GASNet-EX API: Memory Kinds

Revision 2025.8.0

Paul H. Hargrove, Dan Bonachea

1. Introduction

This document contains the specification of the GASNet Memory Kinds feature, which is now a normative part of the GASNet-EX specification (located in GASNet-EX.txt), but currently appears in this document for historical reasons.

The majority of the API additions in this document are centered on providing multiple endpoints (`gex_EP_t`) per process, each with a potentially distinct bound memory segment (`gex_Segment_t`). Additional APIs are provided for creating memory segments for GPU device memory. Some others are intended to address needs identified by/with our current clients.

The APIs described in this document are sufficient to provide "Memory Kinds" support for RMA operations to/from device memory. Specifically, this includes NVIDIA GPUs, as demonstrated using the subset prototyped in GASNet-EX 2020.11.0, AMD GPUs as demonstrated in GASNet-EX 2021.9.0, and Intel GPUs as demonstrated in GASNet-EX 2023.9.0. However, implementations of some features and capabilities have been deferred. We are not promising that any specific capabilities in this document will be implemented in any particular release. Please consult release notes and other documentation which accompany a given source code release for the implementation status of the APIs described in this document.

Feedback may be directed to gasnet-users@lbl.gov if you feel it is suitable for open discussion with other users, or gasnet-staff@lbl.gov if you wish to reach *only* the authors of this document.

Version History

2020.6.1	Delivered with Jira P6 Activity STPM17-22. Was titled "GASNet-EX API Proposal: Multi-EP"
2020.11.0	Delivered with Jira P6 Activity STPM17-23. Partial implementations in 2020.10.0 and 2020.11.0 GASNet-EX releases.
2021.9.0	Updated to correspond to content of GASNet-EX 2021.9.0. Adds identifiers associated with HIP Memory Kinds and makes corresponding textual changes.
2022.3.0	Updated to correspond to content of GASNet-EX 2022.3.0. Adds gex_Segment_Destroy().
2022.9.0	Updated to correspond to content of GASNet-EX 2022.9.0. Revises gex_EP_BindSegment()
2024.5.0	Document renamed, was previously “GASNet-EX API Proposal: Memory Kinds”. Adds experimental support for oneAPI Level Zero memory kind.
2025.8.0	Minor editorial changes.

Acknowledgments

This work was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. GASNet-EX is developed and maintained by staff in the CLaSS Group at Lawrence Berkeley National Laboratory, funded by the Advanced Scientific Computing Research (ASCR) program of the U.S. Department of Energy's Office of Science and by the U.S. Department of Defense.

2. Overview

The remainder of this document is composed of sections corresponding to four of the GASNet-EX API subsystems, described briefly in the remainder of this section. Subsections each contain the description of one or more related APIs,, with a rationale which should explain the capability it affords to a client. The semantics described for these APIs might not be comprehensive. If there are ambiguities as a result, please bring them to our attention. We assume basic familiarity with the existing GASNet-EX APIs, as described in GASNet-EX API Specification v0.8 (or later).

Many subsections include lists of "Open Issues" and/or "Future Directions", and we are especially interested in feedback on those items. Open Issues are points which we believe would be best to resolve prior to deploying an implementation, while Future Directions describe additional capabilities or modes of operation which may be added in the future. In some cases the Future Directions have influenced the design to ease their later addition/implementation.

2.1. Endpoints

The Endpoints section describes an API for creation of additional "non-primordial" communication endpoints, along with types and APIs used to name endpoints. In GASNet-EX, additional endpoints are used to provide RMA to/from device memory and to provide independent network resources to multiple threads.

2.2. Segments

The Segments section describes APIs to create new memory segments. This is necessary to make device memory or per-thread shared heaps accessible for RMA.

2.3. Teams

The Teams section describes some API extensions for teams, including an API to construct teams containing non-primordial endpoints. Additionally, an API is described for communication directly between a given pair of endpoints, to be used in cases such as device memory where full team membership might not be necessary and/or appropriate.

2.4. Memory Kinds

The Memory Kinds section provides a rough description of a new subsystem to be added to GASNet-EX.

3. Endpoints

In GASNet-EX, an endpoint (type `gex_EP_t`) is a communications context. Every point-to-point communication call involves both a local and remote communications context. Where GASNet-1 had one implicit endpoint per process, GASNet-EX has made endpoints explicit. However, until now there has been only a single "primordial" endpoint created in each process by the call to `gex_Client_Init()`

The addition of multiple endpoints to GASNet-EX is one of the most significant capabilities described in this document, from which the need for several other new APIs arises. This section documents the APIs, constants and types needed to create and name non-primordial endpoints, and to bind segments to endpoints outside the context of `gex_Segment_Attach()`.

3.1. `gex_EP_Create()`

This API provides the mechanism for creating a non-primordial endpoint as may be required, for instance, to communicate with multiple threads per process, or with device memory.

```
int gex_EP_Create(
        gex_EP_t                    *ep_p,          // OUT
        gex_Client_t                client,
        gex_EP_Capabilities_t       capabilities,
        gex_Flags_t                 flags);
```

1. On success, returns `GASNET_OK` and sets `*ep_p` to the handle for a new endpoint.
2. Non-fatal failures return a documented error code and do not write to `*ep_p`.
3. Lack of sufficient resources to satisfy the given request will yield a return of `GASNET_ERR_RESOURCE.`
4. The new endpoint is owned by the provided `client`.
5. The new endpoint is not a member of any team, but may be added to one using an appropriate team creation API, such as the one described in subsection "gex_TM_Create()".
6. The new endpoint does not have a bound segment, but one may be bound using an appropriate API, such as the one described in subsection "gex_EP_BindSegment()".
7. The endpoint index (see subsection "gex_EP_Index_t and gex_EP_Location_t") will be non-zero and unique among all endpoints owned by the given client on the calling process.
8. Certain properties of the endpoint are controlled by the provided `capabilities` and `flags` arguments.
9. The `capabilities` argument is a bitwise-OR of one or more values from the `GEX_EP_CAPABILITY` family of constants (see subsection "GEX_EP_CAPABILITY Constants"). The new endpoint will have at least those "capabilities" requested by this argument. These include options such as whether the endpoint can be used for RMA, AM and Collective operations. These capabilities cannot be changed subsequent to creation.
10. If the value of `capabilities` requests any capability not supported by the implementation, then `GASNET_ERR_BAD_ARG` is returned.
11. A high-quality implementation will return `GASNET_ERR_BAD_ARG` if the `flags` argument does not request any capabilities.
12. The endpoint *may* be allocated scarce resources for use in accelerating certain communication operations if requested by the `GEX_FLAG_HINT_ACCEL` family of flags (see subsection "GEX_FLAG_HINT_ACCEL Constants"). If resources are NOT requested by these flags, then in a system where acceleration requires a scarce resource, none will be allocated to the new endpoint, and acceleration will not be provided.
13. Requests for acceleration resources which are unavailable due to exhaustion or non-existence are non-fatal.

Rationale

1. Independent per-thread communications resources for multi-threaded clients and communication to/from device memory are both dependent on use of a distinct endpoint for the corresponding entity. This API provides the mechanism to create these.

2. Inclusion of the `capabilities` argument and the `GEX_FLAG_HINT_ACCEL` flags are expected to help avoid allocation of resources which will not (or even cannot) be used.

Open Issues

1. Each endpoint has a small integer endpoint index associated with it, with 0 assigned to the primordial endpoint. Though it is specified as unique, we have not yet defined the algorithm for assignment of new endpoint indices. The algorithm will be deterministic and clearly documented so that for common patterns a process can predict the indices assigned in another process, avoiding the need to communicate indices. Candidate algorithms include (1) monotonically increasing (no "recycling") and (2) smallest unused (like file descriptors returned from `open()`). Both match the current documented uniqueness with respect to endpoints owned by the client on the calling process. The former extends this to be unique among all endpoints *ever* owned by the client on the calling process. Each choice has pros and cons for implementation and use. A third option is currently the most likely to be eventually deployed: similar to "smallest unused" but with an added wrinkle that the client must make a collective call to recycle the destroyed endpoints prior to their reuse (without this call the behavior is indistinguishable from the "monotonically increasing" behavior).
2. With the addition of multiple endpoints per process, the semantics currently documented for `gex_TM_TranslateJobrankToRank()` became ambiguous in the presence of teams containing more than one endpoint from the queried process. The most likely resolution is to state that it will return any rank having the requested jobrank, and probably also disclaim "stability" of the value to allow for implementations which may cache partial information.
3. Currently the ACCEL hint flags passed at EP_Create time will not perform any actual resource allocation and will be returned unmodified by gex_EP_QueryFlags(). These hints may eventually be required to be passed to subsequent object allocations to commit actual resources.

Future Directions

1. We anticipate eventually supporting different thread safety models for distinct endpoints. The plan is to use flags to select the model for an endpoint at creation, and for the selected model to be immutable. Currently each library build of GASNet-EX globally supports exactly one thread safety model.
2. There is currently no mechanism to query the *actual* capabilities of an endpoint, even though the documentation is clear this could be a superset of those requested. If there is a credible case for a client making use of capabilities they did not request, then this can be added to the `gex_EP_Info()` API under consideration.

3.2. `GEX_EP_CAPABILITY` Constants

This family of constants are to be used by `gex_EP_Create()` to specify which communications operations a new endpoint must support, where excluding unused capabilities may permit use of fewer resources.

`GEX_EP_CAPABILITY_RMA`	Ep must support `gex_RMA_*` communication operations
`GEX_EP_CAPABILITY_AM`	Ep must support `gex_AM_*` communication operations
`GEX_EP_CAPABILITY_VIS`	Ep must support `gex_VIS_*` communication operations
`GEX_EP_CAPABILITY_COLL`	Ep must support `gex_Coll_*` calls on teams containing it
`GEX_EP_CAPABILITY_AD`	Ep must support `gex_AD_*` calls on teams containing it
`GEX_EP_CAPABILITY_ALL`	bitwise-OR of all values defined in this family

1. Each identifier listed above is defined as a preprocessor macro, expanding to a constant integer expression suitable for combination via bitwise-OR to form a value of type `gex_EP_Capabilities_t`.
2. All identifiers given above must be defined independent of whether an implementation supports the corresponding capability for non-primordial endpoints.
3. When passed to an API documented as accepting a `gex_EP_Capabilities_t`, a bitwise-OR of one or more of these serves to name endpoint capabilities requested by the caller.

4. In any given release, `GEX_EP_CAPABILITY_ALL` will be the bitwise-OR of all flags defined in this family by the then-current release.
5. Not all conduits will support all capabilities in the initial implementation, but a high-quality implementation should (in the long term) support them all. Release notes or other documentation accompanying each release should clarify support.

Rationale

1. Supporting various operations on an endpoint can require non-trivial resources (buffers for AMs being an important example). These constants (or their absence) provide a mechanism for the client to guide the implementation to avoid allocation of resources which will never be used on behalf of the client.

Open Issues

1. In some families of constants, we have a requirement that the identifiers must be distinct (alias free). It has not yet been decided if such a restriction will be placed on this family of constants (would exclude ALL, of course). However, there are other open issues which would introduce intentional aliases, and thus decide this issue.
2. The underlying implementations of VIS, Coll and AD depend (in the general case) on AM and RMA calls. This creates a dependence relationship, which is already expressed in `EP_Create`'s semantic "at least those capabilities requested". It has not been determined if these relationships will be documented as implicitly satisfied and/or expressed explicitly in the values of these constants (which would conflict with the alias-free restriction mentioned in the previous item).
3. Related to the previous item, it is conceivable (but maybe not practical in the critical paths) to constrain algorithm selection in VIS, Coll and AD to AM-only options if the client has not requested the RMA capability for an endpoint. Is that implementation freedom a detail that could/should show though in the documentation?
4. There is strong consideration being given to having distinct flags for the AM categories: Short, Medium and Long. Implementation of Medium has significant buffering requirements, and Long for moderate to large payload sizes requires RMA to be efficient (potentially requiring supporting resources). Separating out Medium in particular would allow an implementation to elide allocation of buffers when only Short or Long is requested by the client, and similar for RMA resources if Long is not requested.
5. It remains to be determined whether a team of endpoints with mixed values for the AD capability is permitted to call `gex_AD_Create(),` and if so what restrictions would be placed upon the use of that AD for remote atomic communication.

Future Directions

1. The current design assumes a certain "symmetry" in which the RMA, AM, VIS and AD capabilities do not distinguish between initiators and targets. If that distinction is determined to be meaningful, then those constants might become aliases for the OR of the initiator and target capabilities. However, this would probably not make sense for Coll without disruptive changes.
2. We are considering an analogous family of hints to `gex_Client_Init()` to guide resource allocation for the primordial endpoint. However, the flexibility in that case will likely be much less than here.

3.3. `GEX_FLAG_HINT_ACCEL` Constants

This family of constants are to be used during `gex_EP_Create()` (and possibly future constructors) as a client-provided hint regarding which families of operations a new endpoint will use in ways that may benefit from the allocation of potentially scarce acceleration resources.

`GEX_FLAG_HINT_ACCEL_AD`	Ep should be allocated resources needed to accelerate atomics
`GEX_FLAG_HINT_ACCEL_COLL`	Ep should be allocated resources needed to accelerate collectives
`GEX_FLAG_HINT_ACCEL_ALL`	bitwise-OR of all values defined in this family

1. Each identifier listed above is defined as a preprocessor macro, expanding to a constant integer expression suitable for combination via bitwise-OR to form a value of type `gex_Flags_t`.
2. All identifiers given above must be defined independent of whether an implementation supports any acceleration for the corresponding API family (AD or Coll).
3. When passed to an API documented as accepting this family of flags, a bitwise-OR of one or more of these serves to name the API families for which allocation of acceleration resources is desired by the caller.
4. In any given release, `GEX_FLAG_HINT_ACCEL_ALL` will be the bitwise-OR of all flags defined in this family by the then-current release.

Rationale

1. Some networks include functional units for the acceleration of atomics and/or collective operations. Of these, some require allocation of a scarce resource (e.g. the Cray Aries Collectives Engine "CE"). This mechanism is needed to help ensure they can be allocated for use by the proper client objects.

Open Issues

1. In some flags families, we have a requirement that the identifiers must be distinct (alias free). It has not yet been decided if such a restriction will be placed on this family (would exclude ALL, of course).
2. There is a question of if/how these flags might be used with `gex_Client_Init()`. The current thinking is that something analogous but distinct would be used to control resource utilization of the primordial endpoint.

Future Directions

1. If/when presented with hardware having diverse functional units, it may become desirable to create finer-grained flags with the ones described here becoming aliases for their bitwise-OR. One hypothetical example would be separate allocation of integer and floating point execution resources, as required for reductions and/or atomics. This would conflict with any alias-free guarantee.

3.4. `gex_EP_Index_t` and `gex_EP_Location_t`

These two types allow for naming of endpoints and are used in some of the APIs appearing later in this document.

```
typedef [...] gex_EP_Index_t;
#define GEX_EP_INDEX_INVALID   ((gex_EP_Index_t)[...])

typedef struct {
     gex_Rank_t      gex_rank;
     gex_EP_Index_t  gex_ep_index;
} gex_EP_Location_t;
```

1. The type `gex_EP_Index_t` is an unsigned integer type of sufficient width to express any valid endpoint index which may be assigned at endpoint creation time.
2. The type `gex_EP_Location_t` is a pair expressing both the process on which a given endpoint lives, and its endpoint index.
3. Within the scope of any given `gex_Client_t`, the value of a `gex_EP_Location_t` containing a jobrank is a globally unique identifier for an endpoint.

Rationale:

1. Addition of these two types enables naming of endpoints other than the primordial ones, and they therefore appear in several of the later APIs.

2. While the use of `gex_Rank_t` for endpoint indices was considered, the use of a distinct type was chosen to allow it to possibly be narrower. Additionally, the distinct type marginally improves the readability of prototypes which include it.
3. Rejected alternatives to "Location" in the name of the tuple type include
    - "Coord" or "Coordinate": gave a false implication of a uniform rectangular domain
    - "Pair" and "Tuple": too generic, failing to convey any significance of the combination

Open Issues

1. None

Future Directions

1. None

3.5. `gex_EP_QueryIndex()`

This API provides the means to obtain the endpoint index of a local endpoint from its handle, as may be required to construct inputs to APIs appearing later in this document.

```
gex_EP_Index_t gex_EP_QueryIndex(gex_EP_t ep);
```

1. Returns the endpoint index of the named endpoint.

Rationale

1. This API provides the means to query the index of any local endpoint, but most importantly one which is not a member of any team (as is the case immediately following creation of a non-primordial endpoint). This index is taken as an input for one of the team APIs and may assist client code in naming endpoints.

Open Issues

1. None

Future Directions

1. We are considering later specification of a `gex_EP_Info()` API, with an interface similar to `gex_Token_Info()`. If that is added, then the description of this API may be changed to be in terms of that API.

3.6. gex_TM_TranslateRankToEP()

This API provides the means to obtain the jobrank and endpoint index of an endpoint from a `(tm,rank)` pair which names it, as may be required to construct inputs to APIs appearing later in this document.

```
gex_EP_Location_t gex_TM_TranslateRankToEP(
           gex_TM_t        tm,
           gex_Rank_t      rank,
           gex_Flags_t     flags);
```

1. Returns a `gex_EP_Location_t` describing the endpoint with the given `rank` in the given `tm.`
2. The `gex_rank` field of the result gives the jobrank of the process where the named endpoint resides.
3. The `gex_ep_index` field of the result gives the endpoint index of the named endpoint on that process.
4. The `rank` argument must valid with respect to the given tm:
   `0 <= rank < gex_TM_QuerySize(tm)`
5. The `flags` argument is reserved for future use and must currently be zero.
6. This call is permitted to communicate.
7. This call is not valid in contexts which prohibit communication, including (but not limited to) AM Handler context or when holding an HSL.

Rationale:

1. This API extends the capability of `gex_TM_TranslateRankToJobrank()` with addition of the endpoint index information to form a `gex_EP_Location_t` that provides a means to query the globally unique id for any EP which is a member of a current team. This id is taken as an input for one of the team construction APIs and may additionally assist client code in naming team members.
2. As with `gex_TM_TranslateRankToJobrank()` the specification of this call as potentially communicating allows for a future more-scalable internal representation of teams which would use distributed data structures (potentially with caching), rather than the current fully-replicated ones.
3. Inclusion of a currently unused `flags` argument permits the possible future addition of temporal locality hints to guide any underlying caching.
4. A design with a by-reference result was considered, but the by-value return was considered to be more usable and (in our estimation) had marginally better opportunities for compiler optimization of an inline implementation.

Open Issues

1. None

Future Directions

1. As with `gex_TM_TranslateRankToJobrank()`, which this API extends, it may be desirable to document "self" queries as explicit exceptions to the "may communicate" semantic.
2. Since this API is a strict superset of `gex_TM_TranslateRankToJobrank()`, we may consider deprecating that API and/or changing its specification to be in terms of this API.
3. We are considering later specification of a `gex_EP_Info()` API, with an interface similar to `gex_Token_Info()`. If that is specified, then the description of this API may be changed to be in terms of that API.
4. We are considering adding a query to convert a `gex_EP_Index_t` and `gex_Client_t` into the corresponding local `gex_EP_t` handle.

3.7. gex_EP_BindSegment()

This API provides the means to bind a segment to an endpoint, enabling more generality than is available using the existing gex_Segment_Attach().

```
int gex_EP_BindSegment(
            gex_EP_t        ep,
            gex_Segment_t   segment,
            gex_Flags_t     flags);
```

1. On success, the given segment becomes the bound segment of the endpoint ep and GASNET_OK is returned.
2. It is erroneous to bind a segment to an endpoint which already has a bound segment.
3. It is erroneous to bind GEX_SEGMENT_INVALID to an endpoint.
4. It is erroneous to bind a segment to GEX_EP_INVALID.
5. It is erroneous to bind a device memory segment to the primordial endpoint created by gex_Client_Init().
6. It is permitted to bind the same Segment to multiple endpoints.
7. The flags argument is reserved for future use and must currently be zero.

See also gex_EP_PublishBoundSegment() in the GASNet-EX.txt document for the means to make a bound segment remotely accessible.

Rationale

1. All RMA operations in GASNet-EX currently require that the remote address range lie within the bound segment of the remote endpoint named explicitly by a (tm, rank) pair or implicitly by a gex_Token_t. This API provides the means to bind a segment to an endpoint, thus enabling RMA access to its memory.
2. Regarding the return type of int. At least libfabric providers with the FI_MR_ENDPOINT bit require binding of memory regions to endpoints. This makes binding a non-trivial operation, beyond just the semantic of creating an association between two objects. Therefore, there is a desire to return non-fatal errors for cases of registration/bind failure.

Future Directions

1. When we have well-defined prerequisites for unbinding a segment from an endpoint, binding to GEX_SEGMENT_INVALID could be defined as an unbind (rather than prohibiting this) and the prohibition on binding to an endpoint with a bound segment could be relaxed to instead provide a replacement semantic.

4. Segments

In GASNet-EX, a segment (type `gex_Segment_t`) defines a range of memory and "binding" a segment to an endpoint makes that memory remotely accessible, such as via RMA and AM Long operations.

Previously, the only means to create a segment was `gex_Segment_Attach()`, which is severely restrictive. Its single-call-per-process limitation prevents creating distinct "shared heaps" for multiple threads, as well as any dynamic management of remotely-addressable storage. It also lacks any means by which to create a segment composed of anything other than host memory allocated by the implementation (client-allocated host memory and device memory being important missing alternatives). The following subsections describe APIs which begin to address the current limitations.

Open Issues

1. The future of `gex_Segment_Attach()` has not yet been decided. It might remain and be reimplemented over new APIs, resulting in a more general capability than it offers today (removing the single-call-per-process limitation in particular). Alternatively, it may become deprecated and specified in terms of calls to new APIs.

4.1. `gex_MK_t`

This subsection provides a brief overview of the type `gex_MK_t` to be used in the `gex_Segment_Create()` API which follows. This work pre-dates the Memory Kinds section (Section 6) of this document, which should also be considered.

```
typedef [...] gex_MK_t;    // An opaque scalar type

#define GEX_MK_HOST        ((gex_MK_t)[...])
```

1. A `gex_MK_t` names a "kind" of memory with specific properties which may require GASNet-EX to address, allocate, access, etc. this memory in ways which differ from how host memory is treated.
2. For this section, it is sufficient to know that
    a. There exists an opaque scalar type `gex_MK_t`, representing an object handle.
    b. `GEX_MK_HOST` is a predefined constant of this type, denoting the “kind” of regular host memory.
    c. Segments created by `gex_Segment_Attach()` always have a kind `GEX_MK_HOST`.
    d. It is possible to construct other instances of this type to describe, for instance, memory on a given GPU.

Rationale

1. GASNet-EX is expanding to allow a client to express communication directly to and from memory which is not accessible in the same manner as host memory. Memory Kinds is the term used to describe that capability.
2. Communication to and from remote memory in GASNet-EX requires a (bound) segment, and the association of a Memory Kind with each `gex_Segment_t` provides the means to convey the differing properties of the memory to the communication call.

Open Issues

1. This section should be merged into Section 6

Future Directions

1. While GPUs are the most immediate application for Memory Kinds, we see an interest in files especially on non-volatile storage which may expose RMA access.

4.2. gex_Segment_Create()

This API provides the means to create a memory segment, with more control than is provided by the existing gex_Segment_Attach().

```
typedef void *gex_Addr_t;  // Type for addresses with optional offset semantics

int gex_Segment_Create(
           gex_Segment_t *       segment_p, // OUT
           gex_Client_t          client,
           gex_Addr_t            address,
           uintptr_t             length,
           gex_MK_t              kind,
           gex_Flags_t           flags);
```

1. On success, this call creates a new gex_Segment_t, writing its handle in the location named by *segment_p, and returning GASNET_OK.
2. The kind argument specifies the Memory Kind for the new segment.
3. Providing a NULL value for address requests that GASNet-EX allocate memory of the given kind and length. This is a "GASNet-allocated" segment.
   a. For GASNet-allocated segments with GEX_MK_HOST, length must be non-zero and not larger than gasnet_getMaxLocalSegmentSize(). The implementation is permitted to round the length up to an appropriate alignment, and a subsequent gex_Segment_QuerySize() will report the actual size of the segment.
4. Providing a non-NULL value for address requests that GASNet-EX create a "client-allocated" segment to describe memory in the range [address, address+length), subject to a kind-specific interpretation of the address. This range must be mapped prior to this call and remain mapped until after segment destruction, subject to a kind-specific definition of the concept "mapped". This call does not modify the contents of this range. The memory must be addressable/accessible by means consistent with the given kind.
   a. For client-allocated segments with GEX_MK_HOST, there are no alignment restrictions on address or length, and the required accessibility includes read and write permissions (PROT_READ | PROT_WRITE). The length must be non-zero.
5. For kinds other than GEX_MK_HOST, restrictions on address and length are documented separately.
6. Any invalid combination of kind, address and length will result in a return value of GASNET_ERR_BAD_ARG and the location named by *segment_p will be unmodified.
7. The flags argument is reserved for future use and must currently be zero.

Rationale

1. Prior to this capability, the only API available to create a gex_Segment_t was gex_Segment_Attach(), which has numerous limitations relative to this API. The most obvious are (1) single-call-per-process, (2) no support for client-allocated segments, and (3) no means to specify a kind other than GEX_MK_HOST.

Open Issues

1. We have yet to determine what restrictions may be necessary to allow for client-allocated segments to overlap (partially or exactly) with each other or with GASNet-allocated segments. We plan to advertise the most permissive semantic that we can determine is safely implementable.
2. There may additionally be (possibly conduit-specific) restrictions on the attributes of client-provided GEX_MK_HOST memory, such as how it was mapped and whether it’s currently mapped into other processes.

Future Directions

1. It would be valuable for the forthcoming memory kinds APIs to include queries for properties like alignment restrictions on `address` and `length`, rather than depending on documentation alone.
2. Future use of flags could specify hints regarding conduit-specific memory registration, such as encouraging registration on-demand versus at-creation.

4.3. `gex_Segment_Destroy()`

This API provides the means to destroy a memory segment which is no longer needed and reclaim associated resources.

```
void gex_Segment_Destroy(
         gex_Segment_t          segment,
         gex_Flags_t            flags);
```

1. Destroys the segment, releasing resources allocated to it by the implementation.
    a. If the implementation has "registered" the segment memory with the underlying network API, this is reversed.
    b. If the memory segment was allocated by the implementation during `gex_Segment_Create()`, then it is freed.
2. It is erroneous to destroy the primordial segment created by `gex_Segment_Attach()`
    a. This restriction may be relaxed in the future
3. Use of `segment` following entry to this call is erroneous. In this context "use" includes, but is not limited to:
    a. Calls to `gex_Segment_*()` which pass the subject segment.
    b. Calls to `gex_EP_BindSegment()` which pass the subject segment.
    c. Calls to `gex_EP_QueryBoundSegmentNB()`, from any process, where the query argument names the subject segment.
    d. Communication calls, from any process, which involve an EP to which the subject segment is bound, and which uses addresses in the segment.
        i. Prohibited communication calls include at least RMA, Coll, and AD calls, and Long AMs.
        ii. This prohibition on communication notably includes communication which was initiated but not sufficiently completed (defined as follows), prior to calling `gex_Segment_Destroy()`.
            1. For `gex_RMA_Put*()`, local completion is required for the source segment (if any) and operation completion is required for the destination segment.
            2. For `gex_RMA_Get*()`, operation completion is required for both the source segment and the destination segment (if any).
            3. For `gex_AM_*Long*()`, local completion is required for the source segment (if any) and for the destination segment it is required that the AM handler has at least started execution.
            4. For `gex_AD_*()`, operation completion is required for the target segment.
            5. For `gex_Coll_*()`, operation completion is required for any segments which are involved as a source or destination.
        iii. It is *permitted* to issue Short and Medium AMs involving an EP to which the subject segment is bound.
4. The `flags` argument is reserved for future use and must currently be zero.

Rationale

1. Prior to this capability, no API was available to destroy a `gex_Segment_t.`

Open Issues

1. Definition and implementation of APIs to "Unbind" and "Unpublish" bound segments are needed. Once provided, their use will become preconditions for segment destruction, to replace the prohibition against communication using a bound segment. Note this call does NOT remove the binding from any endpoints to the segment destroyed by this call; hence the prohibitions against subsequent use of those endpoints in any calls involving the bound segment. The capability to unbind a segment from an endpoint (allowing rebinding to a new segment) will be provided in a future release.
2. In the case of a client-allocated device memory segment or any host memory segment, it is currently unclear if/when/how one can safely use the memory as the local address of (for instance) an RMA operation, following destruction of the segment.

Future Directions

1. Currently the destruction of a segment created by "Attach" is prohibited. This may be relaxed in the future.

5. Teams

In GASNet-EX, a team is an ordered set of endpoints, and the type `gex_TM_t` is a "team member" which represents both the ordered set as a whole and one specific member of the team. This dual role becomes important with the addition of `gex_EP_Create()` which makes it possible for a process to have multiple members in a given team.

The APIs in this section include a previously-missing API for destruction of a team, two APIs for creating new teams, and one that enables communication between endpoints *without* a team.

5.1. `gex_TM_Destroy`

This API provides the means to destroy a team which is no longer needed and reclaim associated resources.

```
int gex_TM_Destroy(
          gex_TM_t            tm,
          gex_Memvec_t        *scratch_p,     // OUT
          gex_Flags_t         flags);

#define GEX_FLAG_GLOBALLY_QUIESCED          [...]
#define GEX_FLAG_SCRATCH_SEG_OFFSET         [...]
```

1. This call must be called collectively over members of the team named by `tm`.
2. Destroys the team, releasing resources allocated to it by the implementation.
3. It is erroneous to destroy the primordial team.
4. Use of `tm` after return from this call is erroneous.
5. Does not destroy the endpoint associated with `tm`.
6. The identifier `GEX_FLAG_GLOBALLY_QUIESCED` is a preprocessor macro expanding to a constant integer expression suitable for use as a value of type `gex_Flags_t.`
7. For the purpose of this API, a `tm` has been "locally quiesced" only when *all* of the following are true with respect to calls initiated on the local process:
    a. No calls taking this `tm` as an argument are executing concurrently on other threads.
    b. All collective operations using this `tm` are complete (client has synced their `gex_Event_t`'s).
    c. Any `gex_AD_t` objects created using this `tm` have been destroyed.
8. By default, the `tm` must be locally quiesced on each caller before it may invoke this API. However, if `GEX_FLAG_GLOBALLY_QUIESCED` is passed in flags, then the caller is additionally asserting that the `tm` has been quiesced on *all* callers (globally) prior to *any* caller invoking this API.
9. The presence/absence of `GEX_FLAG_GLOBALLY_QUIESCED` in `flags` must be single-valued.

10. Regardless of the presence/absence of `GEX_FLAG_GLOBALLY_QUIESCED` in `flags`, this call is permitted but not required to incur barrier synchronization across `tm`.
11. The `scratch_p` argument may be `NULL`. If `non-NULL` then if-and-only-if the collective scratch space used by the team was provided by the client, then its location is written to the location named by the `scratch_p` argument.
12. If a value is written to `*scratch_p` then return value is non-zero. Otherwise, zero is returned.
13. If `GEX_FLAG_SCRATCH_SEG_OFFSET` is set in `flags`, then the `gex_addr` field of `*scratch_p` argument (if non-NULL) is assigned the byte offset into the bound segment of the endpoint associated with `tm`. Otherwise, this field is assigned the virtual address.
14. The presence/absence of `GEX_FLAG_SCRATCH_SEG_OFFSET` in `flags` need not be single-valued, and need not match the value used at team construction.
15. Any cleanup action with respect to ClientData associated with the `tm` is the client's responsibility.

Rationale

1. It is our intention to provide destructors for all object types allocatable through the GASNet-EX APIs. This is just one of the destructors currently missing.
2. The specification of `GEX_FLAG_GLOBALLY_QUIESCED` is intended to make the synchronization optional in order to remove unnecessary barriers. For instance, given a scenario in which a client has a "row team" and a "column team" with a common parent, it would be sufficient to locally quiesce both teams, followed by a barrier over their common parent, followed by making back-to-back calls to destroy these row and column teams with this flag.
3. The definition of "locally quiesced" intentionally excludes completion of non-blocking point-to-point operations using `tm` at their initiation. This is because the semantics of such operations do not have any semantic connection to the `tm` used to initiate them *other* than at the time of initiation. Since the endpoint outlives the destruction of any given team which may contain it, there are no issues anticipated with completion of in-flight operations or hidden communication such as for AM flow-control.
4. The optional `scratch_p` argument is intended to assist the client in reclaiming use of the space it may have granted to the collectives implementation when the team was created, without creating a requirement for the client to track something GASNet-EX already tracks.
5. The choice to provide the `scratch_p` argument rather than a stand-alone query API is based on the principle that the client should not be doing anything with that memory between the creation and destruction of the `tm`.
6. The disclaimer of barrier synchronization exists to permit implementations where no such synchronization is required. For instance, in the case of multiple `tm` per process as members of the same team (such as one per thread), it is conceivable that all but the last thread to enter the call could decrement a reference counter and return immediately (not waiting for remote members to enter the collective).

Open Issues

1. It is possible that the list of conditions for local quiescence is incomplete or otherwise flawed.
2. There may be subtle implications for the implementation of point-to-point operations (AMs in particular) due to allowing a `tm` to be destroyed without draining such operations. However, the use of a `gex_EP_Location_t` or equivalent in place of the `tm` and/or `rank` given at initiation should be sufficient to resolve these. This assertion should be confirmed prior to adoption of the definition of local quiescence.
3. We are considering a non-blocking version of this API, to be provided in lieu of this one.

Future Directions

1. If we add NBI collective operations, then the definition of "complete" in the definition of "locally quiesced" will need to be adjusted (or made to reference a factored definition added to the Glossary?)

5.2. gex_TM_Dup()

This API provides the means to duplicate an existing team more efficiently than is possible with the existing `gex_TM_Split()`.

```
size_t gex_TM_Dup(
           gex_TM_t        *new_tm_p,
           gex_TM_t        orig_tm,
           gex_Addr_t      scratch_addr,
           size_t          scratch_len,
           gex_Flags_t     flags);

#define GEX_FLAG_TM_SYMMETRIC_SCRATCH [...]
#define GEX_FLAG_TM_LOCAL_SCRATCH     [...]
#define GEX_FLAG_TM_NO_SCRATCH        [...]
```

1. This call is collective over members of `orig_tm`.
2. When `flags` contains one of the `GEX_FLAG_TM_SCRATCH_SIZE` family of flags (whose presence must be single-valued), this API behaves in the same manner as documented for `gex_TM_Split()`, returning a minimum or recommended size for the collective scratch space and the arguments `new_tm_p`, `addr` and `len` are ignored. Otherwise, the remaining semantics apply.
3. This call creates a new team, storing the corresponding `gex_TM_t` at the location named by `new_tm_p`.
4. The new team has the same membership (ordered set of endpoints) as `orig_tm`.
5. The new team is created with a new collective scratch space, which may be optionally provided from the bound segment of the corresponding endpoint via the `scratch_addr` and `scratch_len` arguments.
    a. As with `gex_TM_Split()`, this "option" is actually *required* in the current implementation.
6. The minimum valid `scratch_len` is the value returned from a `GEX_FLAG_TM_SCRATCH_SIZE_MIN` query using `gex_TM_Dup()` and the same `orig_tm`.
7. If `GEX_FLAG_SCRATCH_SEG_OFFSET` is set in `flags`, then the `scratch_addr` argument is interpreted as a byte offset into the bound segment of the endpoints associated with `orig_tm`. Otherwise, this argument is a virtual address in the same bound segment.
8. The presence/absence of `GEX_FLAG_SCRATCH_SEG_OFFSET` in `flags` must be single-valued.
9. The range described by the `scratch_addr` and `scratch_len` arguments must fall entirely within the bound segment of the endpoint associated with `tm`, must not be modified by the client between entering this call and return from destruction of the team, and must not overlap any other collective scratch space. In particular, one cannot reuse the scratch space of `orig_tm`.
10. The `scratch_len` argument must be single-valued (same on all callers).
11. Exactly one of `GEX_FLAG_TM_SYMMETRIC_SCRATCH`, `GEX_FLAG_TM_LOCAL_SCRATCH` or `GEX_FLAG_TM_NO_SCRATCH` currently must be present in `flags`, and the selection must be single-valued.
12. Presence of `GEX_FLAG_TM_SYMMETRIC_SCRATCH` in `flags` is an assertion by the caller that `scratch_addr` is single-valued (potentially allowing the implementation to elide both communication and storage). Otherwise `GEX_FLAG_TM_LOCAL_SCRATCH` allows each caller to pass a different `scratch_addr`. Presence of `GEX_FLAG_TM_NO_SCRATCH` means the arguments `scratch_len` and `scratch_addr` are ignored, no scratch space is assigned, and collectives over this team are prohibited (this may be relaxed in the future).
13. This call is guaranteed to provide sufficient synchronization that the caller may begin using `*new_tm_p` immediately following return. The implementation is permitted but not required to include barrier synchronization across `orig_tm`, which may or may not be necessary to provide this guarantee.

Rationale

1. It may be desirable to create a team with the same membership as an existing team, but with its collective ordering requirement being independent from that of the original. This provides the means to do so more efficiently than via `gex_TM_Split()` or any other planned API for team construction.

Open Issues

1. Scratch allocation honors new flags, in part to support efficient symmetric allocation. We are considering (backwards compatible) updates to the semantics to `gex_TM_Split()` to honor these as well.
2. It is uncertain if all scratch allocation modes will be implemented in the initial release of this API.
3. We are considering a non-blocking version of this API, to be provided in lieu of this one.

Future Directions

1. It is imagined that `flags` might be used to request alteration of some boolean properties of the new team, relative to `orig_tm`. However, no candidates have been identified.
2. It is expected that some future release will eliminate the need for clients to manage collectives scratch space. At that time a new flag may be added to request that the implementation perform scratch allocation.
3. This API is not sufficient to duplicate a team that includes endpoints which lack corresponding host CPU threads to perform the collective call. A distinct API for such a case is under consideration.

5.3. gex_TM_Create()

This API provides the means for construction of one or more teams per call (at most one per caller) with greater generality than the existing `gex_TM_Split()`, including the ability to incorporate endpoints not yet in any team.

```
size_t gex_TM_Create(
                gex_TM_t                *new_tms,       // OUT
                size_t                  num_new_tms,
                gex_TM_t                parent_tm,
                gex_EP_Location_t       *args,          // IN
                size_t                  numargs,
                size_t                  scratch_length  // single-valued
                gex_Addr_t              *scratch_addrs, // IN
                gex_Flags_t             flags);

#define GEX_FLAG_SCRATCH_SEG_OFFSET             [...]

#define GEX_FLAG_TM_SYMMETRIC_SCRATCH           [...]
#define GEX_FLAG_TM_LOCAL_SCRATCH               [...]
#define GEX_FLAG_TM_GLOBAL_SCRATCH              [...]
#define GEX_FLAG_TM_NO_SCRATCH                  [...]
```

1. Collective over `parent_tm`, which must contain at least one member for every process named in the `args[]` of any caller.
2. When `flags` contains one of the `GEX_FLAG_TM_SCRATCH_SIZE` family of query flags (whose presence must be single-valued over the *parent* team), this API behaves analogously to that documented for `gex_TM_Split()`: returning a minimum or recommended size for the collective scratch space of the team which would otherwise be created for this caller based on the arguments `num_new_tms`, `numargs` and `args[]`, and ignoring the arguments `new_tms`, `scratch_length` and `scratch_addrs`. Otherwise, the remaining semantics apply.
3. Creates either zero (for `numargs == 0`) teams or one team (for `numargs > 0`) per caller.

4. When passing `numargs == 0`, the caller must provide a value for `flags` which is consistent with any "single-valued over the parent team" constraints. However, all arguments other than `parent_tm`, `numargs` and `flags` are ignored (and subsequent semantics constraining the ignored arguments do not apply).
5. The `args[]` must contain `numargs > 0` distinct elements naming every endpoint to become a member of the team the caller is creating, in rank order.
6. The `gex_rank` field of `args[]` specifies a process by jobrank if `GEX_FLAG_RANK_IS_JOBRANK` is present in `flags`, otherwise the `gex_rank` field is a rank relative to `parent_tm` and the process is the one associated with that team member.
7. The presence/absence of `GEX_FLAG_RANK_IS_JOBRANK` in `flags` must be single-valued *over the output team*.
8. The value of `numargs` and content of `args[]` must be single-valued *over the output team*.
9. Taken over all callers, any two non-empty `args[]` arrays must either be identical (constructing the same team) or name a disjoint set of endpoints (creating a distinct, non-overlapping team). A `numargs == 0` caller is always disjoint.
10. The immediately preceding restriction applies not only to callers in distinct processes, but also to the case of multiple callers per process (due to multiple members in `parent_team`).
11. The value of `numargs` and content of `args[]` are not required to be single-valued over `parent_tm`, allowing for creation of multiple teams per collective call (but at most one per caller).
12. The endpoint corresponding to `parent_tm` is not required to be among the entries in `args[]`.
13. The value of `num_new_tms` must equal the number of local endpoints named in `args[]`, and the location named by `new_tms[]` must have sufficient space to receive `num_new_tms` entries.
14. On output, the array `new_tms[]` will be populated with a distinct `gex_TM_t` for each local member in the newly created team, in their respective rank order. No entries will be populated or skipped/reserved for non-local members.
15. Each new team is created with a collective scratch space, which may be optionally provided from the bound segment of the corresponding endpoint via the `scratch_length` and `scratch_addrs` arguments.
    a. As with `gex_TM_Split()`, this "option" is actually *required* in the current implementation.
16. The argument `scratch_length` must be single-valued *over the output team*.
17. If `GEX_FLAG_SCRATCH_SEG_OFFSET` is set in `flags`, then the value(s) in `scratch_addrs[]` are byte offsets into the respective bound segments of the endpoints being joined into the new team. Otherwise, these values are virtual addresses in those same bound segments.
18. The presence/absence of `GEX_FLAG_SCRATCH_SEG_OFFSET` in `flags` must be single-valued *over the output team*.
19. The length and contents of `scratch_addrs[]` depends on which of the following mutually-exclusive values are included in the value of `flags` (there is currently no default).
    a. `GEX_FLAG_TM_SYMMETRIC_SCRATCH`
       There is exactly one entry in `scratch_addrs[]` and it provides the address or offset used for all members of the output team.
    b. `GEX_FLAG_TM_LOCAL_SCRATCH`
       The array `scratch_offsets[]` has length `num_new_tms` and provides the addresses or offsets for each local member in the output team.
    c. `GEX_FLAG_TM_GLOBAL_SCRATCH`
       The array `scratch_offsets[]` has length `num_args` and provides the addresses or offsets for every member in the output team.
    d. `GEX_FLAG_TM_NO_SCRATCH`
       The arguments `scratch_length` and `scratch_offsets[]` are ignored.
       No scratch space is assigned and collectives over this team are prohibited (this may be relaxed in the future).
20. Scratch space, if any, must always reside in a bound segment with kind `GEX_MK_HOST`. Consequently, calls to this team constructor that include endpoints bound to segments with other memory kinds (such as devices) currently MUST pass `GEX_FLAG_TM_NO_SCRATCH`. This restriction might be relaxed in the future.

21. The mutually exclusive choice of `GEX_FLAG_TM_{SYMMETRIC,LOCAL,GLOBAL,NO}_SCRATCH` in `flags` must be single-valued *over the output team*.
22. This call is guaranteed to provide sufficient synchronization that the caller may begin using the new handles in `new_tms[]` immediately following return. The implementation is permitted but not required to include barrier synchronization, which may or may not be necessary to provide this guarantee.

Rationale

1. Allows construction of `upcxx::local_team` without the off-node communication which is required by the current construction via `gex_TM_Split()`.
2. Allows an endpoint-per-pthread client (such as a hypothetical improvement to the pthreads-as-UPC-threads mode of the Berkeley UPC Runtime) to construct a team including the primordial endpoints together with ones created via `gex_EP_Create()` calls to yield a large team with a rank for every pthread.
3. We are considering a non-blocking version of this API, to be provided in lieu of this one.

Open Issues

1. It is uncertain if all scratch allocation modes will be implemented in the initial release of this API.
2. Undecided if there will be a default among the multiple scratch allocation modes.

Future Directions

1. This API requires the caller to instantiate a full enumeration of the membership of teams it creates, which could require substantial memory for something like the EP-per-pthread case. Therefore, we are also seeking to design team creation APIs with more scalable inputs. One would be a generalization of Split's color-matching semantic to allow inputs which can include endpoints outside the calling team. Another might be a team constructor that exploits possibly single-valued properties like ep_idx to reduce duplication in the metadata.
2. It is expected that some future release will eliminate the need for clients to manage collectives scratch space. At that time a new flag may be added to request that the implementation perform scratch allocation.

5.4. `gex_TM_Pair()`

This API provides the means to locally construct a value which can be passed as the `tm` argument to point-to-point communication calls in lieu of a collectively created team, allowing communication between endpoints which might not be members of any common team.

```
gex_TM_t gex_TM_Pair(
            gex_EP_t             local_ep,
            gex_EP_Index_t       remote_ep_index);
```

1. Returns a value of type `gex_TM_t` representing an ad hoc “TM-pair” consisting of the given `local_ep` in the calling process and the endpoint with index `remote_ep_index` in the process with a jobrank given by the `rank` argument passed along with this `gex_TM_t` in a point-to-point communication call.
2. `gex_TM_Pair` is a lightweight, non-communicating utility call (likely an inline function or macro).
3. The result is a TM-pair value which may be stored, reused or discarded, and has no corresponding free or release call (although it only remains valid for use while the referenced endpoints exist).
4. Two TM-pair values will compare equal if and only if they were created by calls to `gex_TM_Pair()` with the same arguments, and will never compare equal to a `gex_TM_t` created by other means.
5. The result *is not* a valid argument to any API with a prefix of `gex_TM_`, `gex_AD_` or `gex_Coll_`, nor to any API documented as collective over the argument (regardless of prefix).
6. The result *is* valid for use in AM payload limit queries: `gex_AM_Max{Request,Reply}{Medium,Long}()`
7. The result *is* valid for use in the bound segment query: `gex_Segment_QueryBound()`

8. The result *is* valid for use in point-to-point communication calls in the `gex_RMA_*()`, `gex_VIS_*()` and `gex_AM_*()` families when used in a manner similar to what is shown in the following examples.

Here is an example call to `gex_RMA_GetNBI()` to read from the endpoint with index `rem_idx` on the process with the given `jobrank`, and initiated using the local endpoint `loc_ep`.

```
gex_RMA_GetNBI(gex_TM_pair(loc_ep, rem_idx), dest, jobrank, src, nbytes, flags);
```

If there is a need to communicate between a local endpoint `ep0` and the remote endpoints with index 1 in several processes, then a pattern like the following could be used to reuse the value returned by `gex_TM_Pair()` for these calls.

```
gex_TM_t tm_pair_01 = gex_TM_pair(ep0, 1);
for (int i = 0; i < num_peers; ++i)
     gex_RMA_GetNBI(tm_pair_01, dest[i], jobrank[i], src[i], nbytes, flags);
```

Rationale

1. With the exception of AM Replies, all GASNet-EX point-to-point communications APIs name both the local and remote endpoints using a pair of arguments of type `gex_TM_t` and `gex_Rank_t`. However, a `gex_TM_t` corresponding to a team has associated semantics that are not well-suited to inclusion of endpoints which lack corresponding host CPU threads to perform collective calls. This API allows for communication to/from the memory in segments bound to any endpoint in the job without the need to create a team.
2. An alternative approach would be to double the width of the `RMA`, `VIS` and `AM` API families to add variants of all existing calls which take a local `gex_EP_t` and remote `gex_EP_Location_t`. However, that could require client code passing `(tm, rank)` pairs to double their implementation's width as well. This approach allows the `(tm, rank)` pair to remain the sole canonical way to pass a point-to-point communication's contexts.

Open Issues

1. Current expectations are that use of values generated by this API will have a very small performance penalty relative to the use of the primordial team and possibly *better* performance than use of a non-primordial team due to the elimination of a rank-to-jobrank translation in the critical path (assuming the caller isn't making one). Should the documentation reflect this?

Future Directions

1. None

5.5. Strengthened semantics for GEX_FLAG_TM_SCRATCH_SIZE_* queries

The semantics of the GEX_FLAG_TM_SCRATCH_SIZE_* family of query flags currently specify that “the return value is not guaranteed to be single-valued”.

This semantic is being strengthened to guarantee that a given query always returns a resulting size that is single-valued over the new team that would be created, if any. Otherwise, the result is zero.

Rationale

1. There is no reason to suspect a client would request any value for a collective scratch size other than the values returned from these queries (or possibly bounded by these values). Multiple APIs have been specified (or will be in the future) which require single-valued sizes be specified. The strengthened semantic eliminates the implication that the client may need to perform a reduction over the return values to meet such a single-valued restriction.

Open Issues

1. It is likely that the semantics of `gex_TM_Split()` will also be strengthened to require its `scratch_size` argument to be single-valued over the output team. While this is technically a "breaking change", we think it unlikely that any current client would pass a non single-valued argument since, as alluded to in the Rationale, clients are believed to be using only values based on the return from these (now single-valued) queries.

Future Directions

1. None

6. Memory Kinds

This final section is an informal description of the ideas and APIs for Memory Kinds.
While it lacks Rationale, Open Issues and Future Directions, the API in this section are fully formed.

6.1 Overview

A variable of type `gex_MK_t` is intended to mean something roughly like "UVA memory on device 0".

A second CUDA device (or other type of device) would have a distinct `gex_MK_t`.

A `gex_Segment_t` has an associated address range and kind, the latter expressing the address-independent settings, parameters, etc.
So, a "kind" variable is functionally analogous to an instance of a C++ class, having instance-specific data members which hold things like "device 0" and class-specific member functions which the conduit uses to perform a set of operations needed for communications.

`gex_MK_t` is effectively a handle to an opaque object, with one predefined instance handle (corresponding to host memory) and an object factory function to create a new instance corresponding to memory on a specific device.
Of course use of C makes the implementation somewhat different from what the OO design suggests.

6.2 Type `gex_MK_t` type and constants

The following are defined by including `gasnetex.h`

```
// All functions taking (or returning) a memory kind use this type:
typedef [...] gex_MK_t;    // An opaque scalar type

// There exist two predefined values of type gex_MK_t:
#define GEX_MK_INVALID     ((gex_MK_t)0) // will never alias a valid kind
#define GEX_MK_HOST        ((gex_MK_t)[...])
```

6.3 `gex_MK_Create()` : Creating an instance of type gex_MK_t

Each kind of device has a corresponding "class" which corresponds to the access mechanisms and API used to access that kind of device. Creating an instance (variable of type `gex_MK_t`) requires calling `gex_MK_Create()` with a value to specify the class of memory, and the class-specific arguments to identify the specified device (and to open, connect, etc. as may be appropriate) . For a GPU, these arguments are expected to be a device identifier or something semantically similar. For an imagined class for files, class-specific arguments might be a file descriptor or pathname. To handle this polymorphism in C, a "tagged union" is used.

The following are defined in `gasnet_mk.h` (*not* `gasnetex.h`):

```
// Creation of an instance of gex_MK_t must name the "class"
// gex_MK_Class_t is an enum naming available "classes" of memory kinds.
// It includes at least the following values (in unspecified order):
typedef enum {
    GEX_MK_CLASS_HOST,      // "normal" memory (eg GEX_MK_HOST)
    GEX_MK_CLASS_CUDA_UVA,  // CUDA UVA memory     [since 2020.11.0]
    GEX_MK_CLASS_HIP        // HIP device memory   [since 2021.9.0]
    GEX_MK_CLASS_ZE,        // oneAPI Level Zero device memory [EXPERIMENTAL]
    ...
} gex_MK_Class_t;
```

```
// The gex_MK_Create_args_t struct is passed to gex_MK_Create to create a
// per-device instance of a memory kind of the given class. It is a
// struct containing a union and an enum to indicate which member has been populated.
// Each union member is a struct named based on the enum value (lowercase, drop "mk_").
// All types in here are basic types, possibly type-erased/indirected versions of types
// provided in device headers.
// The struct includes at least the following members (in unspecified order):
typedef struct {
    uint64_t        gex_flags; // Reserved.  Must be 0 currently,
                               // and is guaranteed to be the first field
    gex_MK_Class_t  gex_class;
    union {
        struct {// CUDA UVA memory [since 2020.11.0]
            int                  gex_CUdevice;
        }                    gex_class_cuda_uva;
        struct {// HIP device memory [since 2021.9.0]
            int                  gex_hipDevice;
        }                    gex_class_hip;
        struct {// oneAPI Level Zero device memory [EXPERIMENTAL]
            void*                 gex_zeDevice;
            void*                 gex_zeContext;
            uint32_t              gex_zeMemoryOrdinal;
        }                    gex_class_ze;
        ...
    }                    gex_args;
    ...
} gex_MK_Create_args_t;

// Constructor for gex_MK_t
// This is a non-collective call
int gex_MK_Create(
        gex_MK_t                         *kind_p,    // OUT
        gex_Client_t                     client,
        const gex_MK_Create_args_t       *args,      // IN
        gex_Flags_t                      flags       // Reserved.  Must be 0 currently.
    );

// Destructor for gex_MK_t (non collective)
void gex_MK_Destroy(
        gex_MK_t kind,
        gex_Flags_t flags       // Reserved.  Must be 0 currently.
    );
```

We reserve the right to add additional fields to the `gex_MK_Create_args_t` struct. Consequently, client code must avoid assuming any particular field order, and must ensure zero-initialization of any portions of a `gex_MK_Create_args_t` that they do not explicitly initialize.

We have given consideration to per-class "convenience wrappers" which would internally construct the required `gex_MK_Create_args_t` from scalar arguments. This may be particularly valuable if later classes require non-trivial "marshaling". However, an ideal implementation of such wrappers would utilize the proper types specific to the device API (such as CUDA). Since it's not acceptable to include headers such as `cuda.h` from `gasnet_mk.h`, some other "delivery mechanism" would be needed.

A set of `GASNET_HAVE_MK_CLASS_*` identifiers have been documented, and inclusion of `gasnetex.h` will leave each one either undefined or defined to 1. This includes one per supported class, plus one additional identifier `GASNET_HAVE_MK_CLASS_MULTIPLE`, to be defined if and only if support has been compiled in for any memory kinds other than host memory.

6.4 Memory Kinds Example

Putting this together, a client might contain code along the following (contrived) lines:

```
#if !GASNET_HAVE_MK_CLASS_CUDA_UVA
#  error Missing GASNet-EX support for CUDA UVA memory kinds
#endif

// Create memory kinds for N GPUs, with indices [0 .. N)
gex_MK_Create_args_t args = { .gex_flags = 0,
                              .gex_class = GEX_MK_CLASS_CUDA_UVA };
gex_MK_t mk_array[N];
for (int id = 0; id < N; ++id) {
  args.gex_args.gex_class_cuda_uva.gex_CUdevice = id;
  int rc = gex_MK_Create(&mk_array[i], myClient, &args, 0);
  assert(rc == GASNET_OK);
}
```

Subsequent code could then pass `mk_array[?]` as the kind argument to `gex_Segment_Create()` calls, either allowing GASNet-EX to allocate device memory, or using the address and length of some block of memory obtained by the client using `cudaMalloc()`.

6.5 CUDA_UVA Specific Notes

For `MK_CLASS_CUDA_UVA`, we support only devices with the `unifiedAddressing` property, as the "`_UVA`" in the MK class name is intended to convey. A high-quality implementation would be expected to verify this in `gex_MK_Create()`.

The CUDA context associated with the operations GASNet performs using a kind of this class is the device’s primary context, as recorded at the time of the call to `gex_MK_Create()`.

6.6 Host vs Device Addresses

Segment creation specifies an address range in terms of device addresses only. This is motivated by non-UVA devices we expect to eventually support for which there is no host address or no fixed host address. IF we encounter a device which needs more than 64 bits to represent its addresses, then we'd probably introduce `gex_Segement_Create_"byref"()`, or similar, to handle this.

In communication (eg RMA), our *eventual* intent is to support both device addresses and "offset-based" addressing (gex_Addr_t). The current implementation provides only device address support (consistent with what is passed to `gex_Segment_Create()` and returned by `gex_Segment_QueryAddr()`). If (as mentioned above) we encounter a device needing more than 64 bits to represent its address, we still anticipate that implementation of offset-based addressing will be sufficient for communication, since a segment length must be representable in a parameter of type `size_t`.

7. Conclusion

As described at the start of this document, this represents a continuing effort to define APIs relevant to the introduction of "Memory Kinds" to GASNet-EX. This has been, by far, the most significant change planned or implemented in GASNet to date, which motivated the existence of this auxiliary document as a means to start and track this process.

Any future evolution of this document is also likely to include updates to Open Issues and Future Directions as those are resolved. However, the extensions documented here will eventually be merged into a combined GASNet-EX main-line specification.

For GASNet-EX downloads, documentation, publications, etc.: gasnet.lbl.gov
To report bugs: gasnet-bugs.lbl.gov
To reach the community: gasnet-users@lbl.gov
To reach the authors of this document: gasnet-staff@lbl.gov

Thanks for your interest in GASNet-EX!