Notes

Dynamic Memory Spaces Proposal for OpenSHMEM

This proposal introduces a dynamic approach to memory spaces that addresses the scalability concerns of the current environment variable-based approach. Instead of predefined spaces and environment variables for each device type, this proposal uses runtime API routines for space creation and management.

Processing Elements (PEs) and Device Access

Processing Element (PE): A PE is the OpenSHMEM execution unit that represents a logical process participating in the OpenSHMEM program. PEs are typically implemented using operating system processes and are permitted to create additional threads when supported by the OpenSHMEM library. PEs are identified by unique integers assigned in a monotonically increasing manner from zero to one less than the total number of PEs. A PE is not tied to a specific hardware component (CPU core, socket, or device). Every PE is a member of `SHMEM_TEAM_WORLD` and shall have access to the default space. The default space's associated team equals `SHMEM_TEAM_WORLD`. A PE may have access to additional memory spaces.

For a device type `D`, a PE "has access" if and only if the PE is a member of the team associated with a memory space created for `D`. Membership implies that the PE can allocate and free symmetric memory in that space and participate in OpenSHMEM operations that target that space according to implementation capabilities (e.g., RMA, collectives, and, when supported, atomics). Direct execution on device `D` is not required; access may be provided by the implementation (e.g., unified memory, remote access, or proxy mechanisms). The mapping between PEs and hardware resources is implementation-defined.

PE Creation and Space Creation: PEs are created at program startup (e.g., via `oshrun -np 8 ./app` creates 8 PEs total). Memory spaces are created at runtime by these existing PEs using `shmem_space_create()`. The `-np` argument specifies the total number of PEs in the job, not the number of PEs per space. The `shmem_space_create()` routine is collective on `SHMEM_TEAM_WORLD`. PEs that can access the requested device type form the returned space-associated team; on other PEs, the returned team equals `SHMEM_TEAM_INVALID` and the returned space handle equals `SHMEM_SPACE_INVALID`. This maintains the existing OpenSHMEM execution model while enabling heterogeneous device availability across PEs.

Core Design Principles

Spaces are created at runtime and are device-agnostic. Device selection, placement, and mapping are implementation-defined and may be topology-aware. Teams and spaces are orthogonal; collectives on a space use teams whose members are a subset of the space-associated team. Additional teams may be created from the space-associated team using existing team creation routines such as `shmem_team_split_2d` and `shmem_team_split_strided`.

Type Definitions

```
/**
 * @brief Device types supported by the OpenSHMEM implementation
 *
 * The specific device types supported are implementation-defined.
 * Implementations may define any device types appropriate for their
 * target hardware and use cases.
 *
 * Device type names must follow the convention:
SHMEM_DEVICE_<device_type>
 * where <device_type> is the name of the device (e.g., SHMEM_DEVICE_CPU,
 * SHMEM_DEVICE_GPU, SHMEM_DEVICE_FPGA, etc.).
 */
typedef enum {
  SHMEM_DEVICE_CPU = 0,
  // ...
} shmem_device_type_t;

/**
 * @brief Space configuration structure
 */
typedef struct {
  shmem_device_type_t device_type;   /**< Device type */
  size_t size;                       /**< Size per PE in bytes */
  int flags;                         /**< Creation flags */
} shmem_space_config_t;

/**
 * @brief Memory space handle
 */
typedef void *shmem_space_t;

/**
 * @brief Space creation flags
 */
#define SHMEM_SPACE_FLAG_DEFAULT    0x0000  /**< Default space behavior */
...
```

Capability Bitmask Type

```
typedef uint64_t shmem_space_cap_t;

#define SHMEM_SPACE_CAP_RMA           UINT64_C(0x0001)  /* RMA to this
space is supported */
#define SHMEM_SPACE_CAP_COLLECTIVES   UINT64_C(0x0002)  /* Collectives on
this space are supported */
#define SHMEM_SPACE_CAP_ATOMICS       UINT64_C(0x0004)  /* Atomics
targeting this space are supported */
#define SHMEM_SPACE_CAP_DIRECT_ACCESS UINT64_C(0x0008)  /* Direct
load/store access available on member PEs */
#define SHMEM_SPACE_CAP_WORLD_ACCESS  UINT64_C(0x0010)  /* Space-
```

```
  associated team equals SHMEM_TEAM_WORLD */
  #define SHMEM_SPACE_CAP_IDENT_ADDR     UINT64_C(0x0020)  /* Identical
  numeric addresses across member PEs */
```

Structure Field Descriptions

`shmem_space_config_t` Fields

`device_type` - Specifies the type of device for this memory space:

The specific device types supported are implementation-defined. Implementations may support any device types appropriate for their target hardware and use cases. Device type names must follow the convention: `SHMEM_DEVICE_<device_type>` where `<device_type>` is the name of the device (e.g., `SHMEM_DEVICE_CPU`, `SHMEM_DEVICE_GPU`, `SHMEM_DEVICE_FPGA`, etc.).

`size` - Specifies the amount of memory per PE to allocate for this space:

- Size in bytes for the space's symmetric heap
- Must be consistent across all PEs in the space team
- Implementation may round up to alignment requirements

`flags` - Specifies additional creation options:

- `SHMEM_SPACE_FLAG_DEFAULT`: Default space behavior (no special requirements)

Constants

The proposal defines several constants that represent invalid or special space handles. These constants are used throughout the API to indicate invalid states or special conditions.

```
  extern shmem_space_t SHMEM_SPACE_INVALID;  /**< Invalid space handle */
```

Runtime Device Discovery and Initialization

The OpenSHMEM runtime performs device discovery and initialization during the `shmem_init()` routine. This process is implementation-defined but follows a structured approach to identify available devices and their capabilities across the system.

Device Discovery Process

The discovery can cover multiple device types, but the specific device types supported are implementation-defined. Implementations may support any device types appropriate for their target hardware and use cases. Device type names must follow the convention: `SHMEM_DEVICE_<device_type>` where `<device_type>` is the name of the device.

Implementations are not required to support any particular device types. The default space may be located on any device type, and implementations may support any subset of device types based on their target

hardware and use cases. Device type names must follow the `SHMEM_DEVICE_<device_type>` naming convention.

Space Creation Mechanics

When `shmem_space_create()` is called, the runtime determines the set of PEs that can access the requested device type. On PEs that can access the device type, the routine returns a valid space handle and a valid space-associated team that includes exactly those PEs. On PEs that cannot access the device type, the routine returns `SHMEM_SPACE_INVALID` for the space handle and `SHMEM_TEAM_INVALID` for the team. The call is collective on `SHMEM_TEAM_WORLD`.

Creation succeeds and returns zero if the configuration is valid and the resulting space-associated team is non-empty. On member PEs, `*space` and `*team` are set to valid handles; on non-member PEs, `*space` is set to `SHMEM_SPACE_INVALID` and `*team` to `SHMEM_TEAM_INVALID`. If the configuration is invalid or the resulting space-associated team would be empty on all PEs, the routine returns a nonzero value and sets `*space` to `SHMEM_SPACE_INVALID` and `*team` to `SHMEM_TEAM_INVALID` on all PEs.

The runtime allocates device resources for the specified device type on the member PEs, creating a symmetric memory pool within the space. Numeric pointer values for symmetric allocations in device spaces are not required to be identical across PEs. The symmetric property is defined by the existence of the allocation across the member PEs of the space-associated team with identical size and layout.

Default Space Selection

The device type of the default space is implementation-defined. The default space is selected during `shmem_init()` and remains immutable for the duration of the OpenSHMEM program. The default space shall be world-accessible (its space-associated team equals `SHMEM_TEAM_WORLD`). If a user-facing selection mechanism is provided (e.g., an environment variable), the selection shall result in a world-accessible default; otherwise, the implementation shall choose a world-accessible default space.

Team/Space Relationships

How Teams Relate to Spaces

Teams and spaces are orthogonal objects in OpenSHMEM. When a space is created via `shmem_space_create()`, it returns a space-associated team that includes exactly the PEs that can access the space. This team can be used for collective operations on data in that space and can serve as the parent for creating additional teams using existing team creation routines such as `shmem_team_split_2d()` and `shmem_team_split_strided()`.

Contexts are derived from teams. Spaces do not serve as parents of teams or contexts.

All PEs participating in a collective must provide pointers to data allocated from the same space, and the collective must be performed on a team whose members are all members of that space-associated team. If pointers point to data from different spaces, the behavior is undefined.

Team Creation with Spaces

Applications may create additional teams from the space-associated team using existing team creation routines.

Important: Destroying a space does not destroy any teams. All teams created from the space-associated team, including the space-associated team itself, shall be destroyed explicitly by the application prior to destroying the space. If any such team exists at the time of `shmem_space_destroy`, the routine returns a nonzero value and performs no operation.

```
/* Create space and get associated team */
shmem_space_t space;
shmem_team_t space_team;
shmem_space_config_t config = {
    .device_type = SHMEM_DEVICE_CPU,
    .size = 1024 * 1024,
    .flags = SHMEM_SPACE_FLAG_DEFAULT
};

if (shmem_space_create(&config, &space, &space_team) == 0 &&
shmem_team_is_valid(space_team)) {
    /* Create a 2D team from the space team */
    shmem_team_t team_2d;
    if (shmem_team_split_2d(space_team, 2, 2, &team_2d) == 0) {
        /* Use the 2D team for collective operations on space data */
        size_t nelems = 16;
        int *data = shmem_space_malloc(space, nelems * sizeof(int));
        shmem_int_broadcast(team_2d, data, data, nelems, 0);
        shmem_team_destroy(team_2d);
    }

    /* Destroy space-associated team and then the space */
    shmem_team_destroy(space_team);
    shmem_space_destroy(space);
}
```

SHMEM_TEAM_WORLD and Memory Spaces

`SHMEM_TEAM_WORLD` includes all PEs in the job. Collective operations on buffers allocated from a space `S` shall be performed only on teams whose members are all members of the team associated with `S`. `SHMEM_TEAM_WORLD` may be used only when space `S` is world-accessible (its associated team equals `SHMEM_TEAM_WORLD`). The default space is world-accessible by requirement; therefore, world-team collectives on default-space buffers are valid.

Examples of Valid SHMEM_TEAM_WORLD Collectives:

```
/* All data from CPU space using SHMEM_TEAM_WORLD (only if the CPU space
is world-accessible) */
int *cpu_data = shmem_space_malloc(cpu_space, nelems * sizeof(int));
int *cpu_result = shmem_space_malloc(cpu_space, nelems * sizeof(int));
shmem_int_broadcast(SHMEM_TEAM_WORLD, cpu_result, cpu_data, nelems, 0);

/* All data from default space using SHMEM_TEAM_WORLD (backward
compatible) */
```

```
int *world_data = shmem_malloc(nelems * sizeof(int));
int *world_result = shmem_malloc(nelems * sizeof(int));
shmem_int_broadcast(SHMEM_TEAM_WORLD, world_result, world_data, nelems,
0);
```

Examples of Invalid SHMEM_TEAM_WORLD Collectives:

```
/* Mixed-space collective - undefined behavior */
int *cpu_data = shmem_space_malloc(cpu_space, nelems * sizeof(int));
int *gpu_data = shmem_space_malloc(gpu_space, nelems * sizeof(int));
/* This would be undefined if different PEs pass different space pointers
*/
shmem_int_broadcast(SHMEM_TEAM_WORLD, cpu_data, gpu_data, nelems, 0);
```

Collectives

Collective Operations

Collective operations require all PEs to use data from the same memory space. The runtime detects the space of the data and routes the operation accordingly.

Single-Space Requirement:

Collective operations must operate on data from a single memory space. All PEs participating in a collective must provide pointers to data allocated from the same space, and the collective must be performed on a team whose members are all members of that space-associated team. The runtime detects the memory space of the data via pointer type checking.

Conformance (Synchronization Semantics):

For space-aware allocation and deallocation routines, synchronization is defined as follows:

- On successful return from `shmem_space_malloc` and `shmem_space_calloc`, the synchronization effect shall be semantically equivalent to invoking `shmem_team_sync(space_team)` on the space-associated team.
- On entry to `shmem_space_free` when `ptr` is not a null pointer, the routine shall perform a synchronization that is semantically equivalent to invoking `shmem_team_sync(space_team)` on the space-associated team.

Cross-Device Access in Collectives:

PEs participate in collectives on a space only if they are members of the space-associated team. The key requirement is that all PEs in the collective team are members of the space-associated team and all buffers are allocated from that space.

Collective Space Detection:

When a collective is called, the runtime determines which space the data belongs to. If all pointers point to data from the same space, the collective proceeds normally. If pointers point to data from different spaces,

the behavior is undefined.

RMA Operations

RMA operations may target buffers allocated in any space. The runtime routes operations based on the destination buffer's space. Validity is capability-gated: if RMA targeting a given space is not advertised via capabilities, behavior is undefined.

Atomic Operations

Atomic operations targeting buffers in a space are capability-gated and implementation-defined. If atomics targeting a given space are not advertised via capabilities, behavior is undefined.

Undefined Behavior in Memory Spaces (Add to Annex C)

The following cases are undefined behavior specific to space-aware routines:

Inappropriate Usage	**Undefined Behavior**
Use of invalid space handles	In OpenSHMEM space-aware routines that accept a space handle, if the space handle is invalid (e.g., `SHMEM_SPACE_INVALID`), the behavior is undefined.
Mismatched space allocation and free	Memory allocated from a specific space must be freed using the same space handle. If `shmem_space_free` is called with a different space handle than was used for allocation, the behavior is undefined.
Non-symmetric space allocation	Space-aware memory allocation routines are collectives on the space-associated team. If participating PEs provide non-identical `space` or size/count arguments, program behavior after the call is undefined.
Participation by non-members	If a PE that is not a member of the space-associated team calls a space-aware collective (allocation or free) for that space, the behavior is undefined.
Invalid pointer in space query	If `shmem_get_space` is called with a pointer that was not allocated through OpenSHMEM symmetric memory allocation routines, the behavior is undefined.

Space API Routines

X.X SHMEM_SPACE_CREATE

Create a new memory space with specified configuration and associated team.

SYNOPSIS

C/C++:

```
int shmem_space_create(const shmem_space_config_t *config,
                       shmem_space_t *space, shmem_team_t *team);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
`config`	IN	Configuration for the new space
`space`	OUT	Handle to the newly created space
`team`	OUT	Team associated with the new space (collective)

API Description

The `shmem_space_create` routine is a collective operation on `SHMEM_TEAM_WORLD` that creates a new memory space according to the provided configuration and returns a space-associated team. All PEs must call this routine with identical `config` arguments; otherwise, the behavior is undefined. On PEs that can access the requested device type, the routine returns a valid space handle and a valid space-associated team that includes exactly those PEs. On PEs that cannot access the requested device type, the routine returns `SHMEM_SPACE_INVALID` and `SHMEM_TEAM_INVALID`.

Note: There is exactly one default space in an OpenSHMEM program. The default space is automatically created during `shmem_init()` and is sized according to the `SHMEM_SYMMETRIC_SIZE` environment variable. Requirements and constraints for selecting the default space are specified in the Default Space Selection subsection. Only additional spaces beyond the default space need to be created with `shmem_space_create`.

The `config` structure specifies the device type, size per PE, and creation flags for the new space. The device type indicates which type of device the space should be created on. Device type names must follow the convention: `SHMEM_DEVICE_<device_type>` where `<device_type>` is the name of the device. Device selection, placement, and mapping within a space are topology-aware and implementation-defined. The size parameter specifies the amount of memory per PE to allocate for this space. The flags parameter allows applications to specify additional creation options.

The returned team serves as the parent team for the newly created space and can be used with existing team creation routines such as `shmem_team_split_2d` and `shmem_team_split_strided` to create additional teams for that space.

If the device type in the config is invalid, if the resulting space-associated team would be empty, or if the size exceeds the memory capacity of the device, the routine returns a nonzero value and sets `*space` to `SHMEM_SPACE_INVALID` and `*team` to `SHMEM_TEAM_INVALID` on all PEs.

Return Values

Zero on success; otherwise, nonzero.

X.X SHMEM_SPACE_DESTROY

Destroy a memory space and free associated resources.

SYNOPSIS

C/C++:

```
int shmem_space_destroy(shmem_space_t space);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
space	IN	Memory space to destroy

API Description

The shmem_space_destroy routine is a collective operation on the space-associated team that destroys the specified memory space and frees all associated resources owned by the space. Only PEs that are members of the space-associated team shall call this routine. Other PEs perform no operation.

Memory allocated from the space must be freed before destroying the space. If any memory is still allocated from the space, the behavior is undefined.

Teams are not destroyed by this routine. All teams created from the space-associated team, including the space-associated team itself, shall be destroyed explicitly by the application prior to calling shmem_space_destroy. If any such team exists at the time of the call, the routine returns a nonzero value and performs no operation.

If space compares equal to SHMEM_SPACE_INVALID, no operation is performed. If space is otherwise invalid, the behavior is undefined.

Return Values

Zero on success; otherwise, nonzero.

X.X SHMEM_SPACE_MALLOC

Allocate memory in a specific memory space.

SYNOPSIS

C/C++:

```
void *shmem_space_malloc(shmem_space_t space, size_t size);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
`space`	IN	An OpenSHMEM memory space handle
`size`	IN	The size, in bytes, of the memory to allocate

API Description

The `shmem_space_malloc` routine is a collective operation on the space-associated team and returns the symmetric address of a block of at least `size` bytes from the specified memory space, which shall be suitably aligned so that it may be assigned to a pointer to any type of object. This space is allocated from the symmetric memory within the specified space (in contrast to `malloc`, which allocates from the private heap). Only PEs that are members of the space-associated team shall call this routine. When `size` is zero, the `shmem_space_malloc` routine performs no action and returns a null pointer; otherwise, `shmem_space_malloc` calls a procedure that is semantically equivalent to a barrier on the space-associated team on exit. The values of the `space` and `size` arguments must be identical on all PEs in the space-associated team; otherwise, the behavior is undefined.

Memory allocated from a space can only be freed using `shmem_space_free` with the same space handle.

If `space` compares equal to `SHMEM_SPACE_INVALID`, then a null pointer is returned. If `space` is otherwise invalid, the behavior is undefined.

Return Values

The `shmem_space_malloc` routine returns the symmetric address of the allocated space; otherwise, it returns a null pointer.

X.X SHMEM_SPACE_CALLOC

Allocate and zero-initialize memory in a specific memory space.

SYNOPSIS

C/C++:

```
void *shmem_space_calloc(shmem_space_t space, size_t count, size_t size);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
`space`	IN	An OpenSHMEM memory space handle

Parameter	Direction	Description
`count`	IN	The number of elements to allocate
`size`	IN	The size, in bytes, of each element

API Description

The `shmem_space_calloc` routine is a collective operation on the space-associated team that allocates a region of remotely accessible memory for an array of `count` objects of `size` bytes each from the specified memory space and returns a pointer to the lowest byte address of the allocated symmetric memory. Only PEs that are members of the space-associated team shall call this routine. The space is initialized to all bits zero.

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

The values for `space`, `count`, and `size` shall each be equal across all PEs in the space-associated team calling `shmem_space_calloc`; otherwise, the behavior is undefined.

Memory allocated from a space can only be freed using `shmem_space_free` with the same space handle.

When `count` or `size` is 0, the `shmem_space_calloc` routine returns without performing a barrier. Otherwise, this routine calls a procedure that is semantically equivalent to a barrier on the space-associated team on exit.

If `space` compares equal to `SHMEM_SPACE_INVALID`, then a null pointer is returned. If `space` is otherwise invalid, the behavior is undefined.

Return Values

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

X.X SHMEM_SPACE_FREE

Free memory allocated in a specific memory space.

SYNOPSIS

C/C++:

```
void shmem_space_free(shmem_space_t space, void *ptr);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
`space`	IN	An OpenSHMEM memory space handle
`ptr`	IN	A pointer to memory to free

API Description

The `shmem_space_free` routine is a collective operation on the space-associated team that causes the block to which `ptr` points to be deallocated from the specified memory space, that is, made available for further allocation within that space. Only PEs that are members of the space-associated team shall call this routine. If `ptr` is a null pointer, no action is performed; otherwise, `shmem_space_free` calls a barrier on entry on the space-associated team. It is the user's responsibility to ensure that no communication operations involving the given memory block are pending on other communication contexts prior to calling `shmem_space_free`. The `ptr` argument must be a pointer that was returned by `shmem_space_malloc` or `shmem_space_calloc` using the same space handle.

The values of the `space` and `ptr` arguments must be identical on all PEs in the space-associated team; otherwise, the behavior is undefined.

If `space` compares equal to `SHMEM_SPACE_INVALID`, no operation is performed. If `space` is otherwise invalid, or if `ptr` was not allocated from the specified space, the behavior is undefined.

Return Values

None.

EXAMPLE

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

int main(void) {
  shmem_init();

  shmem_space_t space;
  shmem_team_t space_team;
  shmem_space_config_t cfg = {
      .device_type = SHMEM_DEVICE_CPU,
      .size = 128 * 1024 * 1024,
      .flags = SHMEM_SPACE_FLAG_DEFAULT};

  if (shmem_space_create(&cfg, &space, &space_team) == 0 &&
shmem_team_is_valid(space_team)) {
    size_t nelems = 16;
    int *a = (int *) shmem_space_malloc(space, nelems * sizeof(int));
    int *b = (int *) shmem_space_calloc(space, nelems, sizeof(int));

    shmem_team_sync(space_team);

    shmem_space_free(space, a);
```

```
    shmem_space_free(space, b);
    shmem_team_destroy(space_team);
    shmem_space_destroy(space);
  }

  shmem_finalize();
  return 0;
}
```

X.X SHMEM_SPACE_GET_TEAM

Get the team associated with a memory space.

SYNOPSIS

C/C++:

```
int shmem_space_get_team(shmem_space_t space, shmem_team_t *team);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
`space`	IN	An OpenSHMEM memory space handle
`team`	OUT	The team associated with the given space

API Description

The `shmem_space_get_team` routine returns in `*team` the team associated with `space`. On PEs that are not members of the space-associated team, `*team` is set to `SHMEM_TEAM_INVALID`.

If `space` compares equal to `SHMEM_SPACE_INVALID`, the routine returns a nonzero value and the contents of `*team` are undefined. If `team` is a null pointer, the behavior is undefined.

Return Values

Zero on success; otherwise, nonzero.

X.X SHMEM_SPACE_GET_DEVICE_TYPE

Get the device type of a memory space.

SYNOPSIS

C/C++:

```
int shmem_space_get_device_type(shmem_space_t space, shmem_device_type_t
*type);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
space	IN	An OpenSHMEM memory space handle
type	OUT	The device type of the given space

API Description

The shmem_space_get_device_type routine stores in *type the device type on which space was created. If space compares equal to SHMEM_SPACE_INVALID, the routine returns a nonzero value and the contents of *type are undefined. If type is a null pointer, the behavior is undefined.

Return Values

Zero on success; otherwise, nonzero.

X.X SHMEM_SPACE_GET_CAPS

Get capability flags for a memory space.

SYNOPSIS

C/C++:

```
int shmem_space_get_caps(shmem_space_t space, shmem_space_cap_t *caps);
```

DESCRIPTION

Arguments

Parameter	Direction	Description
space	IN	An OpenSHMEM memory space handle
caps	OUT	Capability bitmask for the space

API Description

The shmem_space_get_caps routine stores in *caps a bitwise-OR of capability flags describing operations and properties supported for space. If space compares equal to SHMEM_SPACE_INVALID, the

routine returns a nonzero value and the contents of `*caps` are undefined. If `caps` is a null pointer, the behavior is undefined.

Capabilities include, but are not limited to: support for RMA, collectives, atomics, direct access on member PEs, world accessibility, and identical numeric addresses across member PEs.

Return Values

Zero on success; otherwise, nonzero.