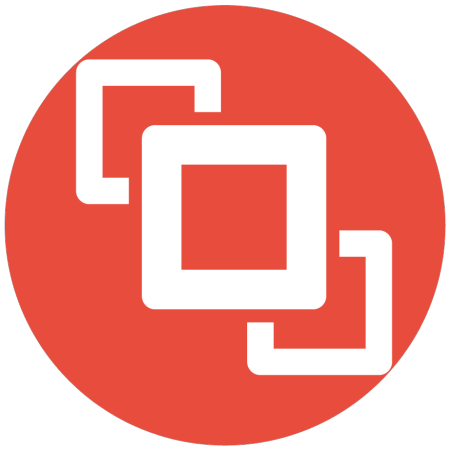
Unified Communication X (UCX)

API Standard

Version 1.0a (DRAFT)



Contents

Chapter 1

Preface

1.1 Scope of the Document

This document describes the UCX programming interface. The programming interface exposes a high performance communication API, which provides basic building blocks for PGAS, Message Passing Interface (MPI), Big-Data, Analytics, File I/O, and storage library developers.

1.2 Audience

This manual is intended for programmers who want to develop parallel programming models like OpenSHMEM, MPI, UPC, Chapel, etc. The manual assumes that the reader is familiar with the following:

- Basic concepts of two-sided, one-sided, atomic, and collective operations
- C programming language

1.3 Document Status

This section briefly describes a list of open issues in the UCX specification.

- UCP API - work in progress
- UCT API - work in progress

1.4 License

UCX project follows open source development model and the software is licensed under BSD-3 license.

Chapter 2

Introduction

2.1 Motivation

A communication middleware abstracts the vendor-specific software and hardware interfaces. They bridge the semantic and functionality gap between the programming models and the software and hardware network interfaces by providing data transfer interfaces and implementation, optimized protocols for data transfer between various memories, and managing network resources. There are many communication middleware APIs and libraries to support parallel programming models such as MPI, OpenSHMEM, and task-based models.

Current communication middleware designs typically take two approaches. First, communication middleware such as Intel's PSM (previously Qlogic), Mellanox's MXM, and IBM's PAMI provide high-performance implementations for specific network hardware. Second, communication middleware such as VMI, Cactus, ARMCI, GASNet, and Open MPI are tightly coupled to a specific programming model. Communication middleware designed with either of this design approach requires significant porting effort to move a new network interface or programming model.

To achieve functional and performance portability across architectures and programming models, we introduce Unified Communication X (UCX).

2.2 UCX

Unified Communication X (UCX) is a set of network APIs and their implementations for high throughput computing. UCX is a combined effort of national laboratories, industry, and academia to design and implement a high-performing and highly-scalable network stack for next generation applications and systems. UCX design provides the ability to tailor its APIs and network functionality to suit a wide variety of application domains. We envision that these APIs will satisfy the networking needs of many programming models such as the Message Passing Interface (MPI), OpenSHMEM, Partitioned Global Address Space (PGAS) languages, task-based paradigms, and I/O bound applications.

The initial focus is on supporting semantics such as point-to-point communications (one-sided and two-sided), collective communication, and remote atomic operations required for popular parallel programming models. Also, the initial UCX reference implementation is targeted to support current network technologies such as:

- Open Fabrics - InfiniBand (Mellanox, Qlogic, IBM), libfabrics, iWARP, RoCE
- Cray GEMINI & ARIES
- Shared memory (MMAP, Posix, CMA, KNEM, XPMEM, etc.)
- Ethernet (TCP/UDP)

UCX design goals are focused on performance and scalability, while efficiently supporting popular and emerging programming models.

UCX's API and design do not impose architectural constraints on the network hardware nor require any specific capabilities to the support the programming model functionality. This is achieved by keeping the API flexible and ability to support the missing functionality efficiently in the software.

Extreme scalability is an important design goal for UCX. To achieve this, UCX follows these design principles :

- Minimal memory consumption : Design avoids data-structures that scale with the number of processing elements (i.e., order N data structures), and share resources among multiple programming models.
- Low-latency Interfaces: Design provides at least two sets of APIs with one set focused on the performance, and the other focused on functionality.
- High bandwidth - With minimal software overhead combined and support for multi-rail and multi-device capabilities, the design provides all the hooks that are necessary for exploiting hardware bandwidth capabilities.
- Asynchronous Progress: API provides non-blocking communication interfaces and design supports asynchronous progress required for communication and computation overlap
- Resilience - the API exposes communication control hooks required for fault tolerant communication library implementation.

UCX design provides native support for hybrid programming models. The design enables resource sharing, optimal memory usage, and progress engine coordination to efficiently implement hybrid programming models. For example, hybrid applications that use both OpenSHMEM and MPI programming models will be able to select between a single-shared UCX network context or a stand alone UCX network context for each one of them. Such flexibility, optimized resource sharing, and reduced memory consumption, improve network and application performance.

Chapter 3

Design

The UCX framework consists of the three main components: UC-Services (UCS), UC-Transports (UCT), and UC-Protocols (UCP). Each one of these components exports a public API, and can be used as a stand-alone library.

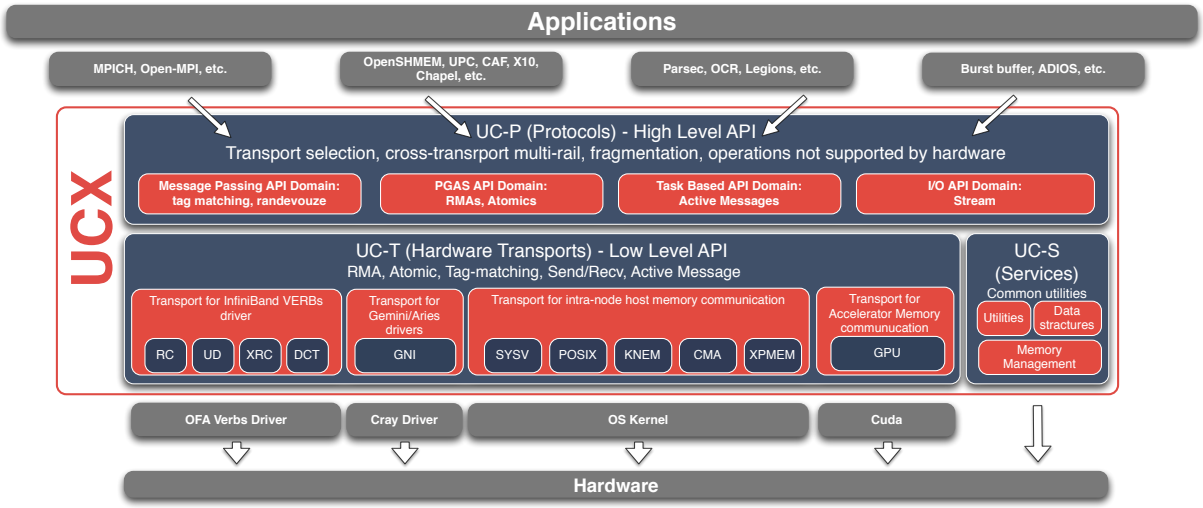


Figure 3.1: UCX Framework Architecture

3.1 UCS

UCS is a service layer that provides the necessary functionality for implementing portable and efficient utilities. This layer includes the following services:

- an abstraction for accessing platform specific functionality (atomic operations, thread safety, etc.),
- tools for efficient memory management (memory pools, memory allocators, and memory allocators hooks),
- commonly used data structures (hashes, trees, lists).

3.2 UCT

UCT is a transport layer that abstracts the differences across various hardware architectures and provides a low-level API that enables the implementation of communication protocols. The primary goal of the layer is to provide direct and efficient access to hardware network functionality. For this purpose, UCT relies on vendor provided

low-level drivers such as InfiniBand Verbs, Cray's uGNI, libfabrics, etc. In addition, the layer provides constructs for communication context management (thread-based and application level), and allocation and management of device-specific memories including those found in accelerators. In terms of communication APIs, UCT defines interfaces for immediate (short), buffered copy-and-send (bcopy), and zero-copy (zcopy) communication operations. The short operations are optimized for small messages that can be posted and completed in place. The bcopy operations are optimized for medium size messages that are typically sent through a so-called bouncing-buffer. Finally, the zcopy operations expose zero-copy memory-to-memory communication semantics.

3.3 UCP

UCP implements higher-level protocols that are typically used by message passing (MPI) and PGAS programming models by using lower-level capabilities exposed through the UCT layer. UCP is provides the following functionality: ability to select different transports for communication, message fragmentation, multi-rail communication, and initializing and finalizing the library. Currently, the API has the following classes of interfaces: Initialization, Remote Memory Access (RMA) communication, Atomic Memory Operations (AMO), Active Message, Tag-Matching, and Collectives.

Initialization: This subset of interfaces defines the communication context setup, queries the network capabilities, and initializes the local communication endpoints. The context represented by the UCX context is an abstraction of the network transport resources. The communication endpoint setup interfaces initialize the UCP endpoint, which is an abstraction of all the necessary resources associated with a particular connection. The communication endpoints are used as input to all communication operations to describe the source and destination of the communication.

RMA: This subset of interfaces defines one-sided communication operations such as PUT and GET, required for implementing low overhead, direct memory access communications constructs needed by both distributed and shared memory programming models. UCP includes a separate set of interfaces for communicating non-contiguous data. This functionality was included to support various programming models' communication requirements and leverage the scatter/gather capabilities of modern network hardware.

AMO: This subset of interfaces provides support for atomically performing operations on the remote memory, an important class of operations for PGAS programming models, particularly OpenSHMEM.

Tag Matching: This interface supports tag-matching for send-receive semantics which is a key communication semantic defined by the MPI specification.

Active Message: A subset of functionality where the incoming packet invokes a sender-specified callback in order to be processed by the receiving process. As an example, the two-sided MPI interface can easily be implemented on top of such a concept (TBD: cite openmpi). However, these interfaces are more general and suited for other programming paradigms where the receiver process does not prepost receives, but expects to react to incoming packets directly. Like RMA and tag-matching interfaces, the active message interface provides separate APIs for different message types and non-contiguous data.

Collectives: This subset of interfaces defines group com- munication and synchronization operations. The collective operations include Barrier, All-to-one, All-to-all, and reduction operations. When possible, we will take advantage of hardware acceleration for collectives (e.g., InfiniBand Switch collective acceleration).

Chapter 4

Conventions and Notations

This section describes the conventions and notations in the UCX specification.

4.1 Blocking Behavior

The blocking UCX routines return only when an UCX operation is complete. After the return, the resources used in the UCX routine are available for reuse.

4.2 Non-blocking Behavior

The non-blocking UCX routines return immediately, independent of operation completion. After the return, the resources used for the routines are not necessarily available for reuse.

4.3 Fairness

UCX routines do not guarantee fairness. However, the routines enable UCX consumers to write efficient and fair programs.

Chapter 5

Module Documentation

5.1 Unified Communication Protocol (UCP) API

Modules

5.1.1 Detailed Description

This section describes UCP API.

5.2 UCP Application Context

Data Structures

- struct ucp_tag_recv_completion

  Completion status of a tag-matched receive. More...
- struct ucp_tag_recv_info

  UCP receive information descriptor. More...

Typedefs

- typedef struct ucp_tag_recv_completion ucp_tag_recv_completion_t

  Completion status of a tag-matched receive.
- typedef struct ucp_context * ucp_context_h

  UCP Application Context.
- typedef void(* ucp_request_init_callback_t) (void *request)

  Request initialization callback.
- typedef void(* ucp_request_cleanup_callback_t) (void *request)

  Request cleanup callback.

Enumerations

- enum ucp_feature { UCP_FEATURE_TAG = UCS_BIT(0), UCP_FEATURE_RMA = UCS_BIT(1), UCP_F↩ EATURE_AMO32 = UCS_BIT(2), UCP_FEATURE_AMO64 = UCS_BIT(3) }

  UCP configuration features.

Functions

- void ucp_get_version (unsigned *major_version, unsigned *minor_version, unsigned *release_number)

  Get UCP library version.
- const char * ucp_get_version_string ()

  Get UCP library version as a string.
- static ucs_status_t ucp_init (const ucp_params_t *params, const ucp_config_t *config, ucp_context_↩ h *context_p)

  UCP context initialization.
- void ucp_cleanup (ucp_context_h context_p)

  Release UCP application context.

5.2.1 Detailed Description

Application context is a primary concept of UCP design which provides an isolation mechanism, allowing resources associated with the context to separate or share network communication context across multiple instances of applications.

This section provides a detailed description of this concept and routines associated with it.

5.2.2 Data Structure Documentation

5.2.2.1 struct ucp_tag_recv_completion

Data Fields

ucp_tag_t	sender_tag	Full sender tag
size_t	rcvd_len	How much data was received

5.2.2.2 struct ucp_tag_recv_info

The UCP receive information descriptor is allocated by application and filled in with the information about the received message by ucp_tag_probe_nb routine.

Data Fields

ucp_tag_t	sender_tag	Sender tag
size_t	length	The size of the received data

5.2.3 Typedef Documentation

5.2.3.1 typedef struct ucp_tag_recv_completion ucp_tag_recv_completion_t

5.2.3.2 typedef struct ucp_context* ucp_context_h

UCP application context (or just a context) is an opaque handle that holds a UCP communication instance's global information. It represents a single UCP communication instance. The communication instance could be an OS process (an application) that uses UCP library. This global information includes communication resources, endpoints, memory, temporary file storage, and other communication information directly associated with a specific UCP instance. The context also acts as an isolation mechanism, allowing resources associated with the context to manage multiple concurrent communication instances. For example, users using both MPI and OpenSHMEM sessions simultaneously can isolate their communication by allocating and using separate contexts for each of them. Alternatively, users can share the communication resources (memory, network resource context, etc.) between them by using the same application context. A message sent or a RMA operation performed in one application context cannot be received in any other application context.

5.2.3.3 typedef void(* ucp_request_init_callback_t) (void *request)

This callback routine is responsible for the request initialization.

Parameters

in	*request*	Request handle to initialize.

5.2.3.4 typedef void(* ucp_request_cleanup_callback_t) (void *request)

This callback routine is responsible for cleanup of the memory associated with the request.

Parameters

in	*request*	Request handle to cleanup.

5.2.4 Enumeration Type Documentation

5.2.4.1 enum ucp_feature

The enumeration list describes the features supported by UCP. An application can request the features using UCP parameters during UCP initialization process.

Enumerator

UCP_FEATURE_TAG Request tag matching support

UCP_FEATURE_RMA Request remote memory access support

UCP_FEATURE_AMO32 Request 32-bit atomic operations support

UCP_FEATURE_AMO64 Request 64-bit atomic operations support

5.2.5 Function Documentation

5.2.5.1 void ucp_get_version (unsigned * *major_version*, unsigned * *minor_version*, unsigned * *release_number*)

This routine returns the UCP library version.

Parameters

out	*major_version*	Filled with library major version.
out	*minor_version*	Filled with library minor version.
out	*release_number*	Filled with library release number.

5.2.5.2 const char* ucp_get_version_string ( )

This routine returns the UCP library version as a string which consists of: "major.minor.release".

5.2.5.3 static ucs_status_t ucp_init (const ucp_params_t * *params*, const ucp_config_t * *config*, ucp_context_h * *context_p*) [inline], [static]

This routine creates and initializes a UCP application context.

Warning

This routine must be called before any other UCP function call in the application.

This routine checks API version compatibility, then discovers the available network interfaces, and initializes the network resources required for discovering of the network and memory related devices. This routine is responsible for initialization all information required for a particular application scope, for example, MPI application, OpenSH↩ MEM application, etc.

Note

- Higher level protocols can add additional communication isolation, as MPI does with it's communicator object. A single communication context may be used to support multiple MPI communicators.
- The context can be used to isolate the communication that corresponds to different protocols. For example, if MPI and OpenSHMEM are using UCP to isolate the MPI communication from the OpenSHMEM communication, users should use different application context for each of the communication libraries.

Parameters

in	*config*	UCP configuration descriptor allocated through ucp_config_read() routine.
in	*params*	User defined tunings for the UCP application context.
out	*context_p*	Initialized UCP application context.

Returns

Error code as defined by ucs_status_t

5.2.5.4 void ucp_cleanup (ucp_context_h *context_p*)

This routine finalizes and releases the resources associated with a UCP application context.

Warning

An application cannot call any UCP routine once the UCP application context released.

The cleanup process releases and shuts down all resources associated with the application context. After calling this routine, calling any UCP routine without calling UCP initialization routine is invalid.

Parameters

in	*context_p*	Handle to UCP application context.

5.3 UCP Worker

Typedefs

- typedef void ucp_address_t

  UCP worker address.
- typedef struct ucp_worker ∗ ucp_worker_h

  UCP Worker.

Functions

- ucs_status_t ucp_worker_create (ucp_context_h context, ucs_thread_mode_t thread_mode, ucp_worker_h ∗worker_p)

  Create a worker object.
- void ucp_worker_destroy (ucp_worker_h worker)

  Destroy a worker object.
- ucs_status_t ucp_worker_get_address (ucp_worker_h worker, ucp_address_t ∗∗address_p, size_t ∗address_length_p)

  Get the address of the worker object.
- void ucp_worker_release_address (ucp_worker_h worker, ucp_address_t ∗address)

  Release an address of the worker object.
- void ucp_worker_progress (ucp_worker_h worker)

  Progress all communications on a specific worker.
- ucs_status_t ucp_worker_fence (ucp_worker_h worker)

  Assures ordering between non-blocking operations.
- ucs_status_t ucp_worker_flush (ucp_worker_h worker)

  Flush all outstanding communication on the worker.

5.3.1 Detailed Description

UCP Worker routines

5.3.2 Typedef Documentation

5.3.2.1 typedef void ucp_address_t

The address handle is an opaque object that is used as an identifier for a worker instance.

5.3.2.2 typedef struct ucp_worker∗ ucp_worker_h

UCP worker is an opaque object representing the communication context. The worker represents an instance of a local communication resource and progress engine associated with it. Progress engine is a construct that is responsible for asynchronous and independent progress of communication directives. The progress engine could be implement in hardware or software. The worker object abstract an instance of network resources such as a host channel adapter port, network interface, or multiple resources such as multiple network interfaces or communication ports. It could also represent virtual communication resources that are defined across multiple devices. Although the worker can represent multiple network resources, it is associated with a single UCX application context. All communication functions require a context to perform the operation on the dedicated hardware resource(s) and an endpoint to address the destination.

Note

Worker are parallel "threading points" that an upper layer may use to optimize concurrent communications.

5.3.3 Function Documentation

5.3.3.1 ucs_status_t ucp_worker_create (ucp_context_h *context*, ucs_thread_mode_t *thread_mode*, ucp_worker_h * *worker_p*)

This routine allocates and initializes a worker object. Each worker is associated with one and only one application context. In the same time, an application context can create multiple workers in order to enable concurrent access to communication resources. For example, application can allocate a dedicated worker for each application thread, where every worker can be progressed independently of others.

Note

The worker object is allocated within context of the calling thread

Parameters

in	*context_p*	Handle to UCP application context.
in	*thread_mode*	Thread safety mode for the worker object and resources associated with it.
out	*worker_p*	A pointer to the worker object allocated by the UCP library

Returns

Error code as defined by ucs_status_t

5.3.3.2 void ucp_worker_destroy (ucp_worker_h *worker*)

This routine releases the resources associated with a UCP worker.

Warning

Once the UCP worker destroy the worker handle cannot be used with any UCP routine.

The destroy process releases and shuts down all resources associated with the worker.

Parameters

in	*worker*	Worker object to destroy.

5.3.3.3 ucs_status_t ucp_worker_get_address (ucp_worker_h *worker*, ucp_address_t ** *address_p*, size_t * *address_length_p*)

This routine returns the address of the worker object. This address can be passed to remote instances of the UCP library in order to to connect to this worker. The memory for the address handle is allocated by this function, and must be released by using ucp_worker_release_address() routine.

Parameters

in	*worker*	Worker object whose address to return.
out	*address_p*	A pointer to the worker address.
out	*address_length_p*	The size in bytes of the address.

Returns

Error code as defined by ucs_status_t

5.3.3.4 void ucp_worker_release_address (ucp_worker_h *worker,* ucp_address_t * *address*)

This routine release an address handle associated within the worker object.

Warning

Once the address released the address handle cannot be used with any UCP routine.

Parameters

in	*worker*	Worker object that is associated with the address object.
in	*address*	Address to release; the address object has to be allocated using ucp_worker_get_address() routine.

5.3.3.5 void ucp_worker_progress (ucp_worker_h *worker*)

This routine explicitly progresses all communication operations on a worker.

Note

- Typically, request wait and test routines call this routine to progress any outstanding operations.
- Transport layers, implementing asynchronous progress using threads, require callbacks and other user code to be thread safe.

Parameters

in	*worker*	Worker to progress.

5.3.3.6 ucs_status_t ucp_worker_fence (ucp_worker_h *worker*)

This routine ensures ordering of non-blocking communication operations on the UCP worker. Communication operations issued on the *worker* prior to this call are guaranteed to be completed before any subsequent communication operations to the same "worker" which follow the call to fence.

Note

The primary diference between ucp_worker_fence() and the ucp_worker_flush() is the fact the fence routine does not gurantee completion of the operations on the call return but only ensures the order between communication operations. The flush operation on return grantees that all operation are completed and corresponding memory regions were updated.

Parameters

in	*worker*	UCP worker.

Returns

Error code as defined by ucs_status_t

5.3.3.7 ucs_status_t ucp_worker_flush (ucp_worker_h *worker*)

This routine flushes all outstanding communication on the worker. Communication operations issued by on the *worker* prior to this call will have completed both at the origin and at the target worker when this call returns.

Note

For description of the differences between flush and fence operations please see ucp_worker_fence()

Parameters

in	*worker*	UCP worker.

Returns

Error code as defined by ucs_status_t

5.4 UCP Memory routines

Typedefs

- typedef struct ucp_rkey * ucp_rkey_h

  UCP Remote memory handle.
- typedef struct ucp_mem * ucp_mem_h

  UCP Memory handle.

Functions

- ucs_status_t ucp_mem_map (ucp_context_h context, void **address_p, size_t length, unsigned flags, ucp_mem_h *memh_p)

  Map or allocate memory for zero-copy operations.
- ucs_status_t ucp_mem_unmap (ucp_context_h context, ucp_mem_h memh)

  Unmap memory segment.
- ucs_status_t ucp_rkey_pack (ucp_context_h context, ucp_mem_h memh, void **rkey_buffer_p, size_t *size_p)

  Pack memory region remote access key.
- void ucp_rkey_buffer_release (void *rkey_buffer)

  Release packed remote key buffer.
- ucs_status_t ucp_ep_rkey_unpack (ucp_ep_h ep, void *rkey_buffer, ucp_rkey_h *rkey_p)

  Create remote access key from packed buffer.
- void ucp_rkey_destroy (ucp_rkey_h rkey)

  Destroy the remote key.
- ucs_status_t ucp_rmem_ptr (ucp_ep_h ep, void *remote_addr, ucp_rkey_h rkey, void **local_addr_p)

  If possible translate remote address into local address which can be accessed using direct load and store operations.

5.4.1 Detailed Description

UCP Memory routines

5.4.2 Typedef Documentation

5.4.2.1 typedef struct ucp_rkey* ucp_rkey_h

Remote memory handle is an opaque object representing remote memory access information. Typically, the handle includes a memory access key and other network hardware specific information, which are input to remote memory access operations, such as PUT, GET, and ATOMIC. The object is communicated to remote peers to enable an access to the memory region.

5.4.2.2 typedef struct ucp_mem* ucp_mem_h

Memory handle is an opaque object representing a memory region allocated through UCP library, which is optimized for remote memory access operations (zero-copy operations). The memory handle is a self-contained object, which includes the information required to access the memory region locally, while remote key is used to access it remotely. The memory could be registered to one or multiple network resources that are supported by UCP, such as InfiniBand, Gemini, and others.

5.4.3 Function Documentation

5.4.3.1 ucs_status_t ucp_mem_map (ucp_context_h *context*, void ** *address_p*, size_t *length*, unsigned *flags*, ucp_mem_h * *memh_p*)

This routine maps or/and allocates a user-specified memory segment with UCP application context and the network resources associated with it. If the application specifies NULL as an address for the memory segment, the routine allocates a mapped memory segment and returns its address in the *address_p* argument. The network stack associated with an application context can typically send and receive data from the mapped memory without CPU intervention; some devices and associated network stacks require the memory to be mapped to send and receive data. The memory handle includes all information required to access the memory locally using UCP routines, while remote registration handle provides an information that is necessary for remote memory access.

Note

Another well know terminology for the "map" operation that is typically used in the context of networking is memory "registration" or "pinning". The UCP library registers the memory the available hardware so it can be assessed directly by the hardware.

Memory mapping assumptions:

- A given memory segment can be mapped by several different communication stacks, if these are compatible.
- The *memh_p* handle returned may be used with any sub-region of the mapped memory.
- If a large segment is registered, and then segmented for subsequent use by a user, then the user is responsible for segmentation and subsequent management.

Parameters

in	*context*	Application context to map (register) and allocate the memory on.
in,out	*address_p*	If the pointer to the address is not NULL, the routine maps (registers) the memory segment. if the pointer is NULL, the library allocates mapped (registered) memory segment and returns its address in this argument.
in	*length*	Length (in bytes) to allocate or map (register).
in	*flags*	Allocation flags (currently reserved - set to 0).
out	*memh_p*	UCP handle for the allocated segment.

Returns

Error code as defined by ucs_status_t

5.4.3.2 ucs_status_t ucp_mem_unmap (ucp_context_h *context*, ucp_mem_h *memh*)

This routine unmaps a user specified memory segment, that was previously mapped using the ucp_mem_map() routine. The unmap routine will also release the resources associated with the memory "handle". When the function returns, the ucp_mem_h and associated remote key will be invalid and cannot be used with any UCP routine.

Note

Another well know terminology for the "unmap" operation that is typically used in the context of networking is memory "de-registration". The UCP library de-registers the memory the available hardware so it can be returned back to the operation system.

Error cases:

- Once memory is unmapped a network access to the region may cause a failure.

Parameters

in	*context*	Application context which was used to allocate/map the memory. [in] memh Handle to memory region.

Returns

Error code as defined by ucs_status_t

5.4.3.3 ucs_status_t ucp_rkey_pack (ucp_context_h *context*, ucp_mem_h *memh*, void ** *rkey_buffer_p*, size_t * *size_p*)

This routine allocates memory buffer and packs into the buffer a remote access key (RKEY) object. RKEY is an opaque object that provides the information that is necessary for remote memory access. This routine packs the RKEY object in a portable format such that the object can be "unpacked" on any platform supported by the UCP library. In order to release the memory buffer allocated by this routine the application is responsible to call the ucp_rkey_buffer_release() routine.

Note

- RKEYs for InfiniBand and Cray Aries networks typically includes InifiniBand and Aries key.
- In order to enable remote direct memory access to the memory associated with the memory handle the application is responsible to share the RKEY with the peers that will initiate the access.

Parameters

in	*context*	Application context which was used to allocate/map the memory.
in	*memh*	Handle to memory region.
out	*rkey_buffer*	Memory buffer allocated by the library. The buffer contains packed RKEY.
out	*size*	Size (in bytes) of the packed RKEY.

Returns

Error code as defined by ucs_status_t

5.4.3.4 void ucp_rkey_buffer_release (void * *rkey_buffer*)

This routine releases the buffer that was allocated using ucp_rkey_pack().

Warning

- Once memory is released an access to the memory may cause a failure.
- If the input memory address was not allocated using ucp_rkey_pack() routine the behaviour of this routine is undefined.

Parameters

in	*rkey_buffer*	Buffer to release.

5.4.3.5 ucs_status_t ucp_ep_rkey_unpack (ucp_ep_h *ep*, void * *rkey_buffer*, ucp_rkey_h * *rkey_p*)

This routine unpacks the remote key (RKEY) object into the local memory such that it can be accesses and used by UCP routines. The RKEY object has to be packed using the ucp_rkey_pack() routine. Application code should not make any alternations to the content of the RKEY buffer.

Parameters

in	*ep*	Endpoint to access using the remote key.
in	*rkey_buffer*	Packed rkey.
out	*rkey*	Remote key handle.

Returns

Error code as defined by ucs_status_t

5.4.3.6 void ucp_rkey_destroy (ucp_rkey_h *rkey*)

This routine destroys the RKEY object and the memory that was allocated using the ucp_ep_rkey_unpack() routine. This routine also releases any resources that are associated with the RKEY object.

Warning

- Once the RKEY object is released an access to the memory will cause an undefined failure.
- If the RKEY object was not created using ucp_rkey_unpack() routine the behaviour of this routine is undefined.

Parameters

in	*rkey*	Romote key to destroy.

5.4.3.7 ucs_status_t ucp_rmem_ptr (ucp_ep_h *ep*, void * *remote_addr*, ucp_rkey_h *rkey*, void ** *local_addr_p*)

This routine returns a local memory address for the remote address such that application can use the local address for direct memory load and store operations. If the underlaying hardware does not support this capability this routine will return a corresponding error.

Parameters

in	*ep*	Endpoint handle that was used for rkey object creation and is used for the remote memory address.
in	*remote_addr*	Remote address to translate.
out	*rkey*	Remote key handle for the remote address.
out	*local_addr*	Local memory address that can by accessed directly using memory load and store operations.

Returns

Error code as defined by ucs_status_t

5.5 UCP Endpoint

Typedefs

- typedef struct ucp_ep * ucp_ep_h

  UCP Endpoint.

Functions

- ucs_status_t ucp_ep_create (ucp_worker_h worker, ucp_address_t *address, ucp_ep_h *ep_p)

  Create and connect an endpoint.
- void ucp_ep_destroy (ucp_ep_h ep)

  Destroy and release the endpoint.

5.5.1 Detailed Description

UCP Endpoint routines

5.5.2 Typedef Documentation

5.5.2.1 typedef struct ucp_ep* ucp_ep_h

The endpoint handle is an opaque object that is used to address a remote worker. It typically provides a description of source, destination, or both. All UCP communication routines address a destination with the endpoint handle. The endpoint handle is associated with only one UCP context. UCP provides the endpoint create routine to create the endpoint handle and the destroy routine to destroy the endpoint handle.

5.5.3 Function Documentation

5.5.3.1 ucs_status_t ucp_ep_create (ucp_worker_h *worker,* ucp_address_t * *address,* ucp_ep_h * *ep_p*)

This routine creates and connects an endpoint on a local worker for a destination address that identifies the remote worker. This function is non-blocking, and communications may begin immediately after it returns. If the connection process is not completed, communications may be delayed. The created endpoint is associated with one and only one worker.

Parameters

in	*worker*	Handle to the worker; the endpoint is associated with the worker.
in	*address*	Destination address; the address must be obtained using ucp_worker_get_address() routine.
out	*ep_p*	A handle to the created endpoint.

Returns

Error code as defined by ucs_status_t

5.5.3.2 void ucp_ep_destroy (ucp_ep_h *ep*)

This routine releases an endpoint. The release process flushes, locally, all outstanding communication operations and releases all memory context associated with the endpoints. Once the endpoint is destroyed, application cannot access it anymore. Nevertheless, if the application is interested, to re-initiate communication with a particular endpoint it can use endpoints create routine to re-open the endpoint.

Parameters

in	*ep*	Handle to the remote endpoint.

5.6 UCP Communication routines

Typedefs

- typedef uint64_t ucp_tag_t

  UCP Tag Identifier.
- typedef uint64_t ucp_datatype_t

  UCP Datatype Identifier.
- typedef void(* ucp_send_callback_t) (void *request, ucs_status_t status)

  Completion callback for non-blocking sends.
- typedef void(* ucp_tag_recv_callback_t) (void *request, ucs_status_t status, ucp_tag_recv_info_t *info)

  Completion callback for non-blocking tag receives.

Functions

- ucs_status_ptr_t ucp_tag_send_nb (ucp_ep_h ep, const void *buffer, size_t count, ucp_datatype_t datatype, ucp_tag_t tag, ucp_send_callback_t cb)

  Non-blocking tagged-send operations.
- ucs_status_ptr_t ucp_tag_recv_nb (ucp_worker_h worker, void *buffer, size_t count, ucp_datatype_t datatype, ucp_tag_t tag, ucp_tag_t tag_mask, ucp_tag_recv_callback_t cb)

  Non-blocking tagged-receive operation.
- ucp_tag_message_h ucp_tag_probe_nb (ucp_worker_h worker, ucp_tag_t tag, ucp_tag_t tag_mask, int remove, ucp_tag_recv_info_t *info)

  Non-blocking probe and return a message.
- ucs_status_ptr_t ucp_tag_msg_recv_nb (ucp_worker_h worker, void *buffer, size_t count, ucp_datatype_t datatype, ucp_tag_message_h message, ucp_tag_recv_callback_t cb)

  Non-blocking receive operation for a probed message.
- ucs_status_t ucp_put (ucp_ep_h ep, const void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)

  Blocking remote memory put operation.
- ucs_status_t ucp_get (ucp_ep_h ep, void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)

  Blocking remote memory get operation.
- ucs_status_t ucp_atomic_add32 (ucp_ep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey)

  Blocking atomic add operation for 32 bit integers.
- ucs_status_t ucp_atomic_add64 (ucp_ep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey)

  Blocking atomic add operation for 64 bit integers.
- ucs_status_t ucp_atomic_fadd32 (ucp_ep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)

  Blocking atomic fetch and add operation for 32 bit integers.
- ucs_status_t ucp_atomic_fadd64 (ucp_ep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)

  Blocking atomic fetch and add operation for 64 bit integers.
- ucs_status_t ucp_atomic_swap32 (ucp_ep_h ep, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)

  Blocking atomic swap operation for 32 bit values.
- ucs_status_t ucp_atomic_swap64 (ucp_ep_h ep, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)

  Blocking atomic swap operation for 64 bit values.
- ucs_status_t ucp_atomic_cswap32 (ucp_ep_h ep, uint32_t compare, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)

  Blocking atomic conditional swap (cswap) operation for 32 bit values.
- ucs_status_t ucp_atomic_cswap64 (ucp_ep_h ep, uint64_t compare, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)

Blocking atomic conditional swap (cswap) operation for 64 bit values.

- int ucp_request_is_completed (void *request)

  Check if a non-blocking request is completed.

- void ucp_request_release (void *request)

  Release a communications request.

- void ucp_request_cancel (ucp_worker_h worker, void *request)

  Cancel an outstanding communications request.

5.6.1 Detailed Description

UCP Communication routines

5.6.2 Typedef Documentation

5.6.2.1 typedef uint64_t ucp_tag_t

UCP tag identifier is a 64bit object used for message identification. UCP tag send and receive operations use the object for an implementation tag matching semantics (derivative of MPI tag matching semantics).

5.6.2.2 typedef uint64_t ucp_datatype_t

UCP datatype identifier is a 64bit object used for datatype identification. Predefined UCP identifiers are defined by ucp_dt_type.

5.6.2.3 typedef void(* ucp_send_callback_t) (void *request, ucs_status_t status)

This callback routine is invoked whenever the send operation is completed. It is important to note that the call-back is only invoked in a case when the operation cannot be completed in place.

Parameters

in	*request*	The completed send request.
in	*status*	Completion status. If the send operation was completed successfully UCX_OK is returned. If send operation was canceled UCS_ERR_CANCELED is returned. Otherwise, an error status is returned.

5.6.2.4 typedef void(* ucp_tag_recv_callback_t) (void *request, ucs_status_t status, ucp_tag_recv_info_t *info)

This callback routine is invoked whenever the receive operation is completed and the data is ready in the receive buffer.

Parameters

in	*request*	The completed receive request.
in	*status*	Completion status. If the send operation was completed successfully UCX_OK is returned. If send operation was canceled UCS_ERR_CANCELED is returned. If the data can not fit into the receive buffer the UCS_ERR_TRUNCATED error code is returned. Otherwise, an error status is returned.

in	*info*	Completion information The *info* descriptor is Valid only if the status is UCS↩ _OK.

5.6.3 Function Documentation

5.6.3.1 ucs_status_ptr_t ucp_tag_send_nb (ucp_ep_h *ep*, const void * *buffer*, size_t *count*, ucp_datatype_t *datatype*, ucp_tag_t *tag*, ucp_send_callback_t *cb*)

This routine sends a messages that is described by the local address *buffer*, size *count*, and *datatype* object to the destination endpoint *ep*. Each message is associated with a *tag* value that is used for message matching on the receiver. The routine is non-blocking and therefore returns immediately, however the actual send operation may be delayed. The send operation is considered completed when it is safe to reuse the source *buffer*. If the operation is completed immediately the routine return UCS_OK and the call-back function *cb* is **not** invoked. If the operation is **not** completed immediately and no error reported then the UCP library will schedule to invoke the call-back *cb* whenever the send operation will be completed. In other words, the completion of a message can be signaled by the return code or the call-back.

Parameters

in	*ep*	Destination endpoint handle.
in	*buffer*	Pointer to the message buffer (payload).
in	*count*	Number of elements to send
in	*datatype*	Datatype descriptor for the elements in the buffer.
in	*tag*	Message tag.
in	*cb*	Callback function that is invoked whenever the send operation is completed. It is important to note that the call-back is only invoked in a case when the operation cannot be completed in place.

Returns

UCS_OK - The send operation was completed completed immediately.
UCS_PTR_IS_ERR(_ptr) - The send operation failed.
otherwise - Operation was scheduled for send. The request handle is returned to the application in order to track progress of the message. The application is responsible to released the handle using ucp_request_↩ release() routine.

5.6.3.2 ucs_status_ptr_t ucp_tag_recv_nb (ucp_worker_h *worker*, void * *buffer*, size_t *count*, ucp_datatype_t *datatype*, ucp_tag_t *tag*, ucp_tag_t *tag_mask*, ucp_tag_recv_callback_t *cb*)

This routine receives a messages that is described by the local address *buffer*, size *count*, and *datatype* object on the *worker*. The tag value of the receive message has to match the *tag* and *tag_mask* values, where the indicates what bits of the tag have to be matched. The routine is a non-blocking and therefore returns immediately. The receive operation is considered completed when the message is delivered to the *buffer*. In order to notify the application about completion of the receive operation the UCP library will invoke the call-back *cb* when the received message is in the receive buffer and ready for application access. If the receive operation cannot be stated the routine returns an error.

Note

This routine cannot return UCS_OK. It always returns a request handle or an error.

Parameters

in	*worker*	UCP worker that is used for the receive operation.
in	*buffer*	Pointer to the buffer to receive the data to.
in	*count*	Number of elements to receive
in	*datatype*	Datatype descriptor for the elements in the buffer.
in	*tag*	Message tag to expect.
in	*tag_mask*	Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.
in	*cb*	Callback function that is invoked whenever the receive operation is completed and the data is ready in the receive *buffer*.

Returns

UCS_PTR_IS_ERR(_ptr) - The receive operation failed.
otherwise - Operation was scheduled for receive. The request handle is returned to the application in order to track progress of the operation. The application is responsible to released the handle using ucp_request_↩release() routine.

5.6.3.3 ucp_tag_message_h ucp_tag_probe_nb (ucp_worker_h *worker*, ucp_tag_t *tag*, ucp_tag_t *tag_mask*, int *remove*, ucp_tag_recv_info_t * *info*)

This routine probes (checks) if a messages described by the *tag* and was received (fully or partially) on the *worker*. The tag value of the received message has to match the *tag* and *tag_mask* values, where the indicates what bits of the tag have to be matched. The function returns immediately and if the message is matched it returns a handle for the message.

Parameters

in	*worker*	UCP worker that is used for the probe operation.
in	*tag*	Message tag to probe for.
in	*tag_mask*	Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.
in	*remove*	The flag indicates if the matched message has to be removed from UCP library. If true (1), the message handle is removed from the UCP library and the application is responsible to call ucp_tag_msg_recv_nb() in order to receive the data and release the resources associated with the message handle.
out	*info*	If the matching message is found the descriptor is filled with the details about the message.

Returns

NULL - No match found.
Message handle (not NULL) - If message is matched the message handle is returned.

5.6.3.4 ucs_status_ptr_t ucp_tag_msg_recv_nb (ucp_worker_h *worker*, void * *buffer*, size_t *count*, ucp_datatype_t *datatype*, ucp_tag_message_h *message*, ucp_tag_recv_callback_t *cb*)

This routine receives a messages that is described by the local address *buffer*, size *count*, *message* handle, and *datatype* object on the *worker*. The handle can be obtain by calling the ucp_tag_probe_nb() routine. ucp_tag↩_msg_recv_nb() routine is a non-blocking and therefore returns immediately. The receive operation is considered completed when the message is delivered to the *buffer*. In order to notify the application about completion of the receive operation the UCP library will invoke the call-back *cb* when the received message is in the receive buffer and ready for application access. If the receive operation cannot be stated the routine returns an error.

Parameters

in	*worker*	UCP worker that is used for the receive operation.
in	*buffer*	Pointer to the buffer to receive the data to.
in	*count*	Number of elements to receive
in	*datatype*	Datatype descriptor for the elements in the buffer.
in	*message*	Message handle.
in	*cb*	Callback function that is invoked whenever the receive operation is completed and the data is ready in the receive *buffer*.

Returns

UCS_PTR_IS_ERR(_ptr) - The receive operation failed.
otherwise - Operation was scheduled for receive. The request handle is returned to the application in order to track progress of the operation. The application is responsible to released the handle using ucp_request_↩release() routine.

5.6.3.5 ucs_status_t ucp_put (ucp_ep_h *ep*, const void * *buffer*, size_t *length*, uint64_t *remote_addr*, ucp_rkey_h *rkey*)

This routine stores contiguous block of data that is described by the local address *buffer* in the remote contiguous memory region described by *remote_addr* address and the memory handle *rkey*. The routine returns when it is safe to reuse the source address *buffer*.

Parameters

in	*ep*	Remote endpoint handle.
in	*buffer*	Pointer to the local source address.
in	*length*	Length of the data (in bytes) stored under the source address.
in	*remote_addr*	Pointer to the destination remote address to write to.
in	*rkey*	Remote memory key associated with the remote address.

Returns

Error code as defined by ucs_status_t

5.6.3.6 ucs_status_t ucp_get (ucp_ep_h *ep*, void * *buffer*, size_t *length*, uint64_t *remote_addr*, ucp_rkey_h *rkey*)

This routine loads contiguous block of data that is described by the remote address *remote_addr* and the memory handle *rkey* in the local contiguous memory region described by *buffer* address. The routine returns when remote data is loaded and stored under the local address *buffer*.

Parameters

in	*ep*	Remote endpoint handle.
in	*buffer*	Pointer to the local source address.
in	*length*	Length of the data (in bytes) stored under the source address.
in	*remote_addr*	Pointer to the destination remote address to write to.
in	*rkey*	Remote memory key associated with the remote address.

Returns

Error code as defined by ucs_status_t

5.6.3.7 ucs_status_t ucp_atomic_add32 (ucp_ep_h *ep*, uint32_t *add*, uint64_t *remote_addr*, ucp_rkey_h *rkey*)

This routine performs an add operation on a 32 bit integer value atomically. The remote integer value is described by the combination of the remote memory address *remote_addr* and the remote memory handle *rkey*. The *add* value

is the value that is used for the add operation. When the operation completes the sum of the original remote value and the operand value (*add*) is stored in remote memory. The call to the routine returns immediately, independent of operation completion.

Note

The remote address must be aligned to 32 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*add*	Value to add.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.

Returns

Error code as defined by ucs_status_t

5.6.3.8 ucs_status_t ucp_atomic_add64 (ucp_ep_h *ep*, uint64_t *add*, uint64_t *remote_addr*, ucp_rkey_h *rkey*)

This routine performs an add operation on a 64 bit integer value atomically. The remote integer value is described by the combination of the remote memory address *remote_addr* and the remote memory handle *rkey*. The *add* value is the value that is used for the add operation. When the operation completes the sum of the original remote value and the operand value (*add*) is stored in remote memory. The call to the routine returns immediately, independent of operation completion.

Note

The remote address must be aligned to 64 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*add*	Value to add.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.

Returns

Error code as defined by ucs_status_t

5.6.3.9 ucs_status_t ucp_atomic_fadd32 (ucp_ep_h *ep*, uint32_t *add*, uint64_t *remote_addr*, ucp_rkey_h *rkey*, uint32_t * *result*)

This routine performs an add operation on a 32 bit integer value atomically. The remote integer value is described by the combination of the remote memory address *remote_addr* and the remote memory handle *rkey*. The *add* value is the value that is used for the add operation. When the operation completes, the original remote value is stored in the local memory *result*, and the sum of the original remote value and the operand value is stored in remote memory. The call to the routine returns when the operation is completed and the *result* value is updated.

Note

The remote address must be aligned to 32 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*add*	Value to add.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.
out	*result*	Pointer to the address that is used to store the previous value of the atomic variable described by the *remote_addr*

Returns

Error code as defined by ucs_status_t

5.6.3.10 ucs_status_t ucp_atomic_fadd64 (ucp_ep_h *ep*, uint64_t *add*, uint64_t *remote_addr*, ucp_rkey_h *rkey*, uint64_t * *result*)

This routine performs an add operation on a 64 bit integer value atomically. The remote integer value is described by the combination of the remote memory address *remote_addr* and the remote memory handle *rkey*. The *add* value is the value that is used for the add operation. When the operation completes, the original remote value is stored in the local memory *result*, and the sum of the original remote value and the operand value is stored in remote memory. The call to the routine returns when the operation is completed and the *result* value is updated.

Note

The remote address must be aligned to 64 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*add*	Value to add.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.
out	*result*	Pointer to the address that is used to store the previous value of the atomic variable described by the *remote_addr*

Returns

Error code as defined by ucs_status_t

5.6.3.11 ucs_status_t ucp_atomic_swap32 (ucp_ep_h *ep*, uint32_t *swap*, uint64_t *remote_addr*, ucp_rkey_h *rkey*, uint32_t * *result*)

This routine swaps a 32 bit value between local and remote memory. The remote value is described by the combination of the remote memory address *remote_addr* and the remote memory handle *rkey*. The *swap* value is the value that is used for the swap operation. When the operation completes, the remote value is stored in the local memory *result*, and the operand value (*swap*) is stored in remote memory. The call to the routine returns when the operation is completed and the *result* value is updated.

Note

The remote address must be aligned to 32 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*swap*	Value to swap.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.
out	*result*	Pointer to the address that is used to store the previous value of the atomic variable described by the *remote_addr*

Returns

Error code as defined by ucs_status_t

5.6.3.12 ucs_status_t ucp_atomic_swap64 (ucp_ep_h *ep*, uint64_t *swap*, uint64_t *remote_addr*, ucp_rkey_h *rkey*, uint64_t * *result*)

This routine swaps a 64 bit value between local and remote memory. The remote value is described by the combination of the remote memory address *remote_addr* and the remote memory handle *rkey*. The *swap* value is the value that is used for the swap operation. When the operation completes, the remote value is stored in the local memory *result*, and the operand value (*swap*) is stored in remote memory. The call to the routine returns when the operation is completed and the *result* value is updated.

Note

The remote address must be aligned to 64 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*swap*	Value to swap.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.
out	*result*	Pointer to the address that is used to store the previous value of the atomic variable described by the *remote_addr*

Returns

Error code as defined by ucs_status_t

5.6.3.13 ucs_status_t ucp_atomic_cswap32 (ucp_ep_h *ep*, uint32_t *compare*, uint32_t *swap*, uint64_t *remote_addr*, ucp_rkey_h *rkey*, uint32_t * *result*)

This routine conditionally swaps a 32 bit value between local and remote memory. The swap occurs only if the condition value (*continue*) is equal to the remote value, otherwise the remote memory is not modified. The remote value is described by the combination of the remote memory address `remote_addr` and the remote memory handle *rkey*. The `swap` value is the value that is used to update the remote memory if the condition is true. The call to the routine returns when the operation is completed and the *result* value is updated.

Note

The remote address must be aligned to 32 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*compare*	Value to compare to.
in	*swap*	Value to swap.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.
out	*result*	Pointer to the address that is used to store the previous value of the atomic variable described by the *remote_addr*

Returns

Error code as defined by ucs_status_t

5.6.3.14 ucs_status_t ucp_atomic_cswap64 (ucp_ep_h *ep*, uint64_t *compare*, uint64_t *swap*, uint64_t *remote_addr*, ucp_rkey_h *rkey*, uint64_t * *result*)

This routine conditionally swaps a 64 bit value between local and remote memory. The swap occurs only if the condition value (*continue*) is equal to the remote value, otherwise the remote memory is not modified. The remote value is described by the combination of the remote memory address `remote_addr` and the remote memory handle *rkey*. The `swap` value is the value that is used to update the remote memory if the condition is true. The call to the routine returns when the operation is completed and the *result* value is updated.

Note

The remote address must be aligned to 64 bit.

Parameters

in	*ep*	Remote endpoint handle.
in	*compare*	Value to compare to.
in	*swap*	Value to swap.
in	*remote_addr*	Pointer to the destination remote address of the atomic variable.
in	*rkey*	Remote memory key associated with the remote address.
out	*result*	Pointer to the address that is used to store the previous value of the atomic variable described by the *remote_addr*

Returns

Error code as defined by ucs_status_t

5.6.3.15 int ucp_request_is_completed (void * *request*)

Parameters

in	*request*	Non-blocking request to check.

Returns

Whether the request is completed.

5.6.3.16 void ucp_request_release (void * *request*)

Parameters

in	*request*	Non-blocking request to release.

Note

If the request is not completed yet, it will actually be released when completed.

5.6.3.17 void ucp_request_cancel (ucp_worker_h *worker,* void * *request*)

Parameters

in	*worker*	UCP worker.
in	*request*	Non-blocking request to cancel.

A request can either be completed or canceled, but not both. After calling this function, the request will complete regardless of what the remote side is doing. If the request is completed successfully, the completion callback will be called with the status parameter equals to UCS_OK, and in case it's canceled the status would be UCS_ERR↩ _CANCELED.

5.7 UCP Configuration

Data Structures

- struct ucp_params

  Tuning parameters for UCP library. More...

Typedefs

- typedef struct ucp_params ucp_params_t

  Tuning parameters for UCP library.
- typedef struct ucp_config ucp_config_t

  UCP configuration descriptor.

Functions

- ucs_status_t ucp_config_read (const char *env_prefix, const char *filename, ucp_config_t **config_p)

  Read UCP configuration descriptor.
- void ucp_config_release (ucp_config_t *config)

  Release configuration descriptor.
- void ucp_config_print (const ucp_config_t *config, FILE *stream, const char *title, ucs_config_print_flags_t print_flags)

  Print configuration information.

5.7.1 Detailed Description

This section describes routines for configuration of the UCP network layer

5.7.2 Data Structure Documentation

5.7.2.1 struct ucp_params

The structure defines the parameters that are used for UCP library tuning during UCP library initialization.

Note

UCP library implementation uses the features parameter to optimize the library functionality that minimize memory footprint. For example, if the application does not require send/receive semantics UCP library may avoid allocation of expensive resources associated with send/receive queues.

Data Fields

uint64_t	features	UCP features that are used for library initialization. It is recommend for applications only request the features that are required for an optimal functionality
size_t	request_size	The size of a reserved space in a non-blocking requests. Typically applications use the this space for caching own structures in order avoid costly memory allocations, pointer dereferences, and cache misses. For example, MPI implementation can use this memory for caching MPI descriptors

ucp_request_↩ init_callback_t	request_init	Pointer to a routine that is used for the request initialization. *NULL* can be used if no such function required.
ucp_request_↩ cleanup_↩ callback_t	request_cleanup	Pointer to a routine that is responsible for cleanup the memory associated with the request. *NULL* can be used if no such function required.

5.7.3 Typedef Documentation

5.7.3.1 typedef struct ucp_params ucp_params_t

The structure defines the parameters that are used for UCP library tuning during UCP library initialization.

Note

UCP library implementation uses the features parameter to optimize the library functionality that minimize memory footprint. For example, if the application does not require send/receive semantics UCP library may avoid allocation of expensive resources associated with send/receive queues.

5.7.3.2 typedef struct ucp_config ucp_config_t

This descriptor defines the configuration for UCP application context. The configuration is loaded from the run-time environment (using configuration files of environment variables) using ucp_config_read routine and can be printed using ucp_config_print routine. In addition, application is responsible to release the descriptor using ucp_config_↩ release routine.

5.7.4 Function Documentation

5.7.4.1 ucs_status_t ucp_config_read (const char * *env_prefix*, const char * *filename*, ucp_config_t ** *config_p*)

The routine fetches the information about UCP library configuration from the run-time environment. Then, the fetched descriptor is used for UCP library initialization. The Application can print out the descriptor using print routine. In addition the application is responsible to release the descriptor back to UCP library.

Parameters

in	*env_prefix*	If non-NULL, the routine searches for the environment variables that start with *UCX_<env_prefix>_* prefix. Otherwise, the routine searches for the environment variables that start with *UCX_* prefix.
in	*filename*	If non-NULL, read configuration from the file defined by *filename*. If the file does not exist, it will be ignored and no error reported to the application.
out	*config_p*	Pointer to configuration descriptor as defined by ucp_config_t.

Returns

Error code as defined by ucs_status_t

5.7.4.2 void ucp_config_release (ucp_config_t * *config*)

The routine releases the configuration descriptor that was allocated through ucp_config_read() routine.

Parameters

out	*config*	Configuration descriptor as defined by ucp_config_t.

5.7.4.3 void ucp_config_print (const ucp_config_t * *config*, FILE * *stream*, const char * *title*, ucs_config_print_flags_t *print_flags*)

The routine prints the configuration information that is stored in configuration descriptor.

Parameters

in	*config*	Configuration descriptor to print.
in	*stream*	Output stream to print the configuration to.
in	*title*	Configuration title to print.
in	*print_flags*	Flags that control various printing options.

5.8 UCP Data type routines

Data Structures

- struct ucp_generic_dt_ops

  UCP generic data type descriptor.

Macros

- #define ucp_dt_make_contig(_elem_size) (((ucp_datatype_t)(_elem_size) << UCP_DATATYPE_SHIFT) | UCP_DATATYPE_CONTIG)

  Generate an identifier for contiguous data type.

Typedefs

- typedef struct ucp_generic_dt_ops ucp_generic_dt_ops_t

  UCP generic data type descriptor.

Enumerations

- enum ucp_dt_type {
  UCP_DATATYPE_CONTIG = 0, UCP_DATATYPE_STRIDED = 1, UCP_DATATYPE_GENERIC = 7, UCP_DATATYPE_SHIFT = 3,
  UCP_DATATYPE_CLASS_MASK = UCS_MASK(UCP_DATATYPE_SHIFT) }

  UCP data type classification.

Functions

- ucs_status_t ucp_dt_create_generic (const ucp_generic_dt_ops_t *ops, void *context, ucp_datatype_t *datatype_p)

  Create a generic datatype.
- void ucp_dt_destroy (ucp_datatype_t datatype)

  Destroy a datatype and release its resources.

Variables

- void *(* ucp_generic_dt_ops::start_pack)(void *context, const void *buffer, size_t count)

  Start a packing request.
- void *(* ucp_generic_dt_ops::start_unpack)(void *context, void *buffer, size_t count)

  Start an unpacking request.
- size_t(* ucp_generic_dt_ops::packed_size)(void *state)

  Get the total size of packed data.
- size_t(* ucp_generic_dt_ops::pack)(void *state, size_t offset, void *dest, size_t max_length)

  Pack data.
- ucs_status_t(* ucp_generic_dt_ops::unpack)(void *state, size_t offset, const void *src, size_t count)

  Unpack data.
- void(* ucp_generic_dt_ops::finish)(void *state)

  Finish packing/unpacking.

5.8.1 Detailed Description

UCP Data type routines

5.8.2 Macro Definition Documentation

5.8.2.1 #define ucp_dt_make_contig(*_elem_size*) (((ucp_datatype_t)(_elem_size) << UCP_DATATYPE_SHIFT) | UCP_DATATYPE_CONTIG)

This macro creates an identifier for contiguous datatype that is defined by the size of the basic element.

Parameters

in	*_elem_size*	Size of the basic element of the type.

Returns

Data-type identifier.

5.8.3 Typedef Documentation

5.8.3.1 typedef struct ucp_generic_dt_ops ucp_generic_dt_ops_t

This structure provides a generic datatype descriptor that is used for definition of application defined datatypes.

Typically, the descriptor is used for an integratoion with datatype engines implemented within MPI and SHMEM implementations.

5.8.4 Enumeration Type Documentation

5.8.4.1 enum ucp_dt_type

The enumeration list describes the datatypes supported by UCP.

Enumerator

UCP_DATATYPE_CONTIG Contiguous datatype

UCP_DATATYPE_STRIDED Strided datatype

UCP_DATATYPE_GENERIC Generic datatype with user-defined pack/unpack routines

UCP_DATATYPE_SHIFT Number of bits defining the datatype classification

UCP_DATATYPE_CLASS_MASK Data-type class mask

5.8.5 Function Documentation

5.8.5.1 ucs_status_t ucp_dt_create_generic (const ucp_generic_dt_ops_t * *ops*, void * *context*, ucp_datatype_t * *datatype_p*)

This routine create a generic datatype object. The generic datatype is described by the *ops* object which provides a table of routines defining the operations for generic datatype manipulation. Typically, generic datatypes are used for integration with datatype engines provided with MPI implementations (MPICH, Open MPI, etc). The application is responsible to release the *datatype_p* object using ucp_dt_destroy() routine.

Parameters

in	*ops*	Generic datatype function table as defined by ucp_generic_dt_ops_t .
in	*context*	Application defined context passed to this routine. The context is passed as a parameter to the routines in the *ops* table.
out	*datatype_p*	A pointer to datatype object.

Returns

Error code as defined by ucs_status_t

5.8.5.2 void ucp_dt_destroy (ucp_datatype_t *datatype*)

This routine destroys the *datatype* object and releases any resources that are associated with the object. The *datatype* object must be allocated using ucp_dt_create_generic() routine.

Warning

- Once the *datatype* object is released an access to this object may cause an undefined failure.

Parameters

in	*datatype*	Datatype object to destroy.

5.8.6 Variable Documentation

5.8.6.1 void*(* ucp_generic_dt_ops::start_pack) (void *context, const void *buffer, size_t count)

The pointer refers to application defined start-to-pack routine. It will be called from the ucp_tag_send_nb routine.

Parameters

in	*context*	User-defined context.
in	*buffer*	Buffer to pack.
in	*count*	Number of elements to pack into the buffer.

Returns

A custom state that is passed to the following pack() routine.

5.8.6.2 void*(* ucp_generic_dt_ops::start_unpack) (void *context, void *buffer, size_t count)

The pointer refers to application defined start-to-unpack routine. It will be called from the ucp_tag_recv_nb routine.

Parameters

in	*context*	User-defined context.
in	*buffer*	Buffer to unpack to.
in	*count*	Number of elements to unpack in the buffer.

Returns

A custom state that is passed later to the following unpack() routine.

5.8.6.3 size_t(* ucp_generic_dt_ops::packed_size) (void *state)

The pointer refers to user defined routine that returns the size of data in a packed format.

Parameters

in	*state*	State as returned by start_pack() routine.

Returns

The size of the data in a packed form.

5.8.6.4 size_t(∗ ucp_generic_dt_ops::pack) (void ∗state, size_t offset, void ∗dest, size_t max_length)

The pointer refers to application defined pack routine.

Parameters

in	*state*	State as returned by start_pack() routine.
in	*offset*	Virtual offset in the output stream.
in	*dest*	Destination to pack the data to.
in	*max_length*	Maximal length to pack.

Returns

The size of the data that was written to the destination buffer. Must be less than or equal to *max_length*.

5.8.6.5 ucs_status_t(∗ ucp_generic_dt_ops::unpack) (void ∗state, size_t offset, const void ∗src, size_t count)

The pointer refers to application defined unpack routine.

Parameters

in	*state*	State as returned by start_pack() routine.
in	*offset*	Virtual offset in the input stream.
in	*src*	Source to unpack the data from.
in	*length*	Length to unpack.

Returns

UCS_OK or an error if unpacking failed.

5.8.6.6 void(∗ ucp_generic_dt_ops::finish) (void ∗state)

The pointer refers to application defined finish routine.

Parameters

in	*state*	State as returned by start_pack() and start_unpack() routines.

5.9 Unified Communication Transport (UCT) API

Modules

- UCT Communication Resource
- UCT Communication Context
- UCT Protection Domain
- UCT Active messages
- UCT Remote memeory access operations.
- UCT Atomic operations.

5.9.1 Detailed Description

This section describes UCT API.

5.10 UCT Communication Resource

Data Structures

- struct uct_pd_resource_desc

  Protection domain resource descriptor. More...
- struct uct_tl_resource_desc

  Communication resource descriptor. More...
- struct uct_iface_attr

  Interface attributes: capabilities and limitations. More...
- struct uct_iface_attr.cap
- struct uct_iface_attr.cap.put
- struct uct_iface_attr.cap.get
- struct uct_iface_attr.cap.am
- struct uct_completion

  Completion handle. More...
- struct uct_pending_req

  Pending request. More...

Typedefs

- typedef struct uct_pd_resource_desc uct_pd_resource_desc_t

  Protection domain resource descriptor.
- typedef struct uct_tl_resource_desc uct_tl_resource_desc_t

  Communication resource descriptor.

Enumerations

- enum {
  UCT_IFACE_FLAG_AM_SHORT = UCS_BIT(0), UCT_IFACE_FLAG_AM_BCOPY = UCS_BIT(1), UCT_IFACE_FLAG_AM_ZCOPY = UCS_BIT(2), UCT_IFACE_FLAG_PENDING = UCS_BIT(3),
  UCT_IFACE_FLAG_PUT_SHORT = UCS_BIT(4), UCT_IFACE_FLAG_PUT_BCOPY = UCS_BIT(5), UCT_IFACE_FLAG_PUT_ZCOPY = UCS_BIT(6), UCT_IFACE_FLAG_GET_SHORT = UCS_BIT(8),
  UCT_IFACE_FLAG_GET_BCOPY = UCS_BIT(9), UCT_IFACE_FLAG_GET_ZCOPY = UCS_BIT(10), UCT_IFACE_FLAG_ATOMIC_ADD32 = UCS_BIT(16), UCT_IFACE_FLAG_ATOMIC_ADD64 = UCS_BIT(17),
  UCT_IFACE_FLAG_ATOMIC_FADD32 = UCS_BIT(18), UCT_IFACE_FLAG_ATOMIC_FADD64 = UCS_BIT(19), UCT_IFACE_FLAG_ATOMIC_SWAP32 = UCS_BIT(20), UCT_IFACE_FLAG_ATOMIC_SWAP64 = UCS_BIT(21),
  UCT_IFACE_FLAG_ATOMIC_CSWAP32 = UCS_BIT(22), UCT_IFACE_FLAG_ATOMIC_CSWAP64 = UCS_BIT(23), UCT_IFACE_FLAG_ERRHANDLE_SHORT_BUF = UCS_BIT(32), UCT_IFACE_FLAG_ERRHANDLE_BCOPY_BUF = UCS_BIT(33),
  UCT_IFACE_FLAG_ERRHANDLE_ZCOPY_BUF = UCS_BIT(34), UCT_IFACE_FLAG_ERRHANDLE_AM_ID = UCS_BIT(35), UCT_IFACE_FLAG_ERRHANDLE_REMOTE_MEM = UCS_BIT(35), UCT_IFACE_FLAG_ERRHANDLE_BCOPY_LEN = UCS_BIT(36),
  UCT_IFACE_FLAG_CONNECT_TO_IFACE = UCS_BIT(40), UCT_IFACE_FLAG_CONNECT_TO_EP = UCS_BIT(41), UCT_IFACE_FLAG_AM_THREAD_SINGLE = UCS_BIT(44) }

  List of capabilities supported by UCX API.

Functions

- ucs_status_t uct_query_pd_resources (uct_pd_resource_desc_t **resources_p, unsigned *num_resources_p)

  Query for memory resources.

- void uct_release_pd_resource_list (uct_pd_resource_desc_t *resources)

  Release the list of resources returned from uct_query_pd_resources.
- ucs_status_t uct_pd_open (const char *pd_name, const uct_pd_config_t *config, uct_pd_h *pd_p)

  Open a protection domain.
- void uct_pd_close (uct_pd_h pd)

  Close a protection domain.
- ucs_status_t uct_pd_query_tl_resources (uct_pd_h pd, uct_tl_resource_desc_t **resources_p, unsigned *num_resources_p)

  Query for transport resources.
- void uct_release_tl_resource_list (uct_tl_resource_desc_t *resources)

  Release the list of resources returned from uct_pd_query_tl_resources.
- ucs_status_t uct_iface_config_read (const char *tl_name, const char *env_prefix, const char *filename, uct_iface_config_t **config_p)

  Read transport-specific interface configuration.
- void uct_config_release (void *config)

  Release configuration memory returned from uct_iface_config_read() or from uct_pd_config_read().
- void uct_config_print (const void *config, FILE *stream, const char *title, ucs_config_print_flags_t print_flags)

  Print interface/PD configuration to a stream.
- ucs_status_t uct_iface_open (uct_pd_h pd, uct_worker_h worker, const char *tl_name, const char *dev_name, size_t rx_headroom, const uct_iface_config_t *config, uct_iface_h *iface_p)

  Open a communication interface.
- void uct_iface_close (uct_iface_h iface)

  Close and destroy an interface.
- ucs_status_t uct_iface_query (uct_iface_h iface, uct_iface_attr_t *iface_attr)

  Get interface attributes.
- ucs_status_t uct_iface_get_address (uct_iface_h iface, struct sockaddr *addr)

  Get interface address.
- int uct_iface_is_reachable (uct_iface_h iface, const struct sockaddr *addr)

  Check if remote iface address is reachable.
- ucs_status_t uct_iface_mem_alloc (uct_iface_h iface, size_t length, const char *name, uct_allocated_memory_t *mem)
- ucs_status_t uct_ep_create (uct_iface_h iface, uct_ep_h *ep_p)

  Create new endpoint.
- static ucs_status_t uct_ep_create_connected (uct_iface_h iface, const struct sockaddr *addr, uct_ep_h *ep_p)

  Create an endpoint which is connected to remote interface.
- void uct_ep_destroy (uct_ep_h ep)

  Destroy an endpoint.
- ucs_status_t uct_ep_get_address (uct_ep_h ep, struct sockaddr *addr)

  Get endpoint address.
- ucs_status_t uct_ep_connect_to_ep (uct_ep_h ep, const struct sockaddr *addr)

  Connect endpoint to a remote endpoint.
- UCT_INLINE_API ucs_status_t uct_iface_flush (uct_iface_h iface)
- UCT_INLINE_API ucs_status_t uct_ep_pending_add (uct_ep_h ep, uct_pending_req_t *req)

  Add a pending request to an endpoint.
- UCT_INLINE_API void uct_ep_pending_purge (uct_ep_h ep, uct_pending_callback_t cb)

  Remove all pending requests from an endpoint.
- UCT_INLINE_API ucs_status_t uct_ep_flush (uct_ep_h ep)

5.10.1 Detailed Description

This section describes a concept of the Communication Resource and routines associated with the concept.

5.10.2 Data Structure Documentation

5.10.2.1 struct uct_pd_resource_desc

This structure describes a protection domain resource.

Data Fields

char	pd_name[UCT_PD_NAME_MAX]	Protection domain name

5.10.2.2 struct uct_tl_resource_desc

Resource descriptor is an object representing the network resource. Resource descriptor could represent a stand-alone communication resource such as an HCA port, network interface, or multiple resources such as multiple network interfaces or communication ports. It could also represent virtual communication resources that are defined over a single physical network interface.

Data Fields

char	tl_name[UCT_TL_NAME_MAX]	Transport name
char	dev_name[UCT_DEVICE_NAME_MAX]	Hardware device name
uint64_t	latency	Latency, nanoseconds
size_t	bandwidth	Bandwidth, bytes/second

5.10.2.3 struct uct_iface_attr

Data Fields

struct uct_iface_attr	cap	
size_t	iface_addr_len	Size of interface address
size_t	ep_addr_len	Size of endpoint address

5.10.2.4 struct uct_iface_attr.cap

Data Fields

cap	put	
cap	get	
cap	am	

uint64_t	flags	Flags from UCT_IFACE_FLAG_xx

5.10.2.5 struct uct_iface_attr.cap.put

Data Fields

size_t	max_short	Maximal size for put_short
size_t	max_bcopy	Maximal size for put_bcopy
size_t	max_zcopy	Maximal size for put_zcopy

5.10.2.6 struct uct_iface_attr.cap.get

Data Fields

size_t	max_bcopy	Maximal size for get_bcopy
size_t	max_zcopy	Maximal size for get_zcopy

5.10.2.7 struct uct_iface_attr.cap.am

Data Fields

size_t	max_short	Total max. size (incl. the header)
size_t	max_bcopy	Total max. size (incl. the header)
size_t	max_zcopy	Total max. size (incl. the header)
size_t	max_hdr	Max. header size for bcopy/zcopy

5.10.2.8 struct uct_completion

This structure should be allocated by the user and can be passed to communication primitives. User has to initializes both fields of the structure. If the operation returns UCS_INPROGRESS, this structure will be in use by the transport until the operation completes. When the operation completes, "count" field is decremented by 1, and whenever it reaches 0 - the callback is called.

Notes:

- The same structure can be passed multiple times to communication functions without the need to wait for completion.
- If the number of operations is smaller than the initial value of the counter, the callback will not be called at all, so it may be left undefined.

Data Fields

uct_↩ completion_↩ callback_t	func	User callback function
int	count	Completion counter

5.10.2.9 struct uct_pending_req

This structure should be passed to uct_pending_add() and is used to signal new available resources back to user.

Data Fields

uct_pending_↩ callback_t	func	User callback function
char	priv[UCT_PEN↩ DING_REQ_P↩ RIV_LEN]	Used internally by UCT

5.10.3 Typedef Documentation

5.10.3.1 typedef struct uct_pd_resource_desc uct_pd_resource_desc_t

This structure describes a protection domain resource.

5.10.3.2 typedef struct uct_tl_resource_desc uct_tl_resource_desc_t

Resource descriptor is an object representing the network resource. Resource descriptor could represent a stand-alone communication resource such as an HCA port, network interface, or multiple resources such as multiple network interfaces or communication ports. It could also represent virtual communication resources that are defined over a single physical network interface.

5.10.4 Enumeration Type Documentation

5.10.4.1 anonymous enum

The enumeration list presents a full list of operations and capabilities exposed by UCX API.

Enumerator

UCT_IFACE_FLAG_AM_SHORT Short active message

UCT_IFACE_FLAG_AM_BCOPY Buffered active message

UCT_IFACE_FLAG_AM_ZCOPY Zero-copy active message

UCT_IFACE_FLAG_PENDING Pending operations

UCT_IFACE_FLAG_PUT_SHORT Short put

UCT_IFACE_FLAG_PUT_BCOPY Buffered put

UCT_IFACE_FLAG_PUT_ZCOPY Zero-copy put

UCT_IFACE_FLAG_GET_SHORT Short get

UCT_IFACE_FLAG_GET_BCOPY Buffered get

UCT_IFACE_FLAG_GET_ZCOPY Zero-copy get

UCT_IFACE_FLAG_ATOMIC_ADD32 32bit atomic add

UCT_IFACE_FLAG_ATOMIC_ADD64 64bit atomic add

UCT_IFACE_FLAG_ATOMIC_FADD32 32bit atomic fetch-and-add

UCT_IFACE_FLAG_ATOMIC_FADD64 64bit atomic fetch-and-add

UCT_IFACE_FLAG_ATOMIC_SWAP32 32bit atomic swap

UCT_IFACE_FLAG_ATOMIC_SWAP64 64bit atomic swap

UCT_IFACE_FLAG_ATOMIC_CSWAP32 32bit atomic compare-and-swap

UCT_IFACE_FLAG_ATOMIC_CSWAP64 64bit atomic compare-and-swap

UCT_IFACE_FLAG_ERRHANDLE_SHORT_BUF Invalid buffer for short operation

UCT_IFACE_FLAG_ERRHANDLE_BCOPY_BUF Invalid buffer for buffered operation

UCT_IFACE_FLAG_ERRHANDLE_ZCOPY_BUF Invalid buffer for zero copy operation

UCT_IFACE_FLAG_ERRHANDLE_AM_ID Invalid AM id on remote

UCT_IFACE_FLAG_ERRHANDLE_REMOTE_MEM Remote memory access

UCT_IFACE_FLAG_ERRHANDLE_BCOPY_LEN Invalid length for buffered operation

UCT_IFACE_FLAG_CONNECT_TO_IFACE Supports connecting to interface

UCT_IFACE_FLAG_CONNECT_TO_EP Supports connecting to specific endpoint

UCT_IFACE_FLAG_AM_THREAD_SINGLE Active messages callback is invoked only from main thread

5.10.5 Function Documentation

5.10.5.1 ucs_status_t uct_query_pd_resources (uct_pd_resource_desc_t ** *resources_p*, unsigned * *num_resources_p*)

Obtain the list of protection domain resources available on the current system.

Parameters

out	*resources_p*	Filled with a pointer to an array of resource descriptors.
out	*num_resources_p*	Filled with the number of resources in the array.

Returns

Error code.

5.10.5.2 void uct_release_pd_resource_list (uct_pd_resource_desc_t * *resources*)

This routine releases the memory associated with the list of resources allocated by uct_query_pd_resources.

Parameters

in	*resources*	Array of resource descriptors to release.

5.10.5.3 ucs_status_t uct_pd_open (const char * *pd_name*, const uct_pd_config_t * *config*, uct_pd_h * *pd_p*)

Open a specific protection domain. All communications and memory operations are performed in the context of a specific protection domain. Therefore it must be created before communication resources.

Parameters

in	*pd_name*	Protection domain name, as returned from uct_query_pd_resources.
in	*config*	PD configuration options. Should be obtained from uct_pd_config_read() function, or point to PD-specific structure which extends uct_pd_config_t.
out	*pd_p*	Filled with a handle to the protection domain.

Returns

Error code.

5.10.5.4 void uct_pd_close (uct_pd_h *pd*)

Parameters

in	*pd*	Protection domain to close.

5.10.5.5 ucs_status_t uct_pd_query_tl_resources (uct_pd_h *pd*, uct_tl_resource_desc_t ** *resources_p*, unsigned * *num_resources_p*)

This routine queries the protection domain for communication resources that are available for it.

Parameters

in	*pd*	Handle to protection domain.
out	*resources_p*	Filled with a pointer to an array of resource descriptors.
out	*num_resources_p*	Filled with the number of resources in the array.

Returns

Error code.

5.10.5.6 void uct_release_tl_resource_list (uct_tl_resource_desc_t * *resources*)

This routine releases the memory associated with the list of resources allocated by uct_query_tl_resources.

Parameters

in	*resources*	Array of resource descriptors to release.

5.10.5.7 ucs_status_t uct_iface_config_read (const char * *tl_name*, const char * *env_prefix*, const char * *filename*, uct_iface_config_t ** *config_p*)

Parameters

in	*tl_name*	Transport name.
in	*env_prefix*	If non-NULL, search for environment variables starting with this UCT_<prefix>_. Otherwise, search for environment variables starting with just UCT_.
in	*filename*	If non-NULL, read configuration from this file. If the file does not exist, it will be ignored.
out	*config_p*	Filled with a pointer to configuration.

Returns

Error code.

5.10.5.8 void uct_config_release (void * *config*)

Parameters

in	*config*	Configuration to release.

5.10.5.9 void uct_config_print (const void * *config*, FILE * *stream*, const char * *title*, ucs_config_print_flags_t *print_flags*)

Parameters

in	*config*	Configuration to print.
in	*stream*	Output stream to print to.
in	*title*	Title to the output.
in	*print_flags*	Controls how the configuration is printed.

5.10.5.10 ucs_status_t uct_iface_open (uct_pd_h *pd*, uct_worker_h *worker*, const char * *tl_name*, const char * *dev_name*, size_t *rx_headroom*, const uct_iface_config_t * *config*, uct_iface_h * *iface_p*)

Parameters

in	*pd*	Protection domain to create the interface on.
in	*worker*	Handle to worker which will be used to progress communications on this interface.
in	*tl_name*	Transport name.
in	*dev_name*	Hardware device name,
in	*rx_headroom*	How much bytes to reserve before the receive segment.
in	*config*	Interface configuration options. Should be obtained from uct_iface_config_read() function, or point to transport-specific structure which extends uct_iface_config_t.
out	*iface_p*	Filled with a handle to opened communication interface.

Returns

Error code.

5.10.5.11 void uct_iface_close (uct_iface_h *iface*)

Parameters

in	*iface*	Interface to close.

5.10.5.12 ucs_status_t uct_iface_query (uct_iface_h *iface*, uct_iface_attr_t * *iface_attr*)

Parameters

in	*iface*	Interface to query.

5.10.5.13 ucs_status_t uct_iface_get_address (uct_iface_h *iface*, struct sockaddr * *addr*)

Parameters

in	*iface*	Interface to query.
out	*iface_addr*	Filled with interface address. The size of the buffer provided must be at least uct_iface_attr_t::iface_addr_len.

5.10.5.14 int uct_iface_is_reachable (uct_iface_h *iface*, const struct sockaddr * *addr*)

This function checks if a remote address can be reached from a local interface. If the function returns true, it does not necessarily mean a connection and/or data transfer would succeed, since the reachability check is a local operation it does not detect issues such as network mis-configuration or lack of connectivity.

Parameters

in	*iface*	Interface to check reachability from.
in	*addr*	Address to check reachability to.

Returns

Nonzero if reachable, 0 if not.

5.10.5.15 ucs_status_t uct_iface_mem_alloc (uct_iface_h *iface*, size_t *length*, const char * *name*, uct_allocated_memory_t * *mem*)

5.10.5.16 ucs_status_t uct_ep_create (uct_iface_h *iface*, uct_ep_h * *ep_p*)

Parameters

in	*iface*	Interface to create the endpoint on.
out	*ep_p*	Filled with handle to the new endpoint.

5.10.5.17 static ucs_status_t uct_ep_create_connected (uct_iface_h *iface*, const struct sockaddr * *addr*, uct_ep_h * *ep_p*) [inline], [static]

Parameters

in	*iface*	Interface to create the endpoint on.
in	*addr*	Remote interface address to connect to.
out	*ep_p*	Filled with handle to the new endpoint.

5.10.5.18 void uct_ep_destroy (uct_ep_h *ep*)

Parameters

in	*ep*	Endpoint to destroy.

5.10.5.19 ucs_status_t uct_ep_get_address (uct_ep_h *ep*, struct sockaddr * *addr*)

Parameters

in	*ep*	Endpoint to query.
out	*ep_addr*	Filled with endpoint address. The size of the buffer provided must be at least uct_iface_attr_t::ep_addr_len.

5.10.5.20 ucs_status_t uct_ep_connect_to_ep (uct_ep_h *ep*, const struct sockaddr * *addr*)

Parameters

in	*ep*	Endpoint to connect.
in	*iface_addr*	Remote interface address.

in	*ep_addr*	Remote endpoint address.

5.10.5.21 UCT_INLINE_API ucs_status_t uct_iface_flush (uct_iface_h *iface*)

5.10.5.22 UCT_INLINE_API ucs_status_t uct_ep_pending_add (uct_ep_h *ep,* uct_pending_req_t * *req*)

Add a pending request to the endpoint pending queue. The request will be dispatched when the endpoint could potentially have additional send resources.

Parameters

in	*ep*	Endpoint to add the pending request to.
in	*req*	Pending request, which would be dispatched when more resources become available. The user is expected to initialize the "func" field. After passed to the function, the request is owned by UCT, until the callback is called and returns UCS_OK.

Returns

UCS_OK - request added to pending queue UCS_ERR_BUSY - request was not added to pending queue, because send resources are available now. The user is advised to retry.

5.10.5.23 UCT_INLINE_API void uct_ep_pending_purge (uct_ep_h *ep,* uct_pending_callback_t *cb*)

Remove pending requests from the given endpoint and pass them to the provided callback function. The callback return value is ignored.

Parameters

in	*ep*	Endpoint to remove pending requests from.
in	*cb*	Callback to pass the removed requests to.

5.10.5.24 UCT_INLINE_API ucs_status_t uct_ep_flush (uct_ep_h *ep*)

5.11 UCT Communication Context

Enumerations

- enum uct_alloc_method_t {
  UCT_ALLOC_METHOD_PD, UCT_ALLOC_METHOD_HEAP, UCT_ALLOC_METHOD_MMAP, UCT_AL↩
  LOC_METHOD_HUGE,
  UCT_ALLOC_METHOD_LAST, UCT_ALLOC_METHOD_DEFAULT = UCT_ALLOC_METHOD_LAST }

  Memory allocation methods.

Functions

- ucs_status_t uct_worker_create (ucs_async_context_t *async, ucs_thread_mode_t thread_mode, uct_↩
  worker_h *worker_p)

  Create a worker object.
- void uct_worker_destroy (uct_worker_h worker)

  Destroy a worker object.
- void uct_worker_progress (uct_worker_h worker)

  Explicit progress for UCT worker.
- void uct_worker_progress_register (uct_worker_h worker, ucs_notifier_chain_func_t func, void *arg)

  Add a callback function to a worker progress.
- void uct_worker_progress_unregister (uct_worker_h worker, ucs_notifier_chain_func_t func, void *arg)

  Remove a callback function from worker's progress.
- ucs_status_t uct_config_modify (void *config, const char *name, const char *value)

  Modify interface/PD configuration.

5.11.1 Detailed Description

UCT context abstracts all the resources required for network communication. It is designed to enable either share or isolate resources for multiple programming models used by an application.

This section provides a detailed description of this concept and routines associated with it.

5.11.2 Enumeration Type Documentation

5.11.2.1 enum uct_alloc_method_t

Enumerator

UCT_ALLOC_METHOD_PD Allocate using protection domain

UCT_ALLOC_METHOD_HEAP Allocate from heap usign libc allocator

UCT_ALLOC_METHOD_MMAP Allocate from OS using mmap() syscall

UCT_ALLOC_METHOD_HUGE Allocate huge pages

UCT_ALLOC_METHOD_LAST

UCT_ALLOC_METHOD_DEFAULT Use default method

5.11.3 Function Documentation

5.11.3.1 ucs_status_t uct_worker_create (ucs_async_context_t * *async,* ucs_thread_mode_t *thread_mode,* uct_worker_h * *worker_p*)

The worker represents a progress engine. Multiple progress engines can be created in an application, for example to be used by multiple threads. Transports can allocate separate communication resources for every worker, so that every worker can be progressed independently of others.

Parameters

in	*async*	Context for async event handlers. Can be NULL, which means that event handlers will not have particular context.
in	*thread_mode*	Thread access mode to the worker and all interfaces and endpoints associated with it.
out	*worker_p*	Filled with a pointer to the worker object.

5.11.3.2 void uct_worker_destroy (uct_worker_h *worker*)

Parameters

in	*worker*	Worker object to destroy.

5.11.3.3 void uct_worker_progress (uct_worker_h *worker*)

This routine explicitly progresses any outstanding communication operations and active message requests.

Note

- In the current implementation, users **MUST** call this routine to receive the active message requests.

Parameters

in	*worker*	Handle to worker.

5.11.3.4 void uct_worker_progress_register (uct_worker_h *worker,* ucs_notifier_chain_func_t *func,* void * *arg*)

Add a function which will be called every time a progress is made on the worker.

Parameters

in	*worker*	Handle to worker.
in	*func*	Pointer to callback function.
in	*arg*	Argument to the function.

Note

If the same function and argument are already on the list, their reference count will be incremented. This operation could potentially be slow.

5.11.3.5 void uct_worker_progress_unregister (uct_worker_h *worker,* ucs_notifier_chain_func_t *func,* void * *arg*)

Remove a previously added function from worker's progress.

Parameters

in	*worker*	Handle to worker.
in	*func*	Pointer to callback function.
in	*arg*	Argument to the function.

Note

If the reference count of the function is >1, it will be decremented and the function will not be removed. This operation could potentially be slow.

5.11.3.6 ucs_status_t uct_config_modify (void * *config,* const char * *name,* const char * *value*)

Parameters

in	*config*	Configuration to modify.
in	*name*	Configuration variable name.
in	*value*	Value to set.

Returns

Error code.

5.12 UCT Protection Domain

Data Structures

- struct uct_pd_attr

  Protection domain attributes. More...
- struct uct_pd_attr.cap
- struct uct_allocated_memory

  Describes a memory allocated by UCT. More...
- struct uct_rkey_bundle

  Remote key with its type. More...

Typedefs

- typedef struct uct_allocated_memory uct_allocated_memory_t

  Describes a memory allocated by UCT.
- typedef struct uct_rkey_bundle uct_rkey_bundle_t

  Remote key with its type.

Enumerations

- enum { UCT_PD_FLAG_ALLOC = UCS_BIT(0), UCT_PD_FLAG_REG = UCS_BIT(1) }

  Protection domain capability flags.

Functions

- ucs_status_t uct_pd_query (uct_pd_h pd, uct_pd_attr_t *pd_attr)

  *Query for protection domain attributes. *.*
- ucs_status_t uct_pd_mem_alloc (uct_pd_h pd, size_t *length_p, void **address_p, const char *name, uct↩ _mem_h *memh_p)

  Allocate memory for zero-copy sends and remote access.
- ucs_status_t uct_pd_mem_free (uct_pd_h pd, uct_mem_h memh)

  Release memory allocated by uct_pd_mem_alloc.
- ucs_status_t uct_pd_mem_reg (uct_pd_h pd, void *address, size_t length, uct_mem_h *memh_p)

  Register memory for zero-copy sends and remote access.
- ucs_status_t uct_pd_mem_dereg (uct_pd_h pd, uct_mem_h memh)

  Undo the operation of uct_pd_mem_reg().
- ucs_status_t uct_mem_alloc (size_t min_length, uct_alloc_method_t *methods, unsigned num_methods, uct_pd_h *pds, unsigned num_pds, const char *name, uct_allocated_memory_t *mem)

  Allocate memory for zero-copy communications and remote access.
- ucs_status_t uct_mem_free (const uct_allocated_memory_t *mem)

  Release allocated memory.
- ucs_status_t uct_pd_mkey_pack (uct_pd_h pd, uct_mem_h memh, void *rkey_buffer)

  Pack a remote key.
- ucs_status_t uct_rkey_unpack (const void *rkey_buffer, uct_rkey_bundle_t *rkey_ob)

  Unpack a remote key.
- ucs_status_t uct_rkey_release (const uct_rkey_bundle_t *rkey_ob)

  Release a remote key.

5.12.1 Detailed Description

The Protection Domain abstracts resources required for network communication, which typically includes memory, transport mechanisms, compute and network resources. It is an isolation mechanism that can be employed by the applications for isolating resources between multiple programming models. The attributes of the Protection Domain are defined by the structure uct_pd_attr(). The communication and memory operations are defined in the context of Protection Domain.

5.12.2 Data Structure Documentation

5.12.2.1 struct uct_pd_attr

This structure defines the attributes of a Protection Domain which includes maximum memory that can be allocated, credentials required for accessing the memory, and CPU mask indicating the proximity of CPUs.

Data Fields

struct uct_pd_attr	cap	
char	component_name[UCT_PD_COMPONENT_NAME_MAX]	PD component name
size_t	rkey_packed_size	Size of buffer needed for packed rkey
cpu_set_t	local_cpus	Mask of CPUs near the resource

5.12.2.2 struct uct_pd_attr.cap

Data Fields

size_t	max_alloc	Maximal allocation size
size_t	max_reg	Maximal registration size
uint64_t	flags	UCT_PD_FLAG_xx

5.12.2.3 struct uct_allocated_memory

This structure describes the memory block which includes the address, size, and Protection Domain used for allocation. This structure is passed to interface and the memory is allocated by memory allocation functions uct_mem_alloc.

Data Fields

void *	address	Address of allocated memory
size_t	length	Real size of allocated memory
uct_alloc_method_t	method	Method used to allocate the memory
uct_pd_h	pd	if method==PD: PD used to allocate the memory
uct_mem_h	memh	if method==PD: PD memory handle

5.12.2.4 struct uct_rkey_bundle

This structure describes the credentials (typically key) and information required to access the remote memory by the communication interfaces.

Data Fields

uct_rkey_t	rkey	Remote key descriptor, passed to RMA functions
void *	handle	Handle, used internally for releasing the key
void *	type	Remote key type

5.12.3 Typedef Documentation

5.12.3.1 typedef struct uct_allocated_memory uct_allocated_memory_t

This structure describes the memory block which includes the address, size, and Protection Domain used for allocation. This structure is passed to interface and the memory is allocated by memory allocation functions uct_mem_alloc.

5.12.3.2 typedef struct uct_rkey_bundle uct_rkey_bundle_t

This structure describes the credentials (typically key) and information required to access the remote memory by the communication interfaces.

5.12.4 Enumeration Type Documentation

5.12.4.1 anonymous enum

Enumerator

UCT_PD_FLAG_ALLOC PD support memory allocation

UCT_PD_FLAG_REG PD support memory registration

5.12.5 Function Documentation

5.12.5.1 ucs_status_t uct_pd_query (uct_pd_h *pd*, uct_pd_attr_t * *pd_attr*)

Parameters

in	*pd*	Protection domain to query.
out	*pd_attr*	Filled with protection domain attributes.

5.12.5.2 ucs_status_t uct_pd_mem_alloc (uct_pd_h *pd*, size_t * *length_p*, void ** *address_p*, const char * *name*, uct_mem_h * *memh_p*)

Allocate memory on the protection domain. In order to use this function, PD must support UCT_PD_FLAG_ALLOC flag.

Parameters

in	*pd*	Protection domain to allocate memory on.
in,out	*length_p*	Points to the size of memory to allocate. Upon successful return, filled with the actual size that was allocated, which may be larger than the one requested. Must be >0.

in	*name*	Name of the allocated region, used to track memory usage for debugging and profiling.
out	*memh_p*	Filled with handle for allocated region.

5.12.5.3 ucs_status_t uct_pd_mem_free (uct_pd_h *pd*, uct_mem_h *memh*)

Parameters

in	*pd*	Protection domain memory was allocateed on.
in	*memh*	Memory handle, as returned from uct_pd_mem_alloc.

5.12.5.4 ucs_status_t uct_pd_mem_reg (uct_pd_h *pd*, void * *address*, size_t *length*, uct_mem_h * *memh_p*)

Register memory on the protection domain. In order to use this function, PD must support UCT_PD_FLAG_REG flag.

Parameters

in	*pd*	Protection domain to register memory on.
out	*address*	Memory to register.
in	*length*	Size of memory to register. Must be >0.
out	*memh_p*	Filled with handle for allocated region.

5.12.5.5 ucs_status_t uct_pd_mem_dereg (uct_pd_h *pd*, uct_mem_h *memh*)

Parameters

in	*pd*	Protection domain which was used to register the memory. [in] memh Local access key to memory region.

5.12.5.6 ucs_status_t uct_mem_alloc (size_t *min_length*, uct_alloc_method_t * *methods*, unsigned *num_methods*, uct_pd_h * *pds*, unsigned *num_pds*, const char * *name*, uct_allocated_memory_t * *mem*)

Allocate potentially registered memory. Every one of the provided allocation methods will be used, in turn, to perform the allocation, until one succeeds. Whenever the PD method is encountered, every one of the provided PDs will be used, in turn, to allocate the memory, until one succeeds, or they are exhausted. In this case the next allocation method from the initial list will be attempted.

Parameters

in	*min_length*	Minimal size to allocate. The actual size may be larger, for example because of alignment restrictions.
in	*methods*	Array of memory allocation methods to attempt.
in	*num_method*	Length of 'methods' array.
in	*pds*	Array of protection domains to attempt to allocate the memory with, for PD allocation method.
in	*num_pds*	Length of 'pds' array. May be empty, in such case 'pds' may be NULL, and PD allocation method will be skipped.
in	*name*	Name of the allocation. Used for memory statistics.
out	*mem*	In case of success, filled with information about the allocated memory. uct_allocated_memory_t.

5.12.5.7 ucs_status_t uct_mem_free (const uct_allocated_memory_t * *mem*)

Release the memory allocated by uct_mem_alloc.

Parameters

in	*mem*	Description of allocated memory, as returned from uct_mem_alloc.

5.12.5.8 ucs_status_t uct_pd_mkey_pack (uct_pd_h *pd*, uct_mem_h *memh*, void * *rkey_buffer*)

Parameters

in	*pd*	Handle to protection domain.
in	*memh*	Local key, whose remote key should be packed.
out	*rkey_buffer*	Filled with packed remote key.

Returns

Error code.

5.12.5.9 ucs_status_t uct_rkey_unpack (const void * *rkey_buffer*, uct_rkey_bundle_t * *rkey_ob*)

Parameters

in	*rkey_buffer*	Packed remote key buffer.
out	*rkey_ob*	Filled with the unpacked remote key and its type.

Returns

Error code.

5.12.5.10 ucs_status_t uct_rkey_release (const uct_rkey_bundle_t * *rkey_ob*)

Parameters

in	*rkey_ob*	Remote key to release.

5.13 UCT Active messages

Functions

- ucs_status_t uct_iface_set_am_handler (uct_iface_h iface, uint8_t id, uct_am_callback_t cb, void *arg)

  Set active message handler for the interface.
- ucs_status_t uct_iface_set_am_tracer (uct_iface_h iface, uct_am_tracer_t tracer, void *arg)

  Set active message tracer for the interface.
- UCT_INLINE_API void uct_iface_release_am_desc (void *desc)

  Release active message descriptor, which was passed to the active message callback, and owned by the callee.
- UCT_INLINE_API ucs_status_t uct_ep_am_short (uct_ep_h ep, uint8_t id, uint64_t header, const void *payload, unsigned length)
- UCT_INLINE_API ssize_t uct_ep_am_bcopy (uct_ep_h ep, uint8_t id, uct_pack_callback_t pack_cb, void *arg)
- UCT_INLINE_API ucs_status_t uct_ep_am_zcopy (uct_ep_h ep, uint8_t id, void *header, unsigned header_length, const void *payload, size_t length, uct_mem_h memh, uct_completion_t *comp)

5.13.1 Detailed Description

Defines active message functions.

5.13.2 Function Documentation

5.13.2.1 ucs_status_t uct_iface_set_am_handler (uct_iface_h *iface*, uint8_t *id*, uct_am_callback_t *cb*, void * *arg*)

Only one handler can be set of each active message ID, and setting a handler replaces the previous value. If cb == NULL, the current handler is removed.

Parameters

in	*iface*	Interface to set the active message handler for.
in	*id*	Active message id. Must be 0..UCT_AM_ID_MAX-1.
in	*cb*	Active message callback. NULL to clear.
in	*arg*	Active message argument.

5.13.2.2 ucs_status_t uct_iface_set_am_tracer (uct_iface_h *iface*, uct_am_tracer_t *tracer*, void * *arg*)

Sets a function which dumps active message debug information to a buffer, which is printed every time the an active message is sent or received, when data tracing is on. Without the trancer, only transport-level information is printed.

Parameters

in	*iface*	Interface to set the active message handler for.
in	*tracer*	Active message tracer. NULL to clear.
in	*arg*	Tracer custom argument.

5.13.2.3 UCT_INLINE_API void uct_iface_release_am_desc (void * *desc*)

Parameters

in	*desc*	Descriptor to release.

5.13.2.4 UCT_INLINE_API ucs_status_t uct_ep_am_short (uct_ep_h *ep,* uint8_t *id,* uint64_t *header,* const void * *payload,* unsigned *length*)

5.13.2.5 UCT_INLINE_API ssize_t uct_ep_am_bcopy (uct_ep_h *ep,* uint8_t *id,* uct_pack_callback_t *pack_cb,* void * *arg*)

5.13.2.6 UCT_INLINE_API ucs_status_t uct_ep_am_zcopy (uct_ep_h *ep,* uint8_t *id,* void * *header,* unsigned *header_length,* const void * *payload,* size_t *length,* uct_mem_h *memh,* uct_completion_t * *comp*)

5.14 UCT Remote memeory access operations.

Functions

- UCT_INLINE_API ucs_status_t uct_ep_put_short (uct_ep_h ep, const void *buffer, unsigned length, uint64_t remote_addr, uct_rkey_t rkey)
- UCT_INLINE_API ssize_t uct_ep_put_bcopy (uct_ep_h ep, uct_pack_callback_t pack_cb, void *arg, uint64_t remote_addr, uct_rkey_t rkey)
- UCT_INLINE_API ucs_status_t uct_ep_put_zcopy (uct_ep_h ep, const void *buffer, size_t length, uct_mem_h memh, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t *comp)
- UCT_INLINE_API ucs_status_t uct_ep_get_bcopy (uct_ep_h ep, uct_unpack_callback_t unpack_cb, void *arg, size_t length, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t *comp)
- UCT_INLINE_API ucs_status_t uct_ep_get_zcopy (uct_ep_h ep, void *buffer, size_t length, uct_mem_h memh, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t *comp)

5.14.1 Detailed Description

Defines remote memory access operairons.

5.14.2 Function Documentation

5.14.2.1 UCT_INLINE_API ucs_status_t uct_ep_put_short (uct_ep_h *ep*, const void * *buffer*, unsigned *length*, uint64_t *remote_addr*, uct_rkey_t *rkey*)

5.14.2.2 UCT_INLINE_API ssize_t uct_ep_put_bcopy (uct_ep_h *ep*, uct_pack_callback_t *pack_cb*, void * *arg*, uint64_t *remote_addr*, uct_rkey_t *rkey*)

5.14.2.3 UCT_INLINE_API ucs_status_t uct_ep_put_zcopy (uct_ep_h *ep*, const void * *buffer*, size_t *length*, uct_mem_h *memh*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uct_completion_t * *comp*)

5.14.2.4 UCT_INLINE_API ucs_status_t uct_ep_get_bcopy (uct_ep_h *ep*, uct_unpack_callback_t *unpack_cb*, void * *arg*, size_t *length*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uct_completion_t * *comp*)

5.14.2.5 UCT_INLINE_API ucs_status_t uct_ep_get_zcopy (uct_ep_h *ep*, void * *buffer*, size_t *length*, uct_mem_h *memh*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uct_completion_t * *comp*)

5.15 UCT Atomic operations.

Functions

- UCT_INLINE_API ucs_status_t uct_ep_atomic_add64 (uct_ep_h ep, uint64_t add, uint64_t remote_addr, uct_rkey_t rkey)
- UCT_INLINE_API ucs_status_t uct_ep_atomic_fadd64 (uct_ep_h ep, uint64_t add, uint64_t remote_addr, uct_rkey_t rkey, uint64_t *result, uct_completion_t *comp)
- UCT_INLINE_API ucs_status_t uct_ep_atomic_swap64 (uct_ep_h ep, uint64_t swap, uint64_t remote_addr, uct_rkey_t rkey, uint64_t *result, uct_completion_t *comp)
- UCT_INLINE_API ucs_status_t uct_ep_atomic_cswap64 (uct_ep_h ep, uint64_t compare, uint64_t swap, uint64_t remote_addr, uct_rkey_t rkey, uint64_t *result, uct_completion_t *comp)
- UCT_INLINE_API ucs_status_t uct_ep_atomic_add32 (uct_ep_h ep, uint32_t add, uint64_t remote_addr, uct_rkey_t rkey)
- UCT_INLINE_API ucs_status_t uct_ep_atomic_fadd32 (uct_ep_h ep, uint32_t add, uint64_t remote_addr, uct_rkey_t rkey, uint32_t *result, uct_completion_t *comp)
- UCT_INLINE_API ucs_status_t uct_ep_atomic_swap32 (uct_ep_h ep, uint32_t swap, uint64_t remote_addr, uct_rkey_t rkey, uint32_t *result, uct_completion_t *comp)
- UCT_INLINE_API ucs_status_t uct_ep_atomic_cswap32 (uct_ep_h ep, uint32_t compare, uint32_t swap, uint64_t remote_addr, uct_rkey_t rkey, uint32_t *result, uct_completion_t *comp)

5.15.1 Detailed Description

Defines atomic operations..

5.15.2 Function Documentation

5.15.2.1 UCT_INLINE_API ucs_status_t uct_ep_atomic_add64 (uct_ep_h *ep*, uint64_t *add*, uint64_t *remote_addr*, uct_rkey_t *rkey*)

5.15.2.2 UCT_INLINE_API ucs_status_t uct_ep_atomic_fadd64 (uct_ep_h *ep*, uint64_t *add*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uint64_t * *result*, **uct_completion_t** * *comp*)

5.15.2.3 UCT_INLINE_API ucs_status_t uct_ep_atomic_swap64 (uct_ep_h *ep*, uint64_t *swap*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uint64_t * *result*, **uct_completion_t** * *comp*)

5.15.2.4 UCT_INLINE_API ucs_status_t uct_ep_atomic_cswap64 (uct_ep_h *ep*, uint64_t *compare*, uint64_t *swap*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uint64_t * *result*, **uct_completion_t** * *comp*)

5.15.2.5 UCT_INLINE_API ucs_status_t uct_ep_atomic_add32 (uct_ep_h *ep*, uint32_t *add*, uint64_t *remote_addr*, uct_rkey_t *rkey*)

5.15.2.6 UCT_INLINE_API ucs_status_t uct_ep_atomic_fadd32 (uct_ep_h *ep*, uint32_t *add*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uint32_t * *result*, **uct_completion_t** * *comp*)

5.15.2.7 UCT_INLINE_API ucs_status_t uct_ep_atomic_swap32 (uct_ep_h *ep*, uint32_t *swap*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uint32_t * *result*, **uct_completion_t** * *comp*)

5.15.2.8 UCT_INLINE_API ucs_status_t uct_ep_atomic_cswap32 (uct_ep_h *ep*, uint32_t *compare*, uint32_t *swap*, uint64_t *remote_addr*, uct_rkey_t *rkey*, uint32_t * *result*, **uct_completion_t** * *comp*)

Chapter 6

Data Structure Documentation

6.1 ucp_generic_dt_ops Struct Reference

UCP generic data type descriptor.

Data Fields

- void ∗(∗ start_pack)(void ∗context, const void ∗buffer, size_t count)
  Start a packing request.
- void ∗(∗ start_unpack)(void ∗context, void ∗buffer, size_t count)
  Start an unpacking request.
- size_t(∗ packed_size)(void ∗state)
  Get the total size of packed data.
- size_t(∗ pack)(void ∗state, size_t offset, void ∗dest, size_t max_length)
  Pack data.
- ucs_status_t(∗ unpack)(void ∗state, size_t offset, const void ∗src, size_t count)
  Unpack data.
- void(∗ finish)(void ∗state)
  Finish packing/unpacking.

6.1.1 Detailed Description

This structure provides a generic datatype descriptor that is used for definition of application defined datatypes.

Typically, the descriptor is used for an integratoion with datatype engines implemented within MPI and SHMEM implementations.

The documentation for this struct was generated from the following file:

- ucp.h

Index