

None.

API description

The shmem_quiet routine ensures completion of Put, AMOs, memory store, and non-blocking Put and
Get routines on symmetric data objects issued by the calling PE. All Put, AMOs, memory store, and non-blocking Put and Get routines to symmetric data objects are guaranteed to be completed and visible to all PEs when shmem_quiet returns.

Return Values

None.

Notes

shmem_quiet is most useful as a way of ensuring completion of several Put, AMOs, memory store, and non-blocking Put and Get routines to symmetric data objects initiated by the calling PE. For example, you might use shmem_quiet to await delivery of a block of data before issuing another Put or non-blocking Put routine, which sets a completion flag on another PE. shmem_quiet is not usually needed if shmem_barrier_all or shmem_barrier are called. The barrier routines wait for the completion of outstanding writes (Put, AMO, memory stores, and nonblocking Put and Get routines) to symmetric data objects on all PEs.

EXAMPLES

The following example uses shmem_quiet in a C/C++ program:

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

long dest[3] = {0};
int targ = 0;
long source[3] = {1, 2, 3};
int src = 90;

int main(void)
{
    long x[3] = {0};
    int y = 0;

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

        shmem_quiet();

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

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

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

8.8.3 Synchronization and Communication Ordering in OpenSHMEM

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

<table>
<thead>
<tr>
<th>OpenSHMEM API</th>
<th>Working of OpenSHMEM API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-point synchronization</td>
<td></td>
</tr>
<tr>
<td><em>shmem_wait</em>, <em>shmem_wait_until</em></td>
<td></td>
</tr>
<tr>
<td>PE 0</td>
<td><img src="image" alt="Diagram of shmem_int_p" /></td>
</tr>
<tr>
<td>PE 1</td>
<td><img src="image" alt="Diagram of shmem_int_wait_until" /></td>
</tr>
<tr>
<td></td>
<td><em>shmem_wait</em> is a blocking operation therefore it waits until value in addr is updated</td>
</tr>
<tr>
<td></td>
<td><em>The addr is updated to value</em></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Ordering puts issued by a local PE</td>
<td><strong>Ordering puts issued by a local PE</strong></td>
</tr>
<tr>
<td><em>shmem_fence</em></td>
<td></td>
</tr>
<tr>
<td>PE 2</td>
<td><img src="image" alt="Diagram of shmem_int_p" /></td>
</tr>
<tr>
<td>PE 0</td>
<td><img src="image" alt="Diagram of shmem_fence" /></td>
</tr>
<tr>
<td>PE 1</td>
<td><img src="image" alt="Diagram of shmem_int_p" /></td>
</tr>
<tr>
<td></td>
<td><em>value2 is delivered to PE2, before value5</em></td>
</tr>
<tr>
<td></td>
<td><em>value5 will be delivered after value2</em></td>
</tr>
<tr>
<td></td>
<td><em>value1 and value3 are delivered to PE1, before value4</em></td>
</tr>
<tr>
<td></td>
<td><em>value4 will be delivered after value1 and value3</em></td>
</tr>
<tr>
<td></td>
<td><strong>All Put routines, AMOs and store routines on symmetric data issued to same PE are guaranteed to be delivered before Puts (to the same PE) issued after the fence call.</strong></td>
</tr>
</tbody>
</table>
### Working of OpenSHMEM API

**Ordering puts issued by all PE**

```latex
\textit{shmem\_quiet}
```

---

**Collective synchronization over an Active set**

```latex
\textit{shmem\_barrier}
```
## OpenSHMEM API

<table>
<thead>
<tr>
<th>OpenSHMEM API</th>
<th>Working of OpenSHMEM API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collective synchronization over all PEs</td>
<td>All PEs</td>
</tr>
<tr>
<td><code>shmem_barrier_all</code></td>
<td></td>
</tr>
</tbody>
</table>

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.
8.9 Distributed Locking Routines

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

8.9.1 SHMEM_LOCK

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

SYNOPSIS

C/C++:

void shmem_clear_lock(volatile long *lock);

void shmem_set_lock(volatile long *lock);

int shmem_test_lock(volatile long *lock);

FORTRAN:

INTEGER lock, SHMEM_TEST_LOCK

CALL SHMEM_CLEAR_LOCK(lock)

CALL SHMEM_SET_LOCK(lock)

I = SHMEM_TEST_LOCK(lock)

DESCRIPTION

Arguments

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. If you are using Fortran, it must be of default kind.

API description

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

Return Values

The shmem_test_lock routine returns 0 if the lock was originally cleared and this call was able to set the lock. A value of I is returned if the lock had been set and the call returned without waiting to set the lock.

Notes

The term symmetric data object is defined in Section 3. The lock variable should always be initialized to zero and accessed only by the OpenSHMEM locking API. Changing the value of the lock variable by other means without using the OpenSHMEM API, can lead to undefined behavior.

EXAMPLES

The following example uses shmem_lock in a C program.
```c
#include <stdio.h>
#include <unistd.h>
#include <shmem.h>

long L = 0;

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

### 8.10 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.

#### 8.10.1 SHMEM_CACHE

Controls data cache utilities.

**SYNOPSIS**

**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);
```

**FORTRAN:**

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

**DESCRIPTION**

**Arguments**

| IN dest | A data object that is local to the PE. dest can be of any noncharacter type. If you are using Fortran, it can be of any kind. |
API description

shmem_set_cache_inv enables automatic cache coherency mode.

shmem_set_cache_line_inv enables automatic cache coherency mode for the cache line associated with the address of dest only.

shmem_clear_cache_inv disables automatic cache coherency mode previously enabled by shmem_set_cache__inv or shmem_set_cache_line_inv.

shmem_udcflush makes the entire user data cache coherent.

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

Return Values

None.

Notes

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

EXAMPLES

None.
Annex A

Writing OpenSHMEM Programs

Incorporating OpenSHMEM into Programs

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

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

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

int main (int argc, char *argv[])
{
    int nprocs, me;
    shmem_init ();
    nprocs = shmem_n_pes ();
    me = shmem_my_pe ();
    printf ("Hello from %d of %d\n", me, nprocs);
    return 0;
}
```

Listing A.1: Expected Output (4 processors)

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

OpenSHMEM also has a Fortran API, so for completeness we will now give the same program written in Fortran, in listing A:
program hello
  include "shmem.fh"
  integer :: shmem_my_pe, shmem_n_pes
  integer :: npes, me
  call shmem_init ()
  npes = shmem_n_pes ()
  me = shmem_my_pe ()
  write (*, 1000) me, npes
  1000 format ('Hello from', 1X, I4, 1X, 'of', 1X, I4)
end program hello

Listing A.2: Expected Output (4 processors)

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

The following example shows a more complex OpenSHMEM program that illustrates the use of symmetric data objects. Note the declaration of the static short dest array and its use as the remote destination in OpenSHMEM short Put. The use of the static keyword results in the dest array being symmetric on PE 0 and PE 1. Each PE is able to transfer data to the dest array by simply specifying the local address of the symmetric data object which is to receive the data. This aids programmability, as the address of the dest need not be exchanged with the active side (PE 0) prior to the RMA (Remote Memory Access) routine. Conversely, the declaration of the short source array is asymmetric. Because the Put handles the references to the source array only on the active (local) side, the asymmetric source object is handled correctly.
#include <stdio.h>
#include <shmem.h>
#define SIZE 16

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

Listing A.3: Expected Output (4 processors)

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

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

1 Compilation

Programs written in C

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

oshcc <compiler options> -o myprogram myprogram.c

Where the (compiler options) are options understood by the underlying C compiler.

Programs written in C++

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

oshCC <compiler options> -o myprogram myprogram.cpp

Where the (compiler options) are options understood by the underlying C++ compiler called by oshCC.

Programs written in Fortran

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

oshfort <compiler options> -o myprogram myprogram.f

Where the (compiler options) are options understood by the underlying Fortran compiler called by oshfort.

2 Running Programs

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

oshrun <additional options> -np <#> <program> <program arguments>
The program arguments for `oshrun` are:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨additional options⟩</td>
<td>Options passed to the underlying launcher.</td>
</tr>
<tr>
<td>-np ⟨#⟩</td>
<td>The number of PEs to be used in the execution.</td>
</tr>
<tr>
<td>⟨program⟩</td>
<td>The program executable to be launched.</td>
</tr>
<tr>
<td>⟨program arguments⟩</td>
<td>Flags and other parameters to pass to the program.</td>
</tr>
</tbody>
</table>
Annex C

Undefined Behavior in OpenSHMEM

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

<table>
<thead>
<tr>
<th>Inappropriate Usage</th>
<th>Undefined Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized library</td>
<td>If OpenSHMEM is not initialized through a call to <code>shmem_init</code>, subsequent accesses to OpenSHMEM routines have undefined results. An implementation may choose, for example, to try to continue or abort immediately upon the first call to an uninitialized routine.</td>
</tr>
<tr>
<td>Accessing non-existent PEs</td>
<td>If a communications routine accesses a non-existent PE, then the OpenSHMEM library can choose to handle this situation in an implementation-defined way. For example, the library may issue an error message saying that the PE accessed is outside the range of accessible PEs, or may exit without a warning.</td>
</tr>
<tr>
<td>Use of non-symmetric variables</td>
<td>Some routines require remotely accessible variables to perform their function. A <code>Put</code> to a non-symmetric variable can be trapped where possible and the library can abort the program. Another implementation may choose to continue either with a warning or silently.</td>
</tr>
<tr>
<td>Non-symmetric variables</td>
<td>The symmetric memory management routines are collectives, which means that all PEs in the program must issue the same <code>shmemp_malloc</code> call with the same size request. Program behavior after a mismatched <code>shmemp_malloc</code> call is undefined.</td>
</tr>
<tr>
<td>Use of NULL pointers with non-zero <code>len</code> specified</td>
<td>In any OpenSHMEM routine that takes a pointer and <code>len</code> describing the number of elements in that pointer, NULL may not be specified for the pointer unless the corresponding <code>len</code> is also specified as zero. Otherwise, the resulting behavior is undefined. The following cases summarize this behavior:</td>
</tr>
<tr>
<td></td>
<td>• <code>len</code> is 0, pointer is NULL: supported.</td>
</tr>
<tr>
<td></td>
<td>• <code>len</code> is not 0, pointer is NULL: undefined behavior.</td>
</tr>
<tr>
<td></td>
<td>• <code>len</code> is 0, pointer is not NULL: supported.</td>
</tr>
<tr>
<td></td>
<td>• <code>len</code> is not 0, pointer is not NULL: supported.</td>
</tr>
<tr>
<td>Multiple calls to <code>shmemp_init</code></td>
<td>In an OpenSHMEM program where <code>shmemp_init</code> has already be called, any subsequent calls to <code>shmemp_init</code> result in undefined behavior.</td>
</tr>
</tbody>
</table>
Annex D

Interoperability with other Programming Models

1 MPI Interoperability

OpenSHMEM routines can be used in conjunction with MPI routines in the same program. For example, on SGI systems, programs that use both MPI and OpenSHMEM routines call `MPI_Init` and `MPI_Finalize` but omit the call to the `shmem_init` routine. OpenSHMEM PE numbers are equal to the MPI rank within the `MPI_COMM_WORLD` environment variable. Note that this precludes use of OpenSHMEM routines between processes in different `MPI_COMM_WORLDs`. MPI processes started using the `MPI_Comm_spawn` routine, for example, cannot use OpenSHMEM routines to communicate with their parent MPI processes.

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

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

History of OpenSHMEM

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

• A SHMEM Timeline
  – Cray SHMEM
    * SHMEM first introduced by Cray Research Inc. in 1993 for Cray T3D
    * Cray is acquired by SGI in 1996
    * Cray is acquired by Tera in 2000 (MTA)
    * Platforms: Cray T3D, T3E, C90, J90, SV1, SV2, X1, X2, XE, XMT, XT
  – SGI SHMEM
    * SGI purchases Cray Research Inc. and SHMEM was integrated into SGI’s Message Passing Toolkit (MPT)
    * SGI currently owns the rights to SHMEM and OpenSHMEM
    * Platforms: Origin, Altix 4700, Altix XE, ICE, UV
    * SGI was purchased by Rackable Systems in 2009
    * SGI and Open Source Software Solutions, Inc. (OSSS) signed a SHMEM trademark licensing agreement, in 2010
  – Other Implementations
    * Quadrics (Vega UK, Ltd.)
    * Hewlett Packard
    * GPSHMEM
    * IBM
    * QLogic
    * Mellanox
    * University of Florida

• OpenSHMEM Implementations
  – SGI OpenSHMEM
  – University of Houston - OpenSHMEM Reference Implementation
  – Mellanox ScalableSHMEM
  – Portals-SHMEM
  – IBM OpenSHMEM
Annex F

OpenSHMEM Specification and Deprecated API

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

<table>
<thead>
<tr>
<th>Deprecated API</th>
<th>Deprecated Since</th>
<th>Currently Supported(?)</th>
<th>Replaced By</th>
</tr>
</thead>
<tbody>
<tr>
<td>_my_pe</td>
<td>1.2</td>
<td>Yes</td>
<td>shmem_my_pe</td>
</tr>
<tr>
<td>_num_pes</td>
<td>1.2</td>
<td>Yes</td>
<td>shmem_n_pes</td>
</tr>
<tr>
<td>shmalloc</td>
<td>1.2</td>
<td>Yes</td>
<td>shmem_malloc</td>
</tr>
<tr>
<td>shfree</td>
<td>1.2</td>
<td>Yes</td>
<td>shmem_free</td>
</tr>
<tr>
<td>shrealloc</td>
<td>1.2</td>
<td>Yes</td>
<td>shmem_realloc</td>
</tr>
<tr>
<td>shmemalign</td>
<td>1.2</td>
<td>Yes</td>
<td>shmem_align</td>
</tr>
<tr>
<td>start_pes</td>
<td>1.2</td>
<td>Yes</td>
<td>shmem_init</td>
</tr>
<tr>
<td>SHMEM_PUT</td>
<td>1.2</td>
<td>Yes</td>
<td>SHMEM_PUT8 or SHMEM_PUT64</td>
</tr>
<tr>
<td>SHMEM_CACHE</td>
<td>1.3</td>
<td>Yes</td>
<td>(none)</td>
</tr>
<tr>
<td><em>SHMEM</em>* constants</td>
<td>1.3</td>
<td>Yes</td>
<td>(none)</td>
</tr>
</tbody>
</table>
Annex G

Changes to this Document

1 Version 1.3

This section summarizes the changes from the OpenSHMEM specification Version 1.2 to Version 1.3. Many major changes to the specification was introduced in Version 1.3. This includes non-blocking RMA operations, generic interfaces for various OpenSHMEM interfaces, atomic Put and Get operations, and Alltoall interfaces.

The following list describes the specific changes in 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 4.2.
- 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 8.3, 8.4 and 8.5.
- New non-blocking variants of remote memory access, `SHMEM_PUT_NBI` and `SHMEM_GET_NBI`. See Sections 8.4.1 and 8.4.2.
- New atomic elemental read and write operations, `SHMEM_FETCH` and `SHMEM_SET`. See Sections 8.5.7 and 8.5.8
- New alltoall data exchange operations, `SHMEM_ALLTOALL` and `SHMEM_ALLTOALLS`. See Sections 8.6.6 and 8.6.7.
- Added `volatile` to remotely accessible pointer argument in `SHMEM_WAIT` and `SHMEM_LOCK`. See Sections 8.7.1 and 8.9.1.
- Deprecation of `SHMEM_CACHE`. See Section 8.10.1.
2 Version 1.2

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

The following list describes the specific changes in 1.2:

- Added specification of \texttt{pSync} initialization for all routines that use it.

- Replaced all placeholder variable names \texttt{target} with \texttt{dest} to avoid confusion with Fortran ‘target’ keyword.

- New Execution Model for exiting/finishing OpenSHMEM programs. See Section 4.

- New library constants to support API that query version and name information. See Section 6.

- New API \texttt{shmem\_init} to provide mechanism to start an OpenSHMEM program and replace deprecated \texttt{start\_pes}. See Section 8.1.1.

- Deprecation of \texttt{\_my\_pe} and \texttt{\_num\_pes} routines. See Sections 8.1.2 and 8.1.3.

- New API \texttt{shmem\_finalize} to provide collective mechanism to cleanly exit an OpenSHMEM program and release resources. See Section 8.1.4.

- New API \texttt{shmem\_global\_exit} to provide mechanism to exit an OpenSHMEM program. See Section 8.1.5.

- Clarification related to the address of the referenced object in \texttt{shmem\_ptr}. See Section 8.1.8.

- New API to query the version and name information. See Section 8.1.9 and 8.1.10.

- OpenSHMEM library API normalization. All C symmetric memory management API begins with \texttt{shmem\_}. See Section 8.2.1.

- Notes and clarifications added to \texttt{shmem\_malloc}. See Section 8.2.1.

- Deprecation of Fortran API routine \texttt{SHMEM\_PUT}. See Section 8.3.1.

- Clarification related to \texttt{shmem\_wait}. See Section 8.7.1.

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

- Addition of new Annex for clearly specifying deprecated API and its support in the existing specification version. See Annex F.
3 Version 1.1

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

The following list describes the specific changes in 1.1:

- Clarifications of the completion semantics of memory synchronization interfaces.  
  See Section 8.8.

- Clarification of the completion semantics of memory load and store operations in context of shmem_barrier_all and shmem Barrier routines.  
  See Section 8.6.1 and 8.6.2.

- Clarification of the completion and ordering semantics of shmem_quiet and shmem_fence.  
  See Section 8.8.2 and 8.8.1.

- Clarifications of the completion semantics of RMA and AMO routines.  
  See Sections 8.3 and 8.5

- Clarifications of the memory model and the memory alignment requirements for symmetric data objects.  
  See Section 3.

- Clarification of the execution model and the definition of a PE.  
  See Section 4

- Clarifications of the semantics of shmem_pe_accessible and shmem_addr_accessible.  
  See Section 8.1.6 and 8.1.7.

- Added an annex on interoperability with MPI.  
  See Annex D.

- Added examples to the different interfaces.

- Clarification of the naming conventions for constant in C and Fortran.  
  See Section 6 and 8.7.1.

- Added API calls: shmem_char_p, shmem_char_g.  
  See Sections 8.3.2 and 8.3.5.

- Removed API calls: shmem_char_put, shmem_char_get.  
  See Sections 8.3.1 and 8.3.4.

- The usage of ptrdiff_t, size_t, and int in the interface signature was made consistent with the description.  
  See Sections 8.6, 8.3.3, and 8.3.6.

- Revised shmem_barrier example.  
  See Section 8.6.2.

- Clarification of the initial value of pSync work arrays for shmem_barrier.  
  See Section 8.6.2.

- Clarification of the expected behavior when multiple start_pes calls are encountered has been clarified.  
  See Section 8.1.11.

- Corrected the definition of atomic increment operation.  
  See Section 8.5.5.
• Clarification of the size of the symmetric heap and when it is set.
  See Section 8.2.1.

• Clarification of the integer and real sizes for *Fortran* API.
  See Sections 8.5.1, 8.5.2, 8.5.3, 8.5.4, 8.5.5, and 8.5.6.

• 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 8.2.1 and Annex C.

• Fortran programming fixes for examples.
  See Sections 8.6.5 and 8.7.1.

• Clarifications of the reuse *pSync* and *pWork* across collectives.
  See Sections 8.6, 8.6.3, 8.6.4 and 8.6.5.

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