

UCC: Unified Collective Communication API

[Library Handles and Structures](#)

[Library Initialization and Finalization](#)

[C Interface](#)

[Data Structures](#)

[Communication Context](#)

[C Interface](#)

[Data Structures](#)

[Teams](#)

[C interface](#)

[Data Structures](#)

[Split Team Operations](#)

[C Interface](#)

[Team query functions](#)

[C Interface](#)

[Endpoint](#)

[C Interface](#)

[Starting and Completing the Collectives](#)

[C Interface](#)

[Data Structures](#)

Library Handles and Structures

Name	
Library handle	<i>ucc_lib_h</i>
Library Parameters	<i>ucc_lib_params_t</i>

Library attributes	<i>ucc_lib_attrbs_t</i>
Team handle	<i>ucc_team_h</i>
Context handle	<i>ucc_context_h</i>
Context param structure	<i>ucc_context_params_t</i>
Team param structure	<i>ucc_team_params_t</i>
Team attribute structure	<i>ucc_team_attrbs_t</i>
Collective synchronization (enum)	<i>ucc_coll_sync_type_t</i>
OOB Collectives signature	<i>ucc_oob_context_t</i>
OOB Collectives signature	<i>ucc_oob_team_t</i>
Datatype (enum)	<i>ucc_datatype_t</i>
Collective operations info structure	<i>ucc_coll_op_args_t</i>
Request handle	<i>ucc_coll_req_h</i>
Collective type (enum)	<i>ucc_coll_type_t</i>
Reduction operation (enum)	<i>ucc_reduction_op_t</i>
Collective buffer info structure	<i>ucc_coll_buffer_info_t</i>
Memory constraints (enum)	<i>ucc_mem_constraints_t</i>
Memory hints (enum)	<i>ucc_mem_hints_t</i>
Collective tag id	<i>ucc_coll_tag_t</i>

Library Initialization and Finalization

These routines are responsible for allocating, initializing, and finalizing the resources for the library.

The UCC can be configured in two thread modes `UCC_LIB_THREAD_SINGLE` and `UCC_LIB_THREAD_MULTIPLE`. In the `UCC_LIB_THREAD_SINGLE` mode, the user program

must not be multithreaded. In the UCC_LIB_THREAD_MULTIPLE mode, the user program can be multithreaded and any thread may invoke the UCC operations.

The user can request different types of collective operations that vary in their synchronization models. The valid synchronization models are UCC_NO_SYNC_COLLECTIVES and UCC_SYNC_COLLECTIVES. The details of these synchronization models are described in the collective operation section.

The user can request the different collective operations and reduction operations required. The complete set of valid collective operations and reduction types are defined with the structures ucc_coll_type_t and ucc_reduction_op_t.

C Interface

```

/**
 * @ingroup UCC_LIB
 *
 * @brief A local operation to initialize and allocate the resources for the UCC
 * operations. The parameters passed using the ucc_lib_params_t and
 * ucc_lib_config structures will customize and select the functionality of the UCC
 * library. The library can be customized for its interaction with the user threads,
 * types of collective operations, and reductions supported.
 *
 * On success, the library object will be created and ucc_status_t will return
 * UCC_OK. On error, the library object will not be created and corresponding
 * error code as defined by ucc_status_t is returned.
 *
 * @param [in] params user provided parameters to customize the library
 * functionality
 * @param [in] config UCC configuration descriptor allocated through
 * @ref ucc_config_read "ucc_config_read()" routine.
 * @param [out] lib (UCC library handle)
 */
ucc_status_t ucc_init(
    const ucc_lib_params_t *params, const ucc_lib_config_t *config,
    ucc_lib_h *lib_p);

```

```
/**
 * @ingroup UCC_LIB
 * @brief @ref ucc_finalize is a local operation to release the resources and
 * cleanup. All participants that invoked @ref ucc_init should call this routine.
 *
 * @param [in] lib_p Handle to @ref ucc_lib_h
 *          "UCC library".
 */
ucc_status_t ucc_finalize( ucc_lib_h lib_p);

/* @brief A query operation to get the attributes of the library object. The
 * attributes are library configured values and reflect the choices made by the
 * library implementation.
 *
 * /@param [out] lib_attr - Library attributes
 * /@param [in] ucc_lib - Input library object
 */
ucc_status_t ucc_lib_get_attrbs(ucc_lib_h lib_p, ucc_lib_attr_t *lib_attr);

/*
 * @brief @ref ucc_config_read allocates the @ref ucc_lib_config_t structure
 * and fetches the configuration values from the run-time environment. The run
 * environment supported are environment variables or a configuration file.
 *
 * /@param [out] config_p - Pointer to configuration descriptor as defined by
 * @ref ucc_lib_config_t.
 *
 * /@param [out] env_prefix - If not NULL, the routine searches for the
 * environment variables with the prefix UCC_<env_prefix>. Otherwise, the
 * routines search for the environment variables that start with the prefix @ UCC_.
 *
 * /@param [in] filename - If not NULL, read configuration values from the file
 * defined by @e filename. If the file does not exist, it will be ignored and no error
 * will be reported to the user.
 */
```

```
ucc_status_t ucc_lib_config_read(const char *env_prefix, const char *filename,
ucc_lib_config_h **config_p);

/*
 * /@param [in] config_p - Pointer to the configuration descriptor to be
released.
 *
 */
void ucc_lib_config_release(ucc_lib_config_h *config);

/*
 * /@param [in] config - Configuration to be modified by the routine
 * /@param [in] name - Configuration variable to be modified
 * /@param [in] value - Configuration value to set
 */
ucc_status_t ucc_lib_config_modify(ucc_lib_config_h *config, const char
*name, const char *value);
```

Data Structures

ucc_lib_params_t: The UCC library functionality is customized using the structure `ucc_lib_params_t` which has fields `mask`, `ucc_thread_mode_t`, `ucc_reduction_op_t`, and `ucc_coll_sync_t`.

The bitwise mask represents the set of parameters valid for the `ucc_lib_params_t`. The UCC can be configured in two thread modes `UCC_LIB_THREAD_SINGLE` and `UCC_LIB_THREAD_MULTIPLE` using `ucc_thread_mode_t` field. The user can configure different valid synchronization models such as `UCC_NO_SYNC_COLLECTIVES` and `UCC_SYNC_COLLECTIVES` using the `ucc_coll_sync_t` field. The user can request different collective operations and reduction operations using fields `ucc_coll_type_t` and `ucc_reduction_op_t`, respectively.

```

typedef struct ucc_lib_params {
    uint64_t          mask;
    ucc_thread_mode_t thread_mode;
    ucc_coll_type_t   coll_types;
    ucc_reduction_op_t reduction_op_types;
    ucc_coll_sync_t   sync_type;
} ucc_lib_params_t;

typedef struct ucc_lib_attrbs {
    uint64_t          mask;
    ucc_thread_mode_t thread_mode;
    ucc_coll_type_t   provided_coll_types;
    ucc_reduction_op_t provided_reduction_types;
    ucc_coll_sync_t   sync_type;
} ucc_lib_attrbs_t;

typedef enum ucc_lib_params_mask {
    UCC_THREAD_MODE      = UCC_BIT(0),
    UCC_COLL_TYPES       = UCC_BIT(1),
    UCC_REDUCTION_TYPES  = UCC_BIT(3),
    UCC_SYNC_TYPE        = UCC_BIT(4)
} ucc_lib_params_mask_t;

```

Communication Context

The `ucc_context_h` is a communication context handle. It can encapsulate resources required for collective operations on team handles. The contexts are created by the `ucc_context_create` operation and destroyed by the `ucc_context_destroy` operation. The create operation takes in user-configured `ucc_context_params_t` structure to customize the context handle. The attributes of the context created can be queried using the `ucc_context_get_attrbs` operation.

When no out-of-band operation (OOB) is provided, the `ucc_context_create` operation is local requiring no communication with other participants. When OOB operation is provided, all participants of the OOB operation should participate in the create operation. If the context operation is a collective operation, the `ucc_context_destroy` operation is also a collective operation .i.e., all participants should call the destroy operation.

The context can be created as an exclusive type or shared type by passing constants `UCC_CONTEXT_EXCLUSIVE` and `UCC_CONTEXT_SHARED` respectively to the

ucc_context_params_t structure. When context is created as a shared type, the same context handle can be used to create multiple teams. When context is created as an exclusive type, the context can be used to create multiple teams but the team handles cannot be valid at the same time; a valid team is defined as a team object where the user can post collective operations.

Notes : From the user perspective, the context handle represents a communication resource. The user can create one context and use it for multiple teams or use with a single team. This provides a finer control of resources for the user. From the library implementation perspective, the context could represent the network parallelism. The UCC library implementation can choose to abstract injection queues, network endpoints, GPU device context, UCP worker, or UCP endpoints using the communication context handles.

C Interface

```

/*
 * @brief The ucc_context_create creates the context and ucc_context_destroy
 releases the resources and destroys the context state. The creation of context
 does not necessarily indicate its readiness to be used for collective or other
 group operations.

 On success, the context handle will be created and ucc_status_t will return
 UCC_OK. On error, the library object will not be created and corresponding
 error code as defined by ucc_status_t is returned.
 *
 * /@param [in]  lib_context - Library handle
 * /@param [out] params - Customizations for the communication context
 * /@param [out] config - Configuration for the communication context to read
 from environment
 * /@param [out] context - Newly created communication context
 */
ucc_status_t ucc_context_create(
    ucc_lib_h lib_handle,
    const ucc_context_params_t *params,
    const ucc_context_config_t *config,
    ucc_context_h *context);

/* @brief @ref ucc_context_destroy routine releases the resources associated
 with the handle @e context. All teams associated with the team should be

```

released before this. It is invalid to associate any team with this handle after the routine is called.

* /@param [in] context - Communication context to be released

*/

```
ucc_status_t ucc_context_destroy(
    ucc_context_h context);
```

/*

* @brief @ref ucc_context_progress routine progresses the operations on the content handle. It does not block for lack of resources or communication.

* /@param [in] context - Communication context to be progressed

*/

```
ucc_status_t ucc_context_progress(ucc_context_h context);
```

/*

* @brief

@ref ucc_context_get_attris routine queries the context handle attributes. The attributes of the context handle are described by the context attributes @ref ucc_context_attr_t

* /@param [in] context - Communication context

* /@param [out] context_attr - Attributes of the communication context

*/

```
ucc_status_t ucc_context_get_attris(ucc_context_h context, ucc_context_attr_t
    *context_attr);
```

Data Structures

The structure `ucc_context_params_t` is used to customize the functionality of the communication context handle. The context can be created as an exclusive type or shared type by passing constant `UCC_CONTEXT_EXCLUSIVE` or `UCC_CONTEXT_SHARED` respectively to `ucc_context_type_t`. The context can be created for synchronous collectives or non synchronous collectives providing constant `UCC_SYNC_COLLECTIVES` and `UCC_NO_SYNC_COLLECTIVES` to `ucc_coll_sync_type_t`. `oob_func` is passed for creating context as a collective operation to `ucc_context_oob_t`.


```
typedef enum {
    UCC_NO_SYNC_COLLECTIVES = 0,
    UCC_SYNC_COLLECTIVES = 1
} ucc_coll_sync_type_t;

typedef enum {
    UCC_CONTEXT_EXCLUSIVE = 0,
    UCC_CONTEXT_SHARED
} ucc_context_type_t;

enum ucc_context_attrbs_field {
    UCC_CONTEXT_TYPE = UCC_BIT(0),
    UCC_COLL_SYNC_TYPE = UCC_BIT(1),
    UCC_COLL_OOB = UCC_BIT(2)
};

enum ucc_context_params_field {
    UCC_CONTEXT_TYPE = UCC_BIT(0),
    UCC_COLL_SYNC_TYPE = UCC_BIT(1),
    UCC_COLL_OOB = UCC_BIT(2)
};

typedef struct ucc_context_params {
    uint64_t mask;
    ucc_context_type_t ctx_type;
    ucc_coll_sync_type_t sync_type;
    ucc_context_oob_t oob_func;
} ucc_context_params_t;

typedef struct ucc_context_attrbs {
    uint64_t mask;
    ucc_context_type_t ctx_type;
    ucc_coll_sync_type_t sync_type;
} ucc_context_attrbs_t;
```

Teams

The `ucc_team_h` is a team handle, which encapsulates the resources required for group operations such as collective communication operations. The participants of the group operations can either be an OS process, a control thread or a task.

Create and destroy routines: `ucc_team_create_post` routine is used to create the team handle and `ucc_team_create_test` routine for learning the status of the create operation. The team handle is destroyed by the `ucc_team_destroy` operation. A team handle is customized using the user configured `ucc_team_params_t` structure.

Invocation semantics: The `ucc_team_create_post` is a nonblocking collective operation, in which the participants are determined by the user-provided OOB collective operation. Overlapping of multiple `ucc_team_create_post` operations are invalid. Posting a collective operation before the team handle is created is invalid. The team handle is destroyed by a blocking collective operation; the participants of this collective operation are the same as the create operation. When the user does not provide an OOB collective operation, all participants calling the `ucc_create_post` operation will be part of a new team created.

Communication Contexts: Each process or a thread participating in the team creation operation contributes one or more communication contexts to the operation. The number of contexts provided by all participants should be the same and each participant should provide the same type of context. The newly created team uses the context for collective operations. If the communication context abstracts the resources for the library, the collective operations on this team uses the resources provided by the context.

Endpoints: That participants to the `ucc_team_create_post` operation can provide an endpoint, a 64-bit unsigned integer. The endpoint is an address for communication. Each participant of the team has a unique integer as endpoint .i.e., the participants of the team do not share the same endpoint. The user can bind the endpoint to the programming model's index such as MPI rank or OpenSHMEM PE, an OS process ID, or a thread ID. The UCC implementation can use the endpoint as an index to identify the resources required for communication such as communication contexts. When the user does not provide the endpoint, the library generates the endpoint, which can be queried by the user. In addition to the endpoint, the user can provide information about the endpoints such as whether the endpoint is a continuous range or not.

Ordering: The collective operations on the team can either be ordered or unordered. In the ordered model, the UCC collectives follow the MPI ordering model .i.e., on a given team, each of the participants of the collective operation invokes the operation in the same order. In the unordered model, the collective operations are not necessarily invoked in the same order.

Interaction with Threads: The team can be created in either mode .i.e., the library initialized by UCC_LIB_THREAD_MULTIPLE or UCC_LIB_THREAD_SINGLE. In the UCC_LIB_THREAD_MULTIPLE mode, each of the user thread can post a collective operation. However, it is not valid to post concurrent collectives operations from multiple threads to the same team.

Memory per Team: A team can be configured by a memory descriptor described by ucc_mem_map_params_t structure. The memory can be used as an input and output buffers for the collective operation. This is particularly useful for PGAS programming models, where the input and output buffers are defined before the invocation operation. For example, the input and output buffers in the OpenSHMEM programming model are defined during the programming model initialization.

Synchronization Model: The team can be configured to support either synchronized collectives or non-synchronized collectives. If the UCC library is configured with synchronized collective operations and the team is configured with non-synchronized collective operations, the library might not be able to provide any optimizations and might support only synchronized collective operations.

Outstanding Calls: The user can configure maximum number of outstanding collective operations of any type for a given team. This is represented by an unsigned integer. This is provided as a hint to the library for resource management.

C interface

```
/*
 * @brief ucc_team_create_post is a nonblocking collective operation to create the
 * team handle. It takes in parameters ucc_context_h, num_handles, ucc_team_params_t
 * and returns a ucc_team_handle_h. The ucc_team_params_t provides user
 * configuration to customize the team. The routine returns immediately after posting the
 * operation with the new team handle. However, the team handle is not ready for posting
 * the collective operation. ucc_team_create_test operation is used to learn the status of
 * the new team handle. On error, the team handle will not be created and corresponding
 * error code as defined by ucc_status_t is returned.
 *
 * @param [in] contexts - Communication context abstracting the resources
 * @param [in] num_contexts - Number of context provided as input
 * @param [in] params - User defined configurations for the team
 * @param [out] ucc_team - Team handle created
```

```

*/

ucc_status_t ucc_team_create_post(
    ucc_context_h *contexts,
    uint32_t      num_contexts,
    ucc_team_params_t team_params,
    ucc_team_h *new_team);

/*
 * @brief @ref ucc_team_create_test routines tests the status of team handle. If
 * required it can progress the communication but cannot block on the communications.
 * @param [in] ucc_team - Team handle to test
 */

ucc_status_t ucc_team_create_test(ucc_team_h team);
/*
 * @brief ucc_team_destroy is a blocking collective operation to release all resources
 * associated with the team handle, and destroy the team handle. It is invalid to post a
 * collective operation after the ucc_team_destroy operation.
 *
 * @param [in] team - Destroy previously created team and release all resources
 * associated with it.
 */
ucc_status_t ucc_team_destroy(
    ucc_team_h team
);

```

Data Structures

The structure `ucc_team_params_t` is used to customize the functionality of the team handle. The team can be created as accessible by multiple threads by passing constant `UCC_TEAM_THREAD_MULTIPLE` or `UCC_TEAM_THREAD_SHARED` respectively to `ucc_team_thread_type_t`. The team can be created for synchronous collectives or non synchronous collectives providing constant `UCC_SYNC_COLLECTIVES` and `UCC_NO_SYNC_COLLECTIVES` to `ucc_coll_sync_type_t`. `oob_func` is passed to `ucc_team_oob_t` passed for coordinating the participants. The endpoint of the participant is provided as input by the user.

```
typedef struct ucc_team_params {
    uint64_t      mask;
    ucc_post_ordering_t  ordering;
    uint64_t      outstanding_colls;
    uint64_t      ep;
    ucc_ep_range_type_t    ep_range;
    ucc_coll_sync_type_t  sync_type;
    ucc_oob_team_coll_t  oob_collectives;
    ucc_mem_map_params_t  mem_params;
} ucc_team_params_t;

typedef struct ucc_team_attr {
    uint64_t      mask;
    ucc_post_ordering_t  ordering;
    uint64_t      outstanding_colls;
    uint64_t      ep;
    ucc_ep_type_t    ep_range;
    ucc_coll_sync_type_t  sync_type;
    ucc_mem_map_params_t  mem_params;
} ucc_team_attr_t;

typedef struct ucc_mem_map_params {
    void *address;
    size_t len;
    ucc_mem_hints_t hints;
    ucc_mem_constraints_t constraints;
} ucc_mem_map_params_t;

typedef enum {
    UCC_COLLECTIVE_POST_ORDERED = 0,
    UCC_COLLECTIVE_POST_UNORDERED = 1
} ucc_post_ordering_t;

typedef enum {
    UCC_MEM_SYMMETRIC = UCC_BIT(0),
    UCC_MEM_PERSISTENT = UCC_BIT(1),
    UCC_MEM_ALIGN32 = UCC_BIT(2),
    UCC_MEM_ALIGN64 = UCC_BIT(3),
    UCC_MEM_ALIGN128 = UCC_BIT(4),
} ucc_mem_constraints_t;
```

```
typedef enum {  
    REMOTE_ATOMICS,  
    REMOTE_COUNTERS  
} ucc_mem_hints_t;
```

Split Team Operations

The team split routines provide an alternate way to create teams. All split routines require a parent team and all participants of the parent team call the split operation. The participants of the new team may include some or all participants of the parent team.

The newly created team shares the communication contexts with the parent team. The endpoint of the new team is contiguous and is not related to the parent team. It inherits the thread model, synchronization model, collective ordering model, outstanding collectives configuration, and memory descriptor from the parent team.

The split operation can be called by multiple threads, if the parent team to the split operations are different and if it agrees with the thread model of the UCC library.

Notes: The rationale behind requiring all participants of the parent team to participate in the split operation is to avoid overlapping participants between multiple split operations. Also, the MPI and OpenSHMEM programming models impose this constraint.

C Interface

```
/ @brief ucc_team_create_from_parent is a nonblocking collective operation,  
which creates a new team from the parent team. If a participant intends to  
participate in the new team, it passes a TRUE value for the "included" parameter.  
Otherwise, it passes FALSE. The routine returns immediately after the  
post-operation. To learn the completion of the team create operation, the  
ucc_team_create_test operation is used.
```

```
/ @param [in] my_ep - Endpoint of the process/thread calling the split operation
```

```

/@param [in]  parent_team - Parent team handle from which a new team handle is
created
/@param [in]  color - Boolean variable indicating whether the process/thread
participates in the newly created team
/@parm [out]  new_ucc_team - Pointer to the new team handle

ucc_status_t ucc_team_create_from_parent(
    uint64_t my_ep,
    bool included,
    ucc_team_h parent_team,
    ucc_team_h *new_ucc_team);

```

Team query functions

A set of team query operations.

C Interface

```

/* @ref ucc_team_get_attribs routine queries the team handle attributes. The
attributes of the team handle are described by the team attributes structure @ref
ucc_team_attr_t. The attributes should reflect the options selected by the library.
*
* /@param [out] team_attr - Team attributes
* /@param [in]  ucc_team - Team handle
*/

ucc_status_t ucc_team_get_attribs(ucc_team_h ucc_team, ucc_team_attr_t
*team_attr)

/* @ref ucc_team_get_size routine queries the size of the team. It reflects the
number of unique endpoints in the team.
* /@param [in]  ucc_team - Team handle
* /@param [out] size      - The size of team as number of endpoints
*/

```

```
ucc_status_t ucc_team_get_size(ucc_team_h ucc_team, uint32_t *size);

/* @ref ucc_team_my_ep routine queries and returns the endpoint of the
participant invoking the interface.
* /@param [out] ep - Endpoint of the participant calling the routine
* /@param [in] ucc_team - Team handle
*/

ucc_status_t ucc_team_get_my_ep(ucc_team_h ucc_team, uint64_t *ep);

/*
* @ref ucc_team_my_ep routine queries and returns all endpoints of all
participants in the team.
* /@param [out] ep - List of endpoints
* /@param [out] num_eps - Number of endpoints
* /@param [in] ucc_team - Team handle
*/

ucc_status_t ucc_team_get_all_eps(ucc_team_h ucc_team, uint64_t **ep, uint64_t
*num_eps);
```

Endpoint

C Interface

```
/*
* /@param [in] ep - List of Team endpoints
* /@param [in] num_eps - Number of endpoints
* /@param [in] ucc_team - parent Team
* /@param [out] ucc_team - New Team
*/

ucc_status_t ucc_create_team_from_eps(ucc_team_h parent_ucc_team, uint64_t **ep,
uint64_t num_eps, ucc_team_h *new_team);
```


Starting and Completing the Collectives

A UCC collective operation is a group communication operation among the participants of the team. All participants of the team are required to call the collective operation. Each participant is represented by the endpoint that is unique to the team used for the collective operation. This section provides a set of routines for launching, progressing, and completing the collective operations.

Invocation semantics: The `ucc_collective_init` routine is a non-blocking collective operation to initialize the buffers, operation type, reduction type, and other information required for the collective operation. All participants of the team should call the initialize operation. The routine returns once the participants enter the collective initialize operation. The collective operation is invoked using a `ucc_collective_post` operation. `ucc_collective_init_and_post` operation initializes as well as post the collective operation.

Collective type: The collective operation supported by UCC is defined by the enumeration `ucc_coll_type_t`. It supports three types of collective operations: (a) `UCC_{ALLTOALL, ALLGATHER, ALLREDUCE}` operations where all participants contribute to the results and receive the results (b) `UCC_{REDUCE, GATHER, FANIN}` where all participants contribute to the result and one participant receives the result. The participant receiving the result is designated as root. (c) `UCC_{BROADCAST, MULTICAST, SCATTER, FANOUT}` where one participant contributes to the result, and all participants receive the result. The participant contributing to the result is designated as root.

Reduction types: The reduction operation supported by `UCC_{ALLREDUCE, REDUCE}` operation is defined by the enumeration `ucc_op_t`. The valid datatypes for the reduction is defined by the enumeration `ucc_datatype_t`.

Ordering: The team can be configured for ordered collective operations or unordered collective operations. For unordered collectives, the user is required to provide the “tag”, which is an unsigned 64-bit integer.

Synchronized and Non-Synchronized Collectives: In the synchronized collective model, on entry, the participants cannot read or write to other participants without ensuring all participants have entered the collective operation. On the exit of the collective operation, the participants may exit after all participants have completed the reading or writing to the buffers.

In the non-synchronized collective model, on entry, the participants can read or write to other participants. If the input and output buffers are defined on the team and RMA operations are used for data transfer, it is the responsibility of the user to ensure the readiness of the buffer. On exit, the participants may exit once the read and write to the local buffers are completed.

C Interface

```

/*
 * @brief @ref ucc_collective_init is a collective initialization operation, where
 all participants participate. The user provides all information required to start
 and complete the collective operation, which includes the input and output
 buffers, operation type, team handle, size, and any other hints for optimization.
 On success, the request handle is created and returned. On error, the request
 handle is not created and the appropriate error code is returned.
 On return, the ownership of buffers is transferred to the library and users
 should not modify it. If modified, the results of collective operations posted on
 the request handle are undefined.
 *
 * /@param [out] request - Request handle representing the collective operation
 * /@param [in] coll_args - Collective arguments descriptor
 * /@param [in] ucc_team - Team handle
 */

ucc_status_t ucc_collective_init(
    ucc_coll_op_args_t *coll_args,
    ucc_coll_req_h *request,
    ucc_team_h team);

/*
 * @brief @ref ucc_collective_post routine posts the collective operation. It does
 not require synchronization between the participants for the post operation
 * /@param [in] request - Request handle
 */
ucc_status_t ucc_collective_post(ucc_coll_req_h request)

/*
 * @brief @ref ucc_collective_init_and_post initializes the collective operation
 and also posts the operation.

Note: The @ref ucc_collective_init_and_post can be implemented as a
combination of @ref ucc_collective_init and @ref ucc_collective_post routines.
 *

```

```

* /@param [out] request - Request handle representing the collective operation
* /@param [in] coll_args - Collective arguments descriptor
* /@param [in] ucc_team - Input Team
*/

ucc_status_t ucc_collective_init_and_post(
    ucc_coll_op_args_t *coll_args,
    ucc_coll_req_h *request,
    ucc_team_h team);

/*
* @brief @ucc_collective_test tests and returns the status of collective
operation.
* /@param [in] request - Request handle
*/
ucc_status_t ucc_collective_test(ucc_coll_req_h request);

/* @brief @ref ucc_collective_finalize operation releases all resources
associated with the collective operation represented by the request handle.
* /@param [in] request - request handle
*/
ucc_status_t ucc_collective_finalize(ucc_coll_req_h request);

```

Data Structures

```

typedef struct ucc_coll_buffer_info {
    uint64_t    mask;
    void        *src_buffer;
    uint32_t    *scounts;
    uint32_t    *src_displacements;
    void        *dst_buffer;
    uint32_t    *dst_counts;
    uint32_t    *dst_displacements;
}

```

```

    size_t      size;
    ucc_datatype_t src_datatype;
    ucc_datatype_t dst_datatype;
    uint64      flags, /* in-buffer, persistent , symmetric, ready before invocation */
} ucc_coll_buffer_info_t;

```

```

typedef enum {
    UCC_OP_MAX = UCC_BIT(0),
    UCC_OP_MIN = UCC_BIT(1),
    UCC_OP_SUM = UCC_BIT(2),
    UCC_OP_PROD = UCC_BIT(3),
    UCC_OP_AND = UCC_BIT(4),
    UCC_OP_OR = UCC_BIT(5),
    UCC_OP_XOR = UCC_BIT(6),
    UCC_OP_LAND = UCC_BIT(7),
    UCC_OP_LOR = UCC_BIT(8),
    UCC_OP_LXOR = UCC_BIT(9),
    UCC_OP_BAND = UCC_BIT(10),
    UCC_OP_BOR = UCC_BIT(11),
    UCC_OP_BXOR = UCC_BIT(12),
    UCC_OP_MAXLOC = UCC_BIT(13),
    UCC_OP_MINLOC = UCC_BIT(14),
    UCC_OP_LAST_PREDEFINED = UCC_BIT(15),
    UCC_OP_UNSUPPORTED = UCC_BIT(16)
} ucc_reduction_op_t;

```

```

typedef enum {
    UCC_DT_INT8 = 0,
    UCC_DT_INT16,
    UCC_DT_INT32,
    UCC_DT_INT64,
    UCC_DT_INT128,
    UCC_DT_UINT8,
    UCC_DT_UINT16,
    UCC_DT_UINT32,
    UCC_DT_UINT64,
    UCC_DT_UINT128,
    UCC_DT_FLOAT16,
    UCC_DT_FLOAT32,
    UCC_DT_FLOAT64,
    UCC_DT_LAST_PREDEFINED,
    UCC_DT_UNSUPPORTED
}

```

```
} ucc_datatype_t;

typedef enum {
    UCC_BARRIER = UCC_BIT(0),
    UCC_BCAST = UCC_BIT(1),
    UCC_ALLREDUCE = UCC_BIT(2),
    UCC_REDUCE = UCC_BIT(3),
    UCC_ALLTOALL = UCC_BIT(4),
    UCC_ALLGATHER = UCC_BIT(5),
    UCC_GATHER = UCC_BIT(6),
    UCC_SCATTER = UCC_BIT(7),
    UCC_MCAST = UCC_BIT(8),
    UCC_FANIN = UCC_BIT(9),
    UCC_FANOUT = UCC_BIT(10),
    UCC_COLL_LAST = UCC_BIT(11)
} ucc_coll_type_t;

typedef struct ucc_reduction_info {
    ucc_datatype_t dt;
    ucc_reduction_op_t op;
    size_t count;
} ucc_reduction_info_t;

typedef enum {
    UCC_ERR_TYPE_LOCAL=0,
    UCC_ERR_TYPE_GLOBAL=1
} ucc_error_type_t;

typedef uint16_t ucc_coll_id_t;

typedef struct ucc_coll_op_args {
    uint64_t mask;
    ucc_coll_type_t coll_type;
    ucc_coll_ext_op_args_t ext_args;
} ucc_coll_op_args_t;

typedef struct ucc_coll_ext_op_args {
    ucc_coll_buffer_info_t buffer_info;
    ucc_reduction_info_t reduction_info;
    ucc_error_type_t error_type;
    ucc_coll_id_t tag;
```

```
uint64_t      root;  
} ucc_coll_ext_op_args_t;
```