Title

GASNet-EX Specification Collection, Revision 2024.5.0

Authors

Bonachea, Dan

Hargrove, Paul H

Publication Date

2024-05-23

DOI

10.25344/S4160B

GASNet-EX Specification Collection, Revision 2024.5.0

Dan Bonachea, Paul H. Hargrove
Lawrence Berkeley National Laboratory, USA
https://gasnet.lbl.gov
gasnet-staff@lbl.gov

Lawrence Berkeley National Laboratory Technical Report (LBNL-2001595)
doi:10.25344/S4160B

May 23, 2024

Abstract

GASNet-EX is a portable, open-source, high-performance communication library designed to efficiently support the networking requirements of PGAS runtime systems and other alternative models in emerging exascale systems. It provides network-independent, high-performance communication primitives including Remote Memory Access (RMA) and Active Messages (AM). GASNet-EX is an evolution of the popular GASNet communication system, building upon over 20 years of lessons learned, and the primary goals are high performance, interface portability, and expressiveness. The library has been used to implement parallel programming models and libraries such as UPC, UPC++, Fortran coarrays, Legion, Chapel, and many others.

This anthology collects together the four separate volumes that currently comprise the GASNet-EX specification, as of the 2024.5.0 release of GASNet-EX.

Copyright

Legal Disclaimer

Acknowledgments

GASNet is developed and maintained by staff in the Computer Languages & Systems Software (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.

Introduction

The GASNet-EX interface is currently specified by four separate documents, covering disjoint portions of the API:

A. **GASNet-EX API description, v0.18**:

   Specifies most of the modern GASNet-EX interfaces. This is currently written as a "diff" against the GASNet-1 APIs described in documents below.

B. **GASNet-EX API: Memory Kinds, Revision 2024.5.0**:

   Specifies the GASNet-EX Memory Kinds feature, which support RMA operations to/from device memory, such as the memory on-board GPU accelerators.

C. **GASNet-1 Specification, v1.8.1** (Tech Rpt. LBNL-2001064):

   Specifies the original GASNet-1 API, which remains fully supported as a subset of GASNet-EX.

D. **GASNet-1 Extensions for Non-contiguous RMA** (Tech Rpt. LBNL-56495 v2.0):

   Portions of this document describe the non-contiguous RMA extension to GASNet-1, which remains supported and also serves as the specification basis for the Vector, Indexed, Strided RMA operations in GASNet-EX.

These four documents currently comprise the GASNet-EX specification, and are available for separate download from the GASNet home page: https://gasnet.lbl.gov

As a convenience to clients, this anthology collects together these four separate volumes, as of the 2024.5.0 release of GASNet-EX. The library software release available from https://gasnet.lbl.gov implements all of the interfaces described in these documents, except where otherwise noted.

WORK IN PROGRESS The GASNet-EX interface continues to evolve over time.
Feedback and questions should be directed to: gasnet-staff@lbl.gov

GASNet-EX API description, v0.18

```
///////////////////////////////
// GASNet-EX API Description //
///////////////////////////////

// This is *not* a final normative document.
// This is "beta documentation" for a work-in-progress.
//
// This document assumes a reasonable degree of familiarity with the current
// (aka GASNet-1) specification: https://gasnet.lbl.gov/dist/docs/gasnet.pdf
//
// Except where otherwise noted, all definitions in this document
// are provided by gasnetex.h.
//
// See implementation_defined.md, a sibling to this file, for additional
// features provided by the implementation hosted at https://gasnet.lbl.gov,
// which are not required by this specification.

// Document Conventions
//
// This document includes the annotation [UNIMPLEMENTED] in several places
// where we feel we have a suitable design ready for consideration, but
// have yet to provide a complete and/or correct implementation.
//
// This document includes the annotation [EXPERIMENTAL] in several places
// where we feel we have a suitable design and an implementation which is
// sufficiently complete to be used.  However, based on feedback received
// from early use, the design may change in non-trivial ways (to the degree
// that client code may need to change).

// Public Header Files
//
// The following public headers may be included in any order:
//    gasnetex.h       : GASNet-EX API + Tools
//    gasnet.h         : GASNet-1 API (also includes gasnetex.h)
//    gasnet_fwd.h     : See below
//    gasnet_ratomic.h : Remote Atomics API
//    gasnet_vis.h     : Non-contiguous RMA (VIS) API
//    gasnet_coll.h    : Collectives API
//
// No other `*.h` files appearing in the source distribution or install tree
// are intended for direct inclusion by client code.

// The `gasnet_fwd.h` header
//
// The gasnet_fwd.h provides a subset of `gasnetex.h`, for convenience of
// clients who want a minimal header that is safe to include in their own
// client's code.  It defines only preprocessor `#define` constants and
// stand-alone `typedefs` for simple data types.  All preprocessor
// identifiers are in the `GEX_` or `_GEX_` namespace.
//
// `gasnet_fwd.h` contains at least the following type definitions:
//
//    `gex_Rank_t`
//    `gex_EP_Index_t`
//    `gex_Addr_t`
//    `gex_Event_t`
//    `gex_Flags_t`
//    `gex_DT_t`
//    `gex_OP_t`
//    `gex_EP_Capabilities_t`
//
// and the following constants:
//
//    `GEX_RANK_INVALID`
```

```
//    `GEX_EVENT_*`
//    `GEX_FLAG_*`
//    `GEX_DT_*`
//    `GEX_OP_*`
//    `GEX_EP_CAPABILITY_*`
// The semantics of these identifiers are described in subsequent sections.

//
// Specification and release versioning:
//

// Release version tuple
//
// This takes the form YEAR.MONTH.PATCH in GASNet-EX releases,
// (where YY, MM and PP below represent the appropriate digits)
// providing a clear distinction from GASNet-1 with MAJOR==1.
#define GASNET_RELEASE_VERSION_MAJOR 20YY
#define GASNET_RELEASE_VERSION_MINOR MM
#define GASNET_RELEASE_VERSION_PATCH PP

// Major and Minor versions of the GASNet-EX specification.
//
// This is currently a version number for *this* document.
#define GEX_SPEC_VERSION_MAJOR 0
#define GEX_SPEC_VERSION_MINOR 18

// Major and Minor versions of the GASNet-1 specification.
//
// This is the version to which the gasnet_* APIs adhere
// and which prevails for all matters which this document
// does not (yet) address.
#define GASNET_SPEC_VERSION_MAJOR 1
#define GASNET_SPEC_VERSION_MINOR 8

// Major and Minor versions of the GASNet-Tools specification.
//
// This is the spec version for the GASNet Tools
//
// For GASNet Tools API documentation see README-tools, located
// at the top level of the GASNet-EX sources.
#define GASNETT_SPEC_VERSION_MAJOR 1
#define GASNETT_SPEC_VERSION_MINOR 20

//
// Relationship to GASNet-1 APIs:
//
// This release should continue to support nearly all GASNet-1 APIs,
// provided the client #includes <gasnet.h>, which implements the
// GASNet-1 APIs in terms of the new GASNet-EX interfaces.
//
// Most gasnet_ APIs have gex_ counterparts that are either interoperable,
// or which provide a superset of the most closely-related gasnet_ APIs.
//
// Where a gex_/GEX_ identifier is interoperable or synonymous with
// a gasnet_/GASNET_ identifier, that is noted below.
//

// Hybrid/transitional client support:
//
// All clients must initialize GASNet using *either* the legacy
// gasnet_init()/gasnet_attach() calls *or* the new gex_Client_Init() call
// described in a subsequent section.
//
// Clients who are incrementally adopting GASNet-EX may have a period of time
// when they are using deprecated GASNet-1 calls (see below).  A process using
// such deprecated calls must enable legacy support, which is done implicitly
```

```
// when a process calls gasnet_init() or explicitly when a caller passes the
// GEX_FLAG_USES_GASNET1 flag to gex_Client_Init().
//
// The following functions from GASNet-1 have been deprecated in favor of new
// GASNet-EX equivalents. These `gasnet_`-prefixed deprecated functions shall
// only be invoked if legacy support has been enabled.
// + gasnet_AMRequest*()
// + gasnet_get*(), including bulk, non-bulk, value-based and VIS
// + gasnet_put*(), including bulk, non-bulk, value-based and VIS
// + gasnet_memset*()
// NOTE: This list is subject to expansion as new GASNet-EX APIs are
// implemented which displace less-capable GASNet-1 APIs.
//
// Note that gasnet_init()/gasnet_attach() may only be called once per process,
// and currently only one client per process can use the deprecated GASNet-1
// functions listed above.
//
#define GEX_FLAG_USES_GASNET1 ((gex_Flags_t)???)

// The following API allows jobs that enable legacy support (as described
// immediately above) to access the key GASNet-EX objects created explicitly by
// gex_Client_Init(), or implicitly by gasnet_init()/gasnet_attach().  The types
// and usage of these objects are described below.
//
// This call is defined in gasnet.h (not gasnetex.h).
//
// The arguments are all pointers to locations for outputs, each of which
// may be NULL if the caller does not need a particular value.
//
//     client_p: receives the gex_Client_t
//   endpoint_p: receives the gex_EP_t
//         tm_p: receives the gex_TM_t
//    segment_p: receives the gex_Segment_t, if any
//
extern void gasnet_QueryGexObjects(gex_Client_t      *client_p,
                                   gex_EP_t          *endpoint_p,
                                   gex_TM_t          *tm_p,
                                   gex_Segment_t     *segment_p);

// Calls from restricted context
//
// The only GASNet functions which may be called within AM handler context,
// or while holding a GASNet handler-safe lock are as follows:
//
// gasnet_mynode(), gasnet_nodes(), gasnet_hsl_*(), gasnet_exit(),
// gasnet_QueryGexObjects(), gex_System_QueryNbrhdInfo(), gex_System_QueryHostInfo(),
// gex_System_QueryMyPosition(), gex_System_QueryJob{Rank,Size}(), gex_HSL_*(),
// gex_*_{Set,Query}CData(), gex_{Client,Segment,EP,TM,AD}_Query*(), gex_TM_Pair(),
// gex_AM_Max*(), gex_AM_LUB*(), gex_Token_Max*(), gex_Token_Info(), gasnet_AMGetMsgSource(),
// gex_System_GetVerboseErrors(), gex_System_SetVerboseErrors(),
// gex_System_QueryMaxThreads(), gex_System_QueryHiddenAMConcurrencyLevel()
//
// The following are conditionally permitted in handler context, the condition being the
// caller must be within an AMRequest handler and not holding a handler-safe lock:
//
// gasnet_AMReply*(), gex_AM_Reply*() gex_AM_{Prepare,Commit}Reply*(), gex_AM_SrcDesc*()
//
// The following are conditionally permitted in handler context, the condition
// being that the 'flags' argument must include GEX_FLAG_IMMEDIATE:
//
// gex_EP_QueryBoundSegmentNB()
//
// All other functions are prohibited to be called from a thread within the
// dynamic context of an AM handler, or while holding a handler-safe lock.
// This prohibition notably prohibits all communication initiation (aside from Reply
// injection from a Request handler), explicit polling and test/wait operations on handles/events.
```

```
//
// Glossary:
// The following terms will be used with specific meanings in this document.
//

// "Collective Call"
//
// Several APIs in this specification are described as being "collective
// calls".  All collective calls are collective with respect to a specific
// ordered set of participants, which is usually specified by an argument
// naming a team (discussed later in detail).  The designation of a call
// as collective over a given team means:
//   + For every given team that exists in an execution of the program, all
//     collective calls made over that team are initiated "in the same order"
//     by all team members -- otherwise behavior is undefined.
//   + Here "in the same order" means that for every member of a given team,
//     the calls over that team and their arguments are "compatible" across all
//     members at every point in their respective sequence of collective calls
//     over the team.
//   + The definition of "compatible" as used here may vary slightly as
//     defined individually for each call.  However, in the absence of per-call
//     documentation to the contrary the following rules apply:
//     - The function called must be the same, or from a related group of calls
//       explicitly documented as mutually compatible.
//     - Any arguments documented as "single-valued" must be identical across
//       all callers.
//     - Any additional argument compatibility constraints documented for a
//       given call must be satisfied.
//
// In addition to the requirement on compatibility of collective calls over
// any given team, all collective calls over *distinct* teams must be ordered
// such that no deadlock would occur if all such calls were replaced by
// blocking barriers.  A formal specification of this constraint will appear
// in a future revision of this document.

// "Single-valued"
//
// This term is used to designate an argument to a collective call as one that
// must have the same value on all callers participating in the collective, or
// on a well-defined subset of callers.
//
// In the case of 'flags' arguments, this term may be applied in a qualified
// form as "partially single-valued" when the constraint applies only to some
// bits (with freedom to differ in the remaining bits).

//
// Basic types:
//

// Rank

// The type gex_Rank_t is used for a position within, or size of, an ordered
// set (such as a team).
// Guaranteed to be an unsigned integer type
// This type is interoperable with gasnet_node_t
typedef [some unsigned integer type] gex_Rank_t;

// Pre-defined constant used to indicate "not a rank".
// Use may have different semantics in various contexts.
// Guaranteed to be larger than any valid rank.
// However, a specific value is NOT defined by specification.
// In particular, might NOT be equal to GASNET_MAXNODES
#define GEX_RANK_INVALID ((gex_Rank_t)???)
```

```
// "Job rank":
// In a non-resilient build this will be the same as the rank in the team
// constructed by gex_Client_Init() and will be identical across clients.
// This is semantically equivalent to gasnet_mynode().
//
// Semantics in a resilient build will be defined in a later release.
gex_Rank_t gex_System_QueryJobRank(void);

// "Job size":
// In a non-resilient build this will be the same as the size in the team
// constructed by gex_Client_Init() and will be identical across clients.
// This is semantically equivalent to gasnet_nodes().
//
// Semantics in a resilient build will be defined in a later release.
gex_Rank_t gex_System_QueryJobSize(void);

// Utility

// By default, certain non-fatal error returns in GASNet-EX will print messages
// to the console.  This behavior can be queried and set with the following.

// Returns non-zero if console messages are enabled for certain non-fatal errors.
int gex_System_GetVerboseErrors();

// Enable (1) or disable (0) console messages for certain non-fatal errors.
// Values other than 0 and 1 are currently reserved.
void  gex_System_SetVerboseErrors(int enable);

// Client Threads

// The maximum number of live client threads permitted to enter GASNet.
//
// In threaded (non-SEQ) builds of GASNet, client threads making GASNet calls may
// implicitly become associated with thread-specific state managed by the GASNet
// library. When such a thread exits, a thread destructor registered by the library
// cleans up any associated thread-specific state. The library is permitted to limit
// the number of live client threads that may concurrently be implicitly associated
// with GASNet-managed state.
//
// + The limit is per-process and the value returned is for the calling process.
// + The limit is process-wide, independent of gex_Client_t.
// + Threads internal to GASNet, if any, do not count against this limit.
// + Client threads which have not yet entered GASNet do not count against
//   this limit.
// + Client threads which exit after having entered GASNet cease to count
//   against this limit.
//
// In a SEQ build of GASNet, this query always returns 1.
uint64_t gex_System_QueryMaxThreads(void);

// Events

// An "Event" is an opaque scalar type, representing a handle
// to an asynchronous event that will be generated by a pending operation.
// Events are a generalization of GASNet-1 handles, in that
// a single non-blocking operation may expose several events
// associated with its progress (eg local and remote completion).
// Initiation of a event-based (NB-suffix) non-blocking operation will
// usually generate one root event (representing the completion
// of the entire operation), and zero or more leaf events
// (representing completion of intermediate steps).
// Root events must eventually be synchronized by passing them
// to a Wait or successful Test function, which recycles all the
// events (root and leaf) associated with the operation in question.
// Leaf events may optionally be synchronized before that point.
// This type is interoperable with gasnet_handle_t
```

```
// - Sync operation: test/wait with one/all/some flavors
//   + Success consumes the event
typedef ... gex_Event_t;

// Pre-defined output values of type gex_Event_t
// - GEX_EVENT_INVALID
//   + result for already-completed operation
//   + synonymous with GASNET_INVALID_HANDLE
//   + guaranteed to be zero
// - GEX_EVENT_NO_OP
//   + result for a failed communication attempt (eg immediate-mode
//     injection that encountered backpressure)
//   + guaranteed to be non-zero
//   + Erroneous to pass this value to test/wait operations
#define GEX_EVENT_INVALID      ((gex_Event_t)0)
#define GEX_EVENT_NO_OP        ((gex_Event_t)???)

// Pre-defined input values of type gex_Event_t*
// These are passed to communication injection operations
// in place of a pointer to an actual gex_Event_t for certain leaf
// events, to forgo an independent leaf event and instead request
// specific predefined behavior:
// - GEX_EVENT_NOW
//   + Pass to require completion of the leaf event before returning
//     from the initiation call
// - GEX_EVENT_DEFER
//   + Pass to allow deferring completion of the leaf event to as late
//     as completion of the root event
// - GEX_EVENT_GROUP
//   + Pass to NBI initiation calls to allow client to use NBI-based
//     calls to detect event completion (or to use an explicit event
//     returned/generated by gex_NBI_EndAccessRegion()).
#define GEX_EVENT_NOW    ((gex_Event_t*)???)
#define GEX_EVENT_DEFER  ((gex_Event_t*)???)
#define GEX_EVENT_GROUP  ((gex_Event_t*)???)


// Integer flag type used to pass hints/assertions/modifiers to various functions
// Flag value bits to a given API are guaranteed to be disjoint, although
// flag values used for unrelated functions might share bits.
typedef [some integer type] gex_Flags_t;

//
// Flags for point-to-point communication initiation
//
//
// IMMEDIATE
//
// This flag indicates that GASNet-EX *may* return without initiating
// any communication if the conduit could determine that it would
// need to block temporarily to obtain the necessary resources.  In
// this case calls with return type 'gex_Event_t' return
// GEX_EVENT_NO_OP while those with return type 'int' will
// return non-zero.
//
// Additionally, calls with this flag are not required to make any
// progress toward recovery of the "necessary resources".  Therefore,
// clients should not assume that repeated calls with this flag will
// eventually succeed.  In the presence of multiple threads, it is
// even possible that calls with this flag may never succeed due to
// racing for resources.
//
#define GEX_FLAG_IMMEDIATE ((gex_Flags_t)???)
```

```
// LC_COPY_{YES,NO}
//
// This mutually-exclusive pair of flags *may* override GASNet-EX's
// choice of whether or not to make a copy of a source payload (of a
// non-blocking Put or AM) for the purpose of accelerating local
// completion.  In the absence of these flags the conduit-specific
// logic will apply.
//
// NOTE: these need more thought w.r.t. the implementation and
// specification
#define GEX_FLAG_LC_COPY_YES ((gex_Flags_t)???) [UNIMPLEMENTED]
#define GEX_FLAG_LC_COPY_NO  ((gex_Flags_t)???) [UNIMPLEMENTED]
//
// PEER_NEVER_{SELF,NBRHD}
// [Since spec v0.14]
//
// These flags, passed to a supporting communication initiation API, assert to
// the GASNet-EX library that the '(tm,rank)' tuple (or equivalent) does NOT
// name an endpoint in certain processes.
//
// Use of these flags *may* allow the library to omit its own checks for the
// asserted condition.  However, to have this desired impact, the compiler must
// be capable of statically deciding their presence in the 'flags' argument.
// Therefore, non-trivial logic to determine whether or not to pass either of
// these flags is strongly discouraged.  The intended use case for these flags
// is in situations where the asserted property is known without additional
// logic at the specific call site.
//
// Providing an assertion which is untrue will yield undefined results (though
// in a high-quality implementation, a debug build will report the discrepancy).
//
// SELF  - Asserts that the "remote" peer in a communication call is not
//         an endpoint in the initiating process.
// NBRHD - Asserts that the "remote" peer in a communication call is not
//         an endpoint in any process in the initiator's nbrhd.
//
// Use of GEX_FLAG_PEER_NEVER_NBRHD implies GEX_FLAG_PEER_NEVER_SELF.
// However, their use is not mutually exclusive.
//
// Currently these flags are valid to pass to:
//   gex_RMA_*()
//
// A future revision may permit these flags for additional communication
// injection calls.
#define GEX_FLAG_PEER_NEVER_SELF    ((gex_Flags_t)???)
#define GEX_FLAG_PEER_NEVER_NBRHD   ((gex_Flags_t)???)
//
// AD_MY_{RANK,NBRHD}
//
// This mutually-exclusive pair of flags each assert a locality property of
// the target of a remote atomic operation, and are described in detail in
// the "Remote Atomic Operations" section.
//
#define GEX_FLAG_AD_MY_RANK           ((gex_Flags_t)???)
#define GEX_FLAG_AD_MY_NBRHD          ((gex_Flags_t)???)
//
// AD_FAVOR_{MY_RANK,MY_NBRHD,REMOTE}
//
// This mutually-exclusive group of flags each request that gex_AD_Create()
// bias its algorithm selection to favor calls with a given locality property
// for the target locations, and are described in detail in the "Remote Atomic
// Operations" section.
//
#define GEX_FLAG_AD_FAVOR_MY_RANK           ((gex_Flags_t)???)
#define GEX_FLAG_AD_FAVOR_MY_NBRHD          ((gex_Flags_t)???)
#define GEX_FLAG_AD_FAVOR_REMOTE            ((gex_Flags_t)???)
```

```
// AD_{ACQ,REL}
//
// This pair of flags requests memory fencing behaviors for remote atomic
// operations, and are described in detail in the "Remote Atomic Operations"
// section.  It is permitted to include zero, one, or both of these flags
// when calling gex_AD_Op*().
//
#define GEX_FLAG_AD_ACQ           ((gex_Flags_t)???)
#define GEX_FLAG_AD_REL           ((gex_Flags_t)???)
//
// RANK_IS_JOBRANK
//
// This flag indicates, to those calls explicitly documented as accepting it,
// that the 'rank' (or equivalent argument) is a jobrank rather than a rank
// within the normal associated team.
//
// Currently this flags is accepted by:
//   gex_AD_Op*()
//
#define GEX_FLAG_RANK_IS_JOBRANK  ((gex_Flags_t)???)
//
// AM_PREPARE_LEAST_{CLIENT,ALLOC}
//
// This pair of mutually exclusive flags modify the behavior of the
// gex_AM_Max{Request,Reply}{Medium,Long}() queries to request the largest
// legal 'least_payload' argument to the corresponding "gex_AM_Prepare*()"
// rather than the default behavior (returning the largest legal 'nbytes'
// argument to the corresponding "gex_AM_{Request,Reply}*()").
//
// CLIENT - query largest 'least_payload' for a Prepare call with a
//          client-provided buffer (non-NULL 'client_buf' argument).
// ALLOC  - query largest 'least_payload' for a Prepare call with a
//          GASNet-allocated buffer (NULL 'client_buf' argument).
//
// Legal (and meaningful) in gex_AM_Max{Request,Reply}{Medium,Long}() calls.
// Ignored in gex_AM_Prepare{Request,Reply}{Medium,Long}() calls.
// Invalid in gex_AM_{Request,Reply}{Medium,Long}*() calls.
//
#define GEX_FLAG_AM_PREPARE_LEAST_CLIENT   ((gex_Flags_t)???)
#define GEX_FLAG_AM_PREPARE_LEAST_ALLOC    ((gex_Flags_t)???)

// SEGMENT DISPOSITION
//
// The following family of flags assert the segment disposition of
// address ranges provided to communication initiation operations.
//
// The segment disposition flags come in two varieties:
//
// SELF - describes the segment disposition of addresses associated
//        with local buffers and the initiating endpoint (ie the EP
//        which is usually implicitly named by a gex_TM_t argument).
//        Eg in a Put operation this variety describes source locations,
//        and in a Get this variety describes destination locations.
//
// PEER - describes the segment disposition of buffers associated
//        with (potentially) remote memory and the peer endpoint(s)
//        (the EPs usually explicitly named by gex_Rank_t arguments).
//        Eg in a Put operation this variety describes destination locations,
//        and in a Get this variety describes source locations.
//
// The following flags are mutually exclusive within each variety -
// a given operation may specify at most one SELF flag and one PEER flag.
// Unless otherwise noted, the default behavior for each variety in the
// absence of an explicitly provided flag corresponds to:
// + When the local EP is unbound or bound to host memory:
//     GEX_FLAG_SELF_SEG_UNKNOWN, GEX_FLAG_PEER_SEG_BOUND
```

```
// + When the local EP is bound to a device segment:
//     GEX_FLAG_SELF_SEG_BOUND, GEX_FLAG_PEER_SEG_BOUND
// These are backwards-compatible with GASNet-1 segment behavior (where
// there is no support for device memory).
// NOTE: the flags below are currently [UNIMPLEMENTED], and consequently
// these defaults are also the only supported settings for all APIs.
//
// Each explicit flag has a distinct bit pattern.
// Unless otherwise noted, the caller is responsible for ensuring the
// assertions expressed by these flags to a given call remain true for
// the entire period of time that the described address sequences are "active"
// with respect to the operation requested by the call. The definition of
// "active" varies based on call type, but generally extends from entry to
// the call accepting the assertions until completion is signalled for
// all described address ranges.
//
// {SELF,PEER}_SEG_UNKNOWN
//
// These flag bits indicate that the corresponding address range(s)
// are not known by the caller to reside within current GASNet-EX segments.
// Example 1: the address ranges are known to lie partially or entirely
//   outside any segments in the process hosting the respective endpoint(s).
// Example 2: the caller lacks information about the segment disposition
//   of the address ranges, and passes this flag to reflect a lack of
//   such assertions and request maximally permissive behavior
//   (potentially incurring a performance cost).
#define GEX_FLAG_SELF_SEG_UNKNOWN ((gex_Flags_t)???) [UNIMPLEMENTED]
#define GEX_FLAG_PEER_SEG_UNKNOWN ((gex_Flags_t)???) [UNIMPLEMENTED]
//
// {SELF,PEER}_SEG_SOME
//
// These flag bits assert that the corresponding address range(s)
// are contained entirely within the union of current GASNet-EX segments
// created by any client in the process hosting the respective endpoint.
#define GEX_FLAG_SELF_SEG_SOME    ((gex_Flags_t)???) [UNIMPLEMENTED]
#define GEX_FLAG_PEER_SEG_SOME    ((gex_Flags_t)???) [UNIMPLEMENTED]
//
// {SELF,PEER}_SEG_BOUND
//
// These flag bits assert that the corresponding address range(s)
// are contained entirely within the segment bound to the respective endpoint.
// Implies that the respective endpoint has a bound segment.
#define GEX_FLAG_SELF_SEG_BOUND   ((gex_Flags_t)???) [UNIMPLEMENTED]
#define GEX_FLAG_PEER_SEG_BOUND   ((gex_Flags_t)???) [UNIMPLEMENTED]
//
// {SELF,PEER}_SEG_OFFSET
//
// These flag bits indicate that the corresponding address argument(s)
// are byte *offsets* relative to the bound segment base address.
// Implies that the respective endpoint has a bound segment, and
// that the specified range(s) are contained entirely within that segment.
#define GEX_FLAG_SELF_SEG_OFFSET  ((gex_Flags_t)???) [UNIMPLEMENTED]
#define GEX_FLAG_PEER_SEG_OFFSET  ((gex_Flags_t)???) [UNIMPLEMENTED]

// COLLECTIVE SCRATCH ALLOCATION
//
// The following family of flags control the interpretation of address ranges
// provided to team construction APIs to describe collective scratch spaces.
//
// TM_{GLOBAL,LOCAL,SYMMETRIC,NO}_SCRATCH
//   This mutually-exclusive group indicates the number and meaning of
//   a gex_Addr_t specified to certain team construction APIs.
//   [Since spec v0.9:]
//      GLOBAL: gex_Addr_t per member of the output team
//       LOCAL: gex_Addr_t per local member of the output team
//   SYMMETRIC: single gex_Addr_t used for all members of the output team
```

```
//   [Since spec v0.11:]
//          NO: no gex_Addr_t (and no scratch space is allocated).
//
#define GEX_FLAG_TM_GLOBAL_SCRATCH     ((gex_Flags_t)???) // gex_TM_Create only
#define GEX_FLAG_TM_LOCAL_SCRATCH      ((gex_Flags_t)???) // gex_TM_Create only
#define GEX_FLAG_TM_SYMMETRIC_SCRATCH  ((gex_Flags_t)???) // gex_TM_Create only
#define GEX_FLAG_TM_NO_SCRATCH         ((gex_Flags_t)???) // gex_TM_Create and gex_TM_Split
//
// SCRATCH_SEG_OFFSET
//
// This flag bit indicates that the corresponding gex_Addr_t argument(s)
// are byte *offsets* relative to the bound segment base address.
// Implies that the respective endpoint has a bound segment, and
// that the specified range(s) are contained entirely within that segment.
#define GEX_FLAG_SCRATCH_SEG_OFFSET  ((gex_Flags_t)???) [UNIMPLEMENTED]

// GEX_FLAG_GLOBALLY_QUIESCED
// [Since spec v0.10]
//
// This flag bit indicates to the corresponding object destructor call that
// the client has satisfied the call's documented global quiescence criteria.
// This permits, but does not require, the implementation to elide
// synchronization which might otherwise be required.
#define GEX_FLAG_GLOBALLY_QUIESCED   ((gex_Flags_t)???)

// A "token" is an opaque scalar type
// This type is interoperable with gasnet_token_t
typedef ... gex_Token_t;

// Handler index - a fixed-width integer type, used to name an AM handler
// This type is interoperable with gasnet_handler_t
typedef uint8_t gex_AM_Index_t;

// Handler argument - a fixed-width integer type, used for client-defined handler arguments
// This type is interoperable with gasnet_handlerarg_t
typedef int32_t gex_AM_Arg_t;

// Handler function pointer type
typedef ... gex_AM_Fn_t;

// Widest scalar and width
// This type is interoperable with gasnet_register_value_t
typedef [some unsigned integer type] gex_RMA_Value_t;

// Preprocess-time constant size of gex_RMA_Value_t
// Synonymous with SIZEOF_GASNET_REGISTER_VALUE_T
#define SIZEOF_GEX_RMA_VALUE_T ...

// gex_Addr_t
// Type which is suitable to hold both addresses and offsets.
//
// This is always an alias for `void*`, but is given a distinct type to make
// prototypes self-documenting with respect to arguments which may (with the
// proper flags) be interpreted alternatively as addresses or offsets.
typedef void* gex_Addr_t;

// Memvec
// A "memvec" describes a tuple of memory address and length
// gex_Memvec_t is guaranteed to have the same in-memory representation as gasnet_memvec_t;
// these two struct types name their fields differently so they are technically
// incompatible as far as the compiler is concerned -- it *is* safe to type-pun
// pointers to them with explicit casts.
typedef struct {
  void  *gex_addr;  // [EXPERIMENTAL]: will eventually have type gex_Addr_t
  size_t gex_len;
} gex_Memvec_t;
```

```
// gex_EP_t is an opaque scalar handle to an Endpoint (EP),
// a local representative of an isolated communication context
typedef ... gex_EP_t;

// Pre-defined value of type gex_EP_t
// This zero value is guaranteed never to alias a valid endpoint
#define GEX_EP_INVALID ((gex_EP_t)0)

// gex_EP_Index_t is an unsigned integer type.
//
// Every EP within a given gex_Client_t can be uniquely identified by
// the jobrank of a process and an endpoint index.
// The primordial endpoint, created by gex_Client_Init(), will always have
// an index of 0.  At this time, there are no other guarantees regarding how
// endpoint indices are allocated/assigned.
typedef ... gex_EP_Index_t;

// Max supported number of endpoints per client in each process.
// This is an optimistic compile-time constant which cannot account
// for limitations due to scarcity of network resources and/or memory.
// The value is implementation-defined and may be conduit-specific.
#define GASNET_MAXEPS ...

// gex_EP_Location_t is a (rank, ep_index) tuple.
typedef struct {
  gex_Rank_t     gex_rank;
  gex_EP_Index_t gex_ep_index;
} gex_EP_Location_t;

// gex_Client_t is an opaque scalar handle to a Client,
// an instance of the client interface to the GASNet library
typedef ... gex_Client_t;

// Pre-defined value of type gex_Client_t
// This zero value is guaranteed never to alias a valid client
#define GEX_CLIENT_INVALID ((gex_Client_t)0)

// gex_Segment_t is an opaque scalar handle to a Segment,
// a local client-declared memory range for use in communication
typedef ... gex_Segment_t;

// Pre-defined value of type gex_Segment_t
// Used, for instance, to indicate no bound segment
#define GEX_SEGMENT_INVALID ((gex_Segment_t)0)

// In general, gex_TM_t is an opaque scalar handle to a Team Member,
// a collective communication context used for remote endpoint naming.
// There is also a less-general form, known as a "TM-pair" which carries only
// sufficient information for naming an endpoint in point-to-point communication
// or queries.
// In collective calls, an argument of type gex_TM_t specifies both an ordered
// set of Endpoints (local or remote), and a local gex_EP_t, a local
// representative of that team.  Use of a TM-pair is prohibited in such calls.
// In point-to-point calls the local and remote gex_EP_t are named by a tuple
// consisting of one argument of type gex_TM_t and another of type gex_Rank_t
// together.  Similarly, several queries take a '(tm,rank)' tuple to name an
// endpoint.  Use of a TM-pair or a fully general gex_TM_t are both permitted
// in these non-collective calls.

typedef ... gex_TM_t;

// Pre-defined value of type gex_TM_t
// This zero value will never to alias a valid gex_TM_t (including TM-pairs)
#define GEX_TM_INVALID ((gex_TM_t)0)
```

```
//
// Client-Data (CData)
//
// The major opaque object types in GASNet-EX provide the means for the client
// to set and retrieve one void* field of client-specific data for each object
// instance, which is NULL for newly created objects.

void  gex_Client_SetCData(gex_Client_t client, const void *val);
void* gex_Client_QueryCData(gex_Client_t client);
void  gex_Segment_SetCData(gex_Segment_t seg, const void *val);
void* gex_Segment_QueryCData(gex_Segment_t seg);
void  gex_TM_SetCData(gex_TM_t tm, const void *val);
void* gex_TM_QueryCData(gex_TM_t tm);
void  gex_EP_SetCData(gex_EP_t ep, const void *val);
void* gex_EP_QueryCData(gex_EP_t ep);

//
// Operations on gex_Client_t
//

// Query flags passed to gex_Client_Init()
gex_Flags_t  gex_Client_QueryFlags(gex_Client_t client);

// Query client name passed to gex_Client_Init()
const char * gex_Client_QueryName(gex_Client_t client);

// Initialize the client
// This is a collective call over all processes comprising this GASNet job.
// Currently supports only one call per job.
// * clientName must reference a string that uniquely identifies this client
//   within the process, and must match the pattern: [A-Z][A-Z0-9_]+
//   The contents of the string referenced by clientName must be single-valued.
//   In future release this string will be used in such contexts as error messages
//   and naming of environment variables to control per-client aspects of GASNet.
// * argc/argv are optional references to the command-line arguments received by main().
//   The caller is permitted to pass NULL for both arguments, if this is done
//   by all callers.  However, providing pointers to the values received in
//   main() may improve portability or supplementary services.
// * client_p, ep_t and tm_p are OUT parameters that receive references to the
//   newly-created Client, the primordial (thread-safe) Endpoint for this process/client,
//   and the primordial Team (which contains all the primordial Endpoints, one
//   for every process in this job).
// * flags control the creation of the primordial objects. Supported flags:
//   + GEX_FLAG_USES_GASNET1 - created client requests the use of GASNet-1 APIs
//     (defined in gasnet.h). Only permitted for use in one client per process.
//   + GEX_FLAG_DEFER_THREADS [Since spec v0.18] [EXPERIMENTAL]
//     Client requests that the conduit defer launching of internal progress
//     threads.  See `gex_System_QueryProgressThreads()` for information on the
//     means for the client to control launch of the progress threads, if any,
//     after return from `gex_Client_Init()`.
//   This is a single-valued parameter.
//
// There is an implicit barrier synchronization prior to return from this call
// to ensure that creation of communications resources has completed on all
// callers prior to return on any caller.
extern int gex_Client_Init(
            gex_Client_t           *client_p,
            gex_EP_t               *ep_p,
            gex_TM_t               *tm_p,
            const char             *clientName,
            int                    *argc,
            char                   ***argv,
            gex_Flags_t            flags);
```

```
// Operations on gex_Segment_t
//
// NOTE: *currently* gex_Segment_Attach() is the only way to create a segment
// suitable for use as the bound segment of a primordial endpoint (one created
// by gex_Client_Init). In particular, the current release does not *yet*
// support use of the APIs gex_Segment_Create(), gex_EP_BindSegment() and
// gex_EP_PublishBoundSegment() as an alternative to Attach. However, support
// for that usage may appear in a future release.
//
// See also in [PROPOSED] section:
//    gex_Segment_Create()
//    gex_EP_BindSegment()
//

// Query owning client
gex_Client_t gex_Segment_QueryClient(gex_Segment_t seg);

// Query flags passed when segment was created
// There are no segment flags defined in the current release.
gex_Flags_t  gex_Segment_QueryFlags(gex_Segment_t seg);

// Query base address of a segment
// For segments created using gex_Create_Segment() with a 'kind' not equal to
// GEX_MK_HOST, the return value is a device address.
// Otherwise, it is a host address.
void *       gex_Segment_QueryAddr(gex_Segment_t seg);

// Query length of a segment
uintptr_t    gex_Segment_QuerySize(gex_Segment_t seg);


// Collective allocation and creation of Segments
// Analogous to gasnet_attach (but see below)
//
// This is a collective call over the team named by the 'tm' argument that
// allocates and binds a local GASNet segment on each caller.
//
// There is an implicit barrier synchronization prior to return from this call
// to ensure that the creation and binding of a segment has completed on all
// callers prior to return on any caller.
//
// segment_p: An OUT parameter that receives the newly created gex_Segment_t.
//            This is not a single-valued parameter.
// tm:        The call is collective over this team.
// size:      Size of the local segment to allocate and bind to the local
//            Endpoint represented by tm.  The value must be a non-zero
//            multiple of GASNET_PAGESIZE, not larger than
//            gasnet_getMaxLocalSegmentSize().
//            This is not a single-valued parameter.
//
// The current release allows up to one call per process.
//
// The current release requires that 'tm' be the team created by
// gex_Client_Init().
//
// NOTE: gex_Segment_Attach() does not provide alignment of segments across ranks.
// Use of --enable-aligned-segments at configure time and definition of
// GASNET_ALIGNED_SEGMENTS at compile time are relevant only to the legacy
// gasnet_attach() interface.
//
// NOTE: In the current release, when the legacy GASNET_SEGMENT_EVERYTHING
// configuration is in effect, the following additional rules apply:
//  - In this mode, the primordial endpoint is implicitly bound to the entire
//    virtual address space by gex_Client_Init(), and this call has no semantic
//    effect (aside from a barrier synchronization).
//  - If the optional call is made, the size argument is ignored, and the resulting
```

```
//    gex_Segment_t in `*segment_p` shall be GEX_SEGMENT_INVALID.

extern int gex_Segment_Attach(
                gex_Segment_t          *segment_p,
                gex_TM_t               tm,
                uintptr_t              size);


//
// Operations on gex_TM_t
//


// Query owning client
gex_Client_t gex_TM_QueryClient(gex_TM_t tm);

// Query corresponding endpoint
gex_EP_t     gex_TM_QueryEP(gex_TM_t tm);

// Query flags passed when tm was created
gex_Flags_t  gex_TM_QueryFlags(gex_TM_t tm);

// Query rank of team member, and size of team
gex_Rank_t   gex_TM_QueryRank(gex_TM_t tm);
gex_Rank_t   gex_TM_QuerySize(gex_TM_t tm);


// Split a Team into zero or more disjoint teams
//
// This is a collective call over the team named by the 'parent_tm' argument
// that creates zero or more new teams.  While this call is collective, the
// arguments are NOT required to be single-valued over the parent team, except
// as noted for certain bits in 'flags'.  However, the value of 'scratch_size'
// (if applicable) must be collective over callers passing the same 'color'.
//
// + When passing any of the GEX_FLAG_TM_SCRATCH_SIZE_* family of flags, this
//   call is a collective query to determine the minimum or recommended value
//   for the 'scratch_size' argument, based on the other parameters (excluding
//   scratch_addr and scratch_len).  No teams are created and nothing is
//   written into `new_tm_p`.  Otherwise, this call creates zero or more teams
//   as described in the remaining semantics.
// + When not operating as a query, the return value is currently undefined.
// + Callers passing NULL for 'new_tm_p' do not participate in team creation.
//   This assists in following the collective call requirement without the
//   need to create teams that are not needed by the client.
// + For callers passing non-NULL for 'new_tm_p', this call creates a new team
//   consisting of the associated endpoints of all such callers passing the
//   same value of 'color'.
// + Within each newly created team, ranks are assigned (contiguously from
//   zero) by increasing order of the 'key' argument of the members.  In the
//   case of equal 'key', ties are broken by ranks in the 'parent_tm' team.
//   In particular this implies that if all ranks pass the same 'key' value,
//   then relative rank order from the 'parent_tm' is preserved in all created
//   teams.
// + The client may optionally provide scratch space within the bound segment
//   of the endpoint corresponding to 'parent_tm', for use by the
//   implementation.  No portion of this memory may be written by the client
//   or passed to any GASNet function, nor may the segment be destroyed, for
//   the lifetime of the newly created team.  When the team is destroyed,
//   ownership of this memory is returned to the client.
//   To NOT provide a scratch space, the client must pass 'flags' containing
//   'GEX_FLAG_TM_NO_SCRATCH'.
//   [TBD: what about Unbind of the segment w/o destroying it?]
//
//
//
```

```
// new_tm_p:  An OUT parameter that receives the gex_TM_t representing the
//            newly-created team, if any.
// parent_tm: The call is collective over this team.
// color:     A non-negative integer used to match callers to belong to the
//            same new team.
// key:       An integer used to order the ranks within newly created teams.
// scratch_addr, scratch_size:
//            If 'GEX_FLAG_TM_NO_SCRATCH' appears in 'flags', then these two
//            arguments are ignored.  Otherwise, the memory
//               [scratch_addr, scratch_addr+scratch_size)
//            is granted to the implementation for internal use.
//            The value of 'scratch_size' must be single-valued over the members
//            of each new team to be created (non-NULL 'new_tm_p' and same 'color').
//            The value of 'scratch_size' must non-zero.
// flags:
//   Single valued:
//       GEX_FLAG_TM_SCRATCH_SIZE_*
//         These mutually exclusive flags convert this call into a collective query.
//         No team is created in the presence of any flag in this family.
//         - GEX_FLAG_TM_SCRATCH_SIZE_RECOMMENDED
//           This query returns the recommended optimal value to be passed in
//           'scratch_size' for a subsequent call to gex_TM_Split() with the same
//           value for the other arguments.  In particular, a NULL value of the
//           'new_tm_p' indicates the caller will not be a member of any team
//           created by the subsequent split (and thus the return will be zero).
//           Return values are guaranteed to be single-valued over the members
//           of each new team to be created (non-NULL 'new_tm_p' and same 'color').
//         - GEX_FLAG_TM_SCRATCH_SIZE_MIN [DEPRECATED at spec version 0.11]
//           This flag is deprecated and will be removed in a future release.
//           Use in this release will printing a warning at runtime.
//   Partially single valued:
//       GEX_FLAG_TM_NO_SCRATCH
//         This flag causes creation of a team without a scratch space.  The
//         'scratch_addr' and 'scratch_size' arguments are ignored.  This flag
//         is intended for use when creating teams which will not perform any
//         significant collectives, and its use otherwise will most likely
//         degrade the performance of collectives.
//         Presence/absence of this flag must be single-valued over the members
//         of each new team to be created (non-NULL 'new_tm_p' and same 'color').
//   Non-single valued:
//        None currently defined
//
size_t gex_TM_Split(gex_TM_t *new_tm_p, gex_TM_t parent_tm, int color, int key,
                    void *scratch_addr, size_t scratch_size,
                    gex_Flags_t flags);

// Create zero or more new disjoint Teams
// [Since spec v0.9]
//
// This is a collective call which provides the means to construct one or more
// teams per call (at most one per caller) with greater generality than the
// gex_TM_Split(), including the ability to incorporate endpoints not yet in any
// team.
//
// While this call is collective, the arguments are NOT required to be
// single-valued over the parent team, except as noted for certain bits in
// 'flags'.  However, the value of some arguments must be collective over
// callers which comprise the same "output team".
//
// + Collective over parent_tm, which must contain at least one member for every
//   process named in the args[] of any caller.
// + When flags contains GEX_FLAG_TM_SCRATCH_SIZE_RECOMMENDED (presence of which
//   must be single-valued over the parent team), this API behaves analogously
//   to that documented for gex_TM_Split(): returning the 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.
//   Similarly, passing the (deprecated) GEX_FLAG_TM_SCRATCH_SIZE_MIN returns
//   the minimum scratch size.
//   In the absence of these flags, the remaining semantics apply.
// + Creates either zero (for numargs == 0) teams or one team (for numargs > 0)
//   per caller.
// + 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).
// + 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.
// + 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.
// + The presence/absence of GEX_FLAG_RANK_IS_JOBRANK in flags must be
//   single-valued over the output team.
// + The value of numargs and content of args[] must be single-valued over the
//   output team.
// + 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.
// + 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).
// + 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).
// + The endpoint corresponding to parent_tm is not required to be among the
//   entries in args[].
// + 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.
// + 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.
// + 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.
// + As with gex_TM_Split(), this "option" is actually required in the current
//   implementation.
// + The argument scratch_length must be single-valued over the output team.
// + 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.
// + The presence/absence of GEX_FLAG_SCRATCH_SEG_OFFSET in flags must be
//   single-valued over the output team.
// + 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).
//   - 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.
//   - 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.
//   - 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.
//   - 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 prohibition may be relaxed in the future).
// + 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.
// + The mutually exclusive choice of
//   GEX_FLAG_TM_{SYMMETRIC,LOCAL,GLOBAL,NO}_SCRATCH in flags must be
//   single-valued over the output team.
// + This call is guaranteed to provide sufficient synchronization that each
//   caller may begin using the new handles in new_tms[] immediately following
//   return. If flags included GEX_FLAG_TM_LOCAL_SCRATCH then this call provides
//   barrier synchronization individually over each new team created by the call.
//   In all other cases the implementation is permitted but not required to include
//   barrier synchronization, which may or may not be necessary to allow immediate
//   use of the resulting team.
//
// NOTE: The current implementation only supports creation of teams composed
// entirely of primordial endpoints, even with conduits which support creation
// of additional endpoints.  This limitation will be removed in a later release.
//
size_t gex_TM_Create(
            gex_TM_t *new_tms,          // OUT
            size_t num_new_tms,         // Length of new_tms
            gex_TM_t parent_tm,
            gex_EP_Location_t *args,    // IN
            size_t numargs,             // single-valued over output team
            gex_Addr_t *scratch_addrs,  // IN
            size_t scratch_size         // single-valued over output team
            gex_Flags_t flags);         // Flags (partially single-valued)

// Destroy a (quiesced) team
// [Since spec v0.10]
//
// This is a collective call to destroy a team which is no longer needed and
// reclaim associated resources.
//
// + This call is collective over members of the team named by tm.
// + Destroys the team, releasing resources allocated to it by the
//   implementation.
// + It is erroneous to destroy the primordial team.
// + Use of tm after return from this call is erroneous.
// + Does not destroy the endpoint associated with tm.
// + 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:
//   - No calls taking this tm as an argument are executing concurrently on
//     other threads.
//   - All collective operations using this tm are complete (client has synced
//     their gex_Event_t's).
//   - Any gex_AD_t objects created using this tm have been destroyed.
// + 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.
// + 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.
// + The presence/absence of GEX_FLAG_GLOBALLY_QUIESCED in flags must be
//   single-valued.
// + 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.
// + 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.
```

```
// + If a value is written to *scratch_p then return value is non-zero.
//   Otherwise, zero is returned.
// + [UNIMPLEMENTED] If GEX_FLAG_SCRATCH_SEG_OFFSET is set in flags, then the
//   value (if any) written to the gex_addr field of *scratch_p is assigned the
//   byte offset into the bound segment of the endpoint associated with tm.
//   Otherwise, the value (if any) assigned to this field is a virtual address.
// + 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.
// + Any cleanup action with respect to ClientData associated with the tm is
//   the client's responsibility.
//
// 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.
//
// The definition of "locally quiesced" intentionally excludes completion of
// non-blocking point-to-point operations using tm at their initiation.  This
// is possible because the semantics of such operations depend on the endpoints
// involved, and not on the tm used to name them.
//
// 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.
//
int gex_TM_Destroy(
            gex_TM_t      tm,
            gex_Memvec_t  *scratch_p,  // OUT
            gex_Flags_t   flags);

// Create an "ad hoc" TM for point-to-point communication
// [Since spec v0.12]
//
// 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 (or of any team at all).
//
// 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 include it in
// a team.
//
// + This is not a collective operation.
// + 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.
// + gex_TM_Pair is a lightweight, non-communicating utility call.
// + 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).
// + 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.
// + 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).
```

```
// + The result is valid for use in AM payload limit queries:
//   gex_AM_Max{Request,Reply}{Medium,Long}()
// + The result is valid for use in bound segment queries:
//   gex_Segment_QueryBound() [DEPRECATED] and gex_EP_QueryBoundSegmentNB()
// + 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 examples below.
//
// Example 1.
//   A 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);
//
// Example 2.
//   Communicating between a local endpoint ep0 and the remote endpoints with
//   index 1 in several processes, using a single TM-Pair:
//      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);
gex_TM_t gex_TM_Pair(
            gex_EP_t         local_ep,
            gex_EP_Index_t   remote_ep_index);

// Translations between (tm,rank) and jobrank
//
// These functions provide translations in either direction between a
// (tm,rank) pair and a jobrank.
//
// gex_Rank_t gex_TM_TranslateRankToJobrank(tm, rank)
//    Returns the jobrank of the endpoint in 'tm' with the given 'rank'.
//    Requires 0 <= rank < gex_TM_QuerySize(tm)
// gex_Rank_t gex_TM_TranslateJobrankToRank(tm, jobrank)
//    If there is an endpoint in 'tm' with the given 'jobrank', return its
//    rank in 'tm'.  Otherwise, returns GEX_RANK_INVALID.
//    Requires 0 <= jobrank < gex_System_QueryJobSize()
//
// These queries MAY communicate.
// [TBD: exception for 'self' in one both directions?]
// These calls are not legal in contexts which prohibit communication,
// including (but not limited to) AM Handler context or when holding an HSL.
//
gex_Rank_t gex_TM_TranslateRankToJobrank(gex_TM_t tm, gex_Rank_t rank);
gex_Rank_t gex_TM_TranslateJobrankToRank(gex_TM_t tm, gex_Rank_t jobrank);

// Translation from (tm,rank) to gex_EP_Location_t
//
// This function provides translation from a (tm,rank) pair to a
// gex_EP_Location_t, which is a (jobrank,epidx) pair.
//
// tm:      A valid gex_TM_t
// rank:    The rank of some member of tm.
//          Requires 0 <= rank < gex_TM_QuerySize(tm).
// flags:   Flags are reserved for future use and must currently be zero
//
// Returns: A gex_EP_Location_t describing the given member of tm.
//
// This query MAY communicate.
// [TBD: exception for 'self' in one both directions?]
// This call is not legal in contexts which prohibit communication,
// including (but not limited to) AM Handler context or when holding an HSL.
//
gex_EP_Location_t gex_TM_TranslateRankToEP(
                gex_TM_t         tm,
                gex_Rank_t       rank,
                gex_Flags_t      flags);
```

```
//
// Operations on gex_EP_t
// NOTE: currently gex_Client_Init() is the only way to create an EP.
// However, additional APIs for EP creation will be added.
//

// Query owning client
gex_Client_t  gex_EP_QueryClient(gex_EP_t ep);

// Query flags passed when ep was created
gex_Flags_t   gex_EP_QueryFlags(gex_EP_t ep);

// Query the bound segment
// Newly-created EPs have no bound segment and will yield GEX_SEGMENT_INVALID.
gex_Segment_t gex_EP_QuerySegment(gex_EP_t ep);

// Query the endpoint index
gex_EP_Index_t gex_EP_QueryIndex(gex_EP_t ep);

// Query addresses and length of a (possibly remote) bound segment
// [Since spec v0.13]
//
// This query takes a gex_TM_t and gex_Rank_t, which together name an endpoint.
// Other than flags, the remaining arguments are pointers to locations for
// outputs, each of which may be NULL if the caller does not need a particular
// value.
//
// If the value of flags does NOT include GEX_FLAG_IMMEDIATE, then this API
// behaves as follows:
// + The return value is a root event which can be successfully synchronized
//   (return from gex_Event_Wait*() or zero return from gex_Event_Test*())
//   once the query results have been written to the output locations.
//   It is permitted to be GEX_EVENT_INVALID (but not GEX_EVENT_NO_OP).
// + Between entering this call and synchronizing the event it returns, the
//   content of the output locations is undefined.
// + A "successful" query is one in which the endpoint named by (tm, rank) has a
//   bound segment *and* one or more of the following are true:
//     + The endpoint resides in the calling process
//     + The endpoint has a segment that was bound via gex_Segment_Attach()
//     + The endpoint had the bound segment at the time it was the subject of a
//       preceding call to gex_EP_PublishBoundSegment() in which the calling
//       process was a participant.
// + A successful query writes the corresponding segment's properties to each of
//   the non-NULL output locations as described in "Segment properties and output
//   locations", below.
// + If the endpoint named by (tm, rank) does not satisfy the above conditions
//   for a successful query, then the query may be "unsuccessful", whereby the
//   size_p output (unless NULL) will receive the value 0 and the remaining
//   outputs are undefined.  The implementation is thus permitted, but not
//   required, to be successful for a non-primordial bound segment which has not
//   yet been published to the calling process.
// + Since a segment cannot have zero-length, a caller can reliably distinguish
//   between a successful or unsuccessful query via the size_p output.
// + The current definition of "unsuccessful" notably includes the case of a
//   remote endpoint with a bound segment which has not been published to the
//   calling process.  However, the behavior for this case is subject to
//   possible change in a future release.
//
// In the case that flags DOES include GEX_FLAG_IMMEDIATE, then this API
// behaves as follows:
// + If the query can be resolved without communication, then the return value
//   is GEX_EVENT_INVALID, with the behavior otherwise identical to the case
//   without GEX_FLAG_IMMEDIATE.
// + Queries which would require communication to resolve will return
//   GEX_EVENT_NO_OP.
```

```
// + All queries for which (tm, rank) names an endpoint which resides in the
//   calling process are guaranteed to return GEX_EVENT_INVALID.
// + Queries for which (tm, rank) names an endpoint which does not reside in the
//   calling process may return either GEX_EVENT_INVALID or GEX_EVENT_NO_OP and
//   the same query is not guaranteed to return the same value each time.
//   This permits an implementation to cache information for remote endpoints.
//
// Segment properties and output locations:
//   owneraddr_p: receives the address of the segment in the address space
//                of the process which owns the segment.
//                For segments of kind GEX_MK_HOST, this is a host address
//                while for all other kinds this is a device address.  In
//                either case it is the address which would be returned by
//                gex_Segment_QueryAddr() immediately after segment creation
//                (via either gex_Segment_Attach() or gex_Segment_Create()).
//   localaddr_p: receives the address of the segment in the address space
//                of the calling process, *if* mapped, and NULL otherwise.
//   size_p:      receives the length of the segment.
//
// Only segments of kind GEX_MK_HOST may report a non-NULL localaddr property,
// and all other kinds will yield NULL.  The current release additionally
// limits the reporting of non-NULL values to primordial segments (those
// created by gex_Segment_Attach()).
//
// Passing GEX_RANK_INVALID as the rank argument is *not* permitted.
// Use of a TM-pair for the 'tm' argument *is* permitted.
// Passing a '(tm,rank)' tuple naming an endpoint residing on the calling
// process *is* permitted.
//
// When passing a '(tm,rank)' tuple naming an endpoint not residing on the
// calling process, this query MAY communicate unless GEX_FLAG_IMMEDIATE is
// included in flags.
// If and only if GEX_FLAG_IMMEDIATE is included in flags, then this call is
// permitted in contexts which prohibit communication (such as AM Handler
// context or when holding an HSL).
extern gex_Event_t gex_EP_QueryBoundSegmentNB(
            gex_TM_t       tm,
            gex_Rank_t     rank,
            void           **owneraddr_p,
            void           **localaddr_p,
            uintptr_t      *size_p,
            gex_Flags_t    flags);

// Query addresses and length of a (possibly remote) bound segment
// [DEPRECATED since spec v0.13 - see gex_EP_QueryBoundSegmentNB(), above]
//
// This query provides semantics similar to
//   gex_Event_Wait( gex_EP_QueryBoundSegmentNB([...args...], 0) )
// where "[...args...]" represent the five arguments to this query.
//
// The semantic differences are as follows:
// + Success/failure
//   - This call returns zero for a "successful" query, defined as one in which
//     (tm, rank) names an endpoint with a bound segment (and, if remote, that
//     segment is primordial or has been published to the caller).  Otherwise,
//     a non-zero value is returned.
//   - An successful query with gex_EP_QueryBoundSegmentNB() is distinguishable
//     by a non-zero size output, while an unsuccessful query will write zero
//     to the size output.
// + Preservation of outputs on failure
//   - This call guarantees that an unsuccessful query leaves the outputs
//     unmodified.
//   - An unsuccessful query with gex_EP_QueryBoundSegmentNB() writes zero to
//     the size output and leaves the others undefined.
//
// This call is not legal in contexts which prohibit communication, including
```

```
// (but not limited to) AM Handler context or when holding an HSL.
int gex_Segment_QueryBound(
                gex_TM_t    tm,
                gex_Rank_t  rank,
                void        **owneraddr_p,
                void        **localaddr_p,
                uintptr_t   *size_p);

// Publish of EP's Bound Segment "RMA Credentials"
//
// Description:
//   Some conduits require "credentials" to initiate communication targeting
//   the bound segment of a remote endpoint.  This call performs any
//   communication and setup necessary to ensure that after successful return
//   the local process may safely initiate such communication with any
//   endpoint named in this call which had a bound segment at the time of
//   this call.
//
// Semantics:
// + On success, returns GASNET_OK.
// + Non-fatal failures return a documented error code.
// + Lack of sufficient resources to satisfy the given request will yield a
//   return of GASNET_ERR_RESOURCE.
// + This call is collective over tm, which identifies a team used for
//   underlying communication.
// + The eps argument is an array of length num_eps (possibly zero) of valid
//   endpoints.
// + The num_eps argument may vary by caller (it is not required to be
//   single-valued).
// + This call publishes the bound segments, if any, of the endpoints named
//   by the eps argument.
// + The endpoint associated with tm is not implicitly Published, but it may
//   be explicitly included in eps if Publication is desired.
// + The concatenation of eps arrays must name distinct endpoints.
//   Duplication is prohibited both within a given eps array, and across eps
//   arrays passed by multiple tm (from the same team) within a given
//   process.  This restriction may be relaxed in a future release.
// + Upon successful return, the local process may safely initiate
//   communication targeting the bound segment of any endpoint named by the
//   eps arguments which had a bound segment prior to the corresponding entry
//   to this collective call.
// + It is permitted for eps to contain endpoints without a bound segment, in
//   which case no credential will be published for such endpoints.
// + It is permitted for the same endpoint to be the subject of multiple
//   successive Publish operations and any bound segment will replace a prior
//   Publish in which an endpoint had no bound segment.
// + The allowance for multiple Publish operations includes the one implicit
//   in gex_Segment_Attach().
// + The endpoints named by eps must be idle for the duration of this operation.
//   - No communication operations may be in-flight on any named endpoint
//     when this operation starts.
//   - No communication operations may be initiated on any named endpoint
//     concurrent with this operation.
//   - No AM Request may target any named endpoint for the duration of this
//     operation.
//   - As an exception to the restrictions above, inclusion of the endpoint
//     associated with tm in eps is explicitly permitted.
//   - A named endpoint may not be the subject of concurrent segment
//     operations including (but not limited to) gex_Segment_QueryBound,
//     gex_EP_BindSegment, gex_EP_PublishBoundSegment, and
//     gex_EP_QueryBoundSegmentNB.
// + The publication of credentials is per local process and remote endpoint,
//   independent of the specific team used to perform this operation.  This
//   means that upon return, initiation of communication is permitted using
//   any (tm_x, rank) pair from a participating process naming a participating
//   remote endpoint, including initiation using a gex_TM_t created using
```

```
//   gex_TM_Pair().  Additionally, this persists beyond destruction of the
//   team used to Publish.
// + The flags argument is reserved for future use and must currently be
//   zero.
// + This call is permitted but not required to incur barrier synchronization
//   across the team.

extern int gex_EP_PublishBoundSegment(
                gex_TM_t               tm,
                gex_EP_t               *eps,    // IN
                size_t                 num_eps,
                gex_Flags_t            flags);


// Minimum permitted fixed index for AM handler registration.
// Applies to both gasnet_attach() and gex_EP_RegisterHandlers().
// An integer constant, guaranteed to be 128 or less.
#define GEX_AM_INDEX_BASE ???

//
// Conduit-internal progress threads
//
// A conduit may include one or more threads intended to provide asynchronous
// progress.
//
// If present, a "receive progress thread" (or just "receive thread") is
// intended to progress the reception of AMs, and consequently may run
// client-provided handlers.
//
// If present, a "send progress thread" (or just "send thread") is intended to
// perform internal actions to progress various in-flight communication
// operations.  This thread will never run client-provided AM handler code.

// Implementation-induced concurrency of AM handlers
// This value (always defined) has a non-zero value iff the implementation
// may run AM handlers from a receive thread not owned by the client
// (and in particular, concurrently with the client when no client
// thread is inside a synchronous call to GASNet).
// Note this is orthogonal to SEQ/PAR/PARSYNC mode - in particular, in PAR
// mode multiple client threads concurrently entering GASNet may result
// in AM handler concurrency, independent of this value.
#define GASNET_HIDDEN_AM_CONCURRENCY_LEVEL ???

// Returns the runtime value of AM concurrency level for the calling process
// which may be more precise than the conservative static value provided by
// GASNET_HIDDEN_AM_CONCURRENCY_LEVEL.  In particular, this query is sensitive
// to whether a conduit's receive thread(s) are enabled or disabled at run time.
// Only valid after gex_Client_Init().
// The value is constant across multiple calls, and in particular if the client
// has passed `GEX_FLAG_DEFER_THREADS` to `gex_Client_Init()` then the result
// reflects the expected concurrency once the client has started all progress
// threads.
// [Since spec v0.14]
int gex_System_QueryHiddenAMConcurrencyLevel(void);

//
// Deferred progress thread initialization
// [Since spec v0.18] [EXPERIMENTAL]
//

// Query deferred progress threads
// [Since spec v0.18] [EXPERIMENTAL]
//
// The `gex_System_QueryProgressThreads()` query provides information about any
// progress threads enabled in the library that the client is responsible for
// starting due to passing `GEX_FLAG_DEFER_THREADS` to `gex_Client_Init()`.  The
```

```
// information provided is intended to allow the client to make decisions about
// such issues as CPU and memory affinity prior to starting these threads.
//
// + A client which passes `GEX_FLAG_DEFER_THREADS` to `gex_Client_Init()` is
//   required to make exactly one call to this query and to start each of
//   the threads described in the result.
// + If a client which passes `GEX_FLAG_DEFER_THREADS` to `gex_Client_Init()`
//   makes multiple calls to this query, then the behavior is undefined.
// + If `GEX_FLAG_DEFER_THREADS` was not passed to `gex_Client_Init()`, then
//   calls to this query are erroneous (returning `GASNET_ERR_RESOURCE` if
//   there are no other errors).
// + Each thread must be started by invoking `gex_progress_fn(gex_progress_arg)`
//   in a POSIX thread, where `gex_progress_*` name fields of the query result
//   (described below).  This may be accomplished by passing these two field
//   values as the third and fourth arguments to `pthread_create()`, but a
//   client is free to invoke the progress function from a thread it has
//   created by other means.
// + In the case of a normal `gasnet_exit()`, the library will terminate each
//   progress thread either by inducing it to call `pthread_exit()` or via
//   `pthread_cancel()`.  The library assumes responsibility for joining or
//   detaching the progress thread.  The client is responsible for providing
//   the progress thread in a joinable state, and should neither join nor
//   detach the progress thread.
// + In no case will the progress function return.  A client starting a
//   progress thread by means other than by calling `pthread_create()` must not
//   depend on return from the progress function.
// + The client may start the threads in any order and/or concurrently.
// + In general, failure to start all threads prior to `gasnet_exit()` or
//   process termination (whether normal or abnormal) leads to undefined
//   behavior.  However, a given conduit may relax this restriction (see the
//   conduit-specific README).
// + Other than `gasnet_exit()`, there are no restrictions on GASNet calls which
//   the client may make prior to starting the progress threads.  This means the
//   client may communicate as necessary to collect information needed to make
//   decisions regarding CPU and/or memory locality.
// + The result of this query is the tuple `(*count_p, *info_p)`, where
//   the expression `*info_p` points to library-owned memory which must not be
//   modified or freed by the client.
// + The lifetime of this library-owned memory extends from return from this
//   call until all the progress threads start.  Consequently, any accesses to
//   this memory after all progress threads have begun running have undefined
//   behavior.
// + The `client` argument is the client created by `gex_Client_Init()`.
// + The `count_p` argument is a pointer to an `unsigned int` which, upon
//   successful return, will contain the number of deferred progress threads.
//   This may be zero.
// + The `info_p` argument is a pointer to a `const gex_ProgressThreadInfo_t *`
//   which, upon successful return with non-zero `*count_p`, will contain the
//   address of an array (of length `*count_p`) of structures each describing a
//   single progress thread.
//   See the description of `gex_ProgressThreadInfo_t`, below, for details.
// + If `*count_p` is zero, then the value of `*info_p` is undefined.
// + The `flags` argument is reserved for future use and must currently be zero.
// + If the `count_p` or `info_p` arguments are `NULL`, or if the `flags`
//   argument is non-zero, `GASNET_ERR_BAD_ARG` is returned in the absence of
//   other errors.
// + If the call returns an error (non-zero), then the content of the locations
//   named by `count_p` and `info_p` are undefined.
// + If there are multiple reportable errors, the precedence is undefined.
int gex_System_QueryProgressThreads(
                gex_Client_t                     client,
                unsigned int                    *count_p,
                const gex_ProgressThreadInfo_t **info_p,
                gex_Flags_t                      flags);
```

```
// Type to describe deferred progress threads
// [Since spec v0.18] [EXPERIMENTAL]
//
// The result of the `gex_System_QueryProgressThreads()` query is an array
// of zero or more of this structure, each describing a single progress thread
// for which the conduit has deferred launch due to `GEX_FLAG_DEFER_THREADS`
// in the `flags` passed to `gex_Client_Init()`:
//
// + gex_device_list
//   This field is a non-NULL pointer to a comma-delimited string listing the
//   "devices" for which this thread will provide progress.  The nature of the
//   comma-delimited names is conduit-defined (in the respective README).
// + gex_thread_roles
//   This field has a non-zero value generated by bitwise-OR of one or more
//   GEX_THREAD_ROLE_* constants below, or ones added in a later version of this
//   specification.  This value indicates the role or roles this thread takes in
//   progressing communications.
// + gex_progress_fn and gex_progress_arg
//   This pair are the function and argument which the client should ensure
//   will run in a joinable POSIX thread (such as by passing them as the third
//   and fourth arguments to `pthread_create()`).
//
typedef struct {
  const char *      gex_device_list;
  unsigned int      gex_thread_roles;
  void *            (*gex_progress_fn) (void *);
  void *            gex_progress_arg;
} gex_ProgressThreadInfo_t;

// Thread roles in gex_ProgressThreadInfo_t
// [Since spec v0.18] [EXPERIMENTAL]
#define GEX_THREAD_ROLE_RCV ???
#define GEX_THREAD_ROLE_SND ???

// Preprocessor defines advertising configured progress thread support
// [Since spec v0.18]

// GASNET_RCV_THREAD is defined to 1 iff the current library build includes
// support for a receive thread.  Because the user can typically use environment
// variables to enable or disable launch of this thread, this does not indicate
// with certainty that the thread will run.
//
// If the current conduit does not include any receive thread support, or it was
// disabled at configure time, then GASNET_RCV_THREAD will be #undef.
//
// Clients which may set environment variables to request a receive thread
// should make such logic conditional on this preprocessor identifier to avoid
// conduit-specific warnings which may result when requesting a receive thread
// which is not enabled.
#define GASNET_RCV_THREAD 1 or #undef

// GASNET_SND_THREAD is defined to 1 iff the current library build includes
// support for a send thread.  Because the user can typically use environment
// variables to enable or disable launch of this thread, this does not indicate
// with certainty that the thread will run.
//
// If the current conduit does not include any send thread support, or it was
// disabled at configure time, then GASNET_SND_THREAD will be #undef.
//
// Clients which may set environment variables to request a send thread should
// make such logic conditional on this preprocessor identifier to avoid
// conduit-specific warnings which may result when requesting a send thread
// which is not enabled.
#define GASNET_SND_THREAD 1 or #undef
```

```
// AM handlers
//

// Client-facing type for describing one AM handler
// This type is an alternative to (*not* interchangeable with) gasnet_handlerentry_t
//
// gex_index may either be in the range [GEX_AM_INDEX_BASE .. 255] to register
// at a fixed index, or 0 for "don't care" (see gex_EP_RegisterHandlers() for
// more information on this case).
//
// The gex_nargs and gex_flags fields are used by the client to supply the implementation
// with assertions regarding the future invocations and behavior of each AM handler.
// If a handler invocation (eg via an AM injection targeting a given handler) or
// execution of an AM handler violates its registration assertions, behavior is undefined.
typedef struct {
    gex_AM_Index_t          gex_index;      // 0 or in [GEX_AM_INDEX_BASE .. 255]
    gex_AM_Fn_t             gex_fnptr;      // Pointer to the handler on this process
    gex_Flags_t             gex_flags;      // Incl. required S/M/L and REQ/REP, see below
    unsigned int            gex_nargs;      // Required in [0 .. gex_AM_MaxArgs()]

    // Optional fields (both are "shallow copy")
    const void             *gex_cdata;      // Available to handler
    const char             *gex_name;       // Used in debug messages
} gex_AM_Entry_t;

// Required flags for gex_flags field when registering AM handlers.
//
// When registering AM handlers, the gex_flags field of each
// gex_AM_Entry_t must indicate how the handler may be called.
// This requires ORing one constant from each of the following
// two groups.

// AM Category Flags:
#define GEX_FLAG_AM_SHORT      ??? // Called only as a Short
#define GEX_FLAG_AM_MEDIUM     ??? // Called only as a Medium
#define GEX_FLAG_AM_LONG       ??? // Called only as a Long
#define GEX_FLAG_AM_MEDLONG    ??? // Called as a Medium or Long

// AM Request/Reply Flags:
#define GEX_FLAG_AM_REQUEST    ??? // Called only as a Request
#define GEX_FLAG_AM_REPLY      ??? // Called only as a Reply
#define GEX_FLAG_AM_REQREP     ??? // Called as a Request or Reply

// gex_EP_RegisterHandlers()
//
// Registers a client-provided list of AM handlers with the given EP, with
// semantics similar to gasnet_attach().  However, unlike gasnet_attach()
// this function is not collective and does not include an implicit barrier.
// Therefore the client must provide for any synchronization required to
// ensure handlers are registered before any process may send a corresponding
// AM to the Endpoint.
//
// May be called multiple times on the same Endpoint to incrementally register handlers.
// Like gasnet_attach() the handler indices specified in the table (other than
// "don't care" zero indices) must be unique.  That now extends across multiple
// calls on the same gex_EP_t (though provisions to selectively relax this
// restriction are planned for a later release).
//
// Registration of handlers via a call to gasnet_attach() does *not* preclude
// use of this function to register additional handlers.
//
// As in GASNet-1, handlers with a handler index (gex_index) of 0 on entry are
// assigned values by GASNet after the non-zero (fixed index) entries have been
// registered.  While GASNet-1 leaves the algorithm for the assignment
// unspecified (only promising that it is deterministic) this specification
// guarantees that entries with gex_index==0 are processed in the same order
```

```
// they appear in 'table' and are assigned the highest-numbered index which is
// then still unallocated (where 255 is the highest possible).  However, in
// the case of concurrent calls to gex_EP_RegisterHandlers() and/or
// gasnet_attach() on the same endpoint with gex_index==0, the order in which
// such entries are processed is unspecified and may be non-deterministic.
//
// Updating of gex_index fields that were passed as 0 upon input is the only
// modification this function will perform upon the contents of 'table'
// (whose elements are otherwise treated as if const-qualified by this call).
// Upon return from this function, the relevant information from 'table'
// has been copied into storage internal to the endpoint implementation,
// and the client is permitted to overwrite or free the contents of 'table'.
//
// If any sequence of calls attempts register a total of more than (256 -
// GEX_AM_INDEX_BASE) handlers to a single gex_EP_t, the result is undefined
//
// Returns: GASNET_OK == 0 on success
int gex_EP_RegisterHandlers(
        gex_EP_t                ep,
        gex_AM_Entry_t          *table,
        size_t                  numentries);

// Active Message (AM) limit queries

// Maximum number of supported AM arguments
// Semantically identical to gasnet_AMMaxArgs()
unsigned int gex_AM_MaxArgs(void);

// Maximum payload size queries
// Superset of gasnet_AMMax{Medium,LongRequest,LongReply}()
//
// This family of calls provide maximum payload queries of two types:
// + In the absence of the GEX_FLAG_AM_PREPARE_LEAST_{CLIENT,ALLOC} flags,
//   these queries return the maximum legal 'nbytes' argument value for the
//   corresponding gex_AM_{Request,Reply}{Medium,Long}*() call (collectively
//   known as "fixed-payload AM" injection calls) using the named local and
//   remote endpoint and the same 'lc_opt', 'numargs' and 'flags' arguments.
// + When passed either of the GEX_FLAG_AM_PREPARE_LEAST_{CLIENT,ALLOC} flags,
//   these queries return the maximum legal 'least_payload' argument value for
//   the corresponding gex_AM_Prepare{Request,Reply}{Medium,Long}() call
//   (collectively known as "negotiated-payload AM" prepare calls) using the
//   named local and remote endpoint and the same 'lc_opt', numargs' and 'flags'
//   arguments.
//
// 1. If 'tm' names a local endpoint which is not AM-capable, then the call
//    is erroneous.  Here "AM-capable endpoint" is defined as any primordial
//    endpoint or a non-primordial endpoint which was created with
//    GEX_EP_CAPABILITY_AM.
// 2. When (other_rank != GEX_RANK_INVALID)
//    a. The result of each query is a function of the 'numargs', 'lc_opt' and
//       'flags' arguments, and the two endpoints (one local and one remote)
//       named by the tuple consisting of the 'tm' and 'other_rank' arguments.
//    b. The result is independent of *how* the endpoints are named, such as by
//       distinct 'tm' values with overlapping membership or use of a TM-pair.
//    c. If the remote endpoint named by the '(tm,other_rank)' tuple is not
//       AM-capable or does not exist (only possible with a TM-pair), then the
//       call is erroneous.
// 3. When (other_rank == GEX_RANK_INVALID)
//    a. The result of each query is a min-of-maxes over all AM-capable remote
//       endpoints that are addressable with the given 'tm' when 'other_rank'
//       is varied over its valid range, with the given 'numargs', 'lc_opt' and
//       'flags' arguments.
//    b. This valid range excludes any endpoints which are not AM-capable.
//    c. In the case that 'tm' is a TM-pair, the valid range also excludes
//       jobranks which do not have an endpoint at the associated remote
//       endpoint index.
```

```
//    d. If valid range defined above is empty (no AM-capable endpoints are
//       addressable), then the call is erroneous.
// 4. The result of each query function is guaranteed to be symmetric with
//    respect to exchanging the local and remote endpoints.  Two calls, by
//    appropriate processes, that reverse the local and remote endpoint roles
//    while keeping all other input arguments equal, are guaranteed to return
//    the same value.  Note this does NOT imply any relationship between the
//    results of different query functions (eg MaxRequestMedium versus
//    MaxReplyMedium).
// 5. When (other_rank == GEX_RANK_INVALID) all callers providing a 'tm' naming
//    the same set of participating endpoints are guaranteed to get the same
//    result when given the same values for the other input arguments.  Due to
//    the symmetry noted above, this includes two calls using TM-pairs to
//    identify the same two endpoints.
// 6. Due to the symmetry properties described above, 'other_rank' can (and
//    therefore should) always name the other party in the communication,
//    regardless of whether that rank or the caller is to be the sender or the
//    receiver.
// 7. 'numargs' must be between 0 and gex_AM_MaxArgs(), inclusive.  It is
//    guaranteed that increasing 'numargs' will produce monotonically non-
//    increasing results when all other parameters are held fixed.
// 8. 'lc_opt' indicates the payload local completion option to be used for
//    the AM injection or prepare call in question.  The predefined constants
//    GEX_EVENT_NOW and GEX_EVENT_GROUP should be used directly, while a
//    pointer to any variable of type gex_Event_t (or a NULL pointer) may be
//    used interchangeably to indicate that the injection or prepare call
//    passes any such value (without requiring that the same pointer value be
//    passed).
// 9. 'flags' indicates the flags that will be provided to the corresponding
//    AM injection or prepare function (and should not to be confused with the
//    handler registration flags).  The result of the query is only guaranteed
//    to be correct for an injection of prepare call with exactly the same
//    'flags', excepting only that the GEX_FLAG_AM_PREPARE_LEAST_* flags may
//    be omitted from a prepare call.
//
// The result of all four query functions is guaranteed to be at least 512 (bytes).
//
// The result is guaranteed to be stable throughout a given job execution - ie
// for the same set of input arguments, it will always return the same value.
//
// Aside from the explicit guarantees above, the result may otherwise vary with
// the input arguments in unspecified ways, and thus only defines the documented
// limit for an call with corresponding local and remote endpoints and values of
// lc_opt, flags and numargs.  For example, limits often vary between different
// conduits and may also vary based on job layout, between pairs of ranks in
// the same team, or between different pair of endpoints linking the same two
// processes.

size_t gex_AM_MaxRequestLong(
           gex_TM_t tm,
           gex_Rank_t other_rank,
           const gex_Event_t *lc_opt,
           gex_Flags_t flags,
           unsigned int numargs);
size_t gex_AM_MaxReplyLong(
           gex_TM_t tm,
           gex_Rank_t other_rank,
           const gex_Event_t *lc_opt,
           gex_Flags_t flags,
           unsigned int numargs);
size_t gex_AM_MaxRequestMedium(
           gex_TM_t tm,
           gex_Rank_t other_rank,
           const gex_Event_t *lc_opt,
           gex_Flags_t flags,
           unsigned int numargs);
```

```
size_t gex_AM_MaxReplyMedium(
           gex_TM_t tm,
           gex_Rank_t other_rank,
           const gex_Event_t *lc_opt,
           gex_Flags_t flags,
           unsigned int numargs);

// Token-specific max fixed-payload queries for specific nargs, lc_opt and flags
//
// Semantics are identical to the may payload queries above, except that
// a gex_Token_t replaces the (tm,rank) tuple.  The token names the local
// endpoint on which the AM has been received and the remote endpoint which
// sent it. In particular, this implies the queries return the limits
// governing the AM Reply operations that can be performed using this token.
//
// These are only permitted in Request handlers.
size_t gex_Token_MaxReplyLong(
           gex_Token_t token,
           const gex_Event_t *lc_opt,
           gex_Flags_t flags,
           unsigned int numargs);
size_t gex_Token_MaxReplyMedium(
           gex_Token_t token,
           const gex_Event_t *lc_opt,
           gex_Flags_t flags,
           unsigned int numargs);

// Least-upper-bound fixed-payload queries (unknown team/peer, nargs, lc_opt and flags)
// Guaranteed to be less than or equal to the result of the corresponding AM_Max*
// function, for all valid input parameters to that function (excluding use of the
// GEX_FLAG_AM_PREPARE_LEAST_* flags).
// The result of all four query functions is guaranteed to be at least 512 (bytes).
// These functions correspond semantically to the gasnet_AMMax*() queries in GASNet-1,
// which return a globally conservative maximum.
size_t gex_AM_LUBRequestLong(void);
size_t gex_AM_LUBReplyLong(void);
size_t gex_AM_LUBRequestMedium(void);
size_t gex_AM_LUBReplyMedium(void);

// AM Token Info
//
// Struct type for gex_Token_Info queries contains *at least* the following
// fields, in some *unspecified* order
typedef struct {
    // "Job rank" of the sending process, as defined with the description
    //  of gex_System_QueryJobRank().
    gex_Rank_t                gex_srcrank;

    // Destination (receiving) endpoint
    gex_EP_t                  gex_ep;

    // Entry describing the currently-running handler corresponding to this token.
    // The referenced gex_AM_Entry_t object resides in library-owned storage,
    // and should not be directly modified by client code.
    // If handler was registered using the legacy gasnet_attach() call, this
    // value may be set to a valid pointer to a gex_AM_Entry_t, with undefined
    // contents.
    const gex_AM_Entry_t     *gex_entry;

    // 1 if the current handler is a Request, 0 otherwise.
    [some integral type]      gex_is_req;

    // 1 if the current handler is a Long, 0 otherwise.
    [some integral type]      gex_is_long;
} gex_Token_Info_t;
```

```
// Bitmask constants to request specific info from gex_Token_Info():
// All listed constants are required, but the corresponding queries
// are divided into Required ones and Optional ones (with the
// exception of GEX_TI_ALL).
typedef [some integer type] gex_TI_t;

// REQUIRED: All implementations must support these queries:
#define GEX_TI_SRCRANK       ((gex_TI_t)???)     # required since spec v0.1
#define GEX_TI_EP            ((gex_TI_t)???)     # required since spec v0.1

// OPTIONAL: Some implementations might not support these queries:
#define GEX_TI_ENTRY         ((gex_TI_t)???)     # optional since spec v0.1
#define GEX_TI_IS_REQ        ((gex_TI_t)???)     # optional since spec v0.1
#define GEX_TI_IS_LONG       ((gex_TI_t)???)     # optional since spec v0.1

// Convenience: all defined queries (Required and Optional)
#define GEX_TI_ALL           ((gex_TI_t)???)     # required since spec v0.1

// Support indicators for Optional token into queries
// Available since spec v0.17
//
// GASNET_SUPPORTS_TI_* preprocessor identifiers are defined to 1 or #undef
// to indicate whether (or not, respectively) the implementation of
// gex_Token_Info() supports the corresponding query for all valid tokens.
//
// When any of these is defined for an Optional query, it is an indication that
// the current implementation of the current conduit supports the
// corresponding query.  However, it is not a guarantee of such support in
// other conduits or in future releases of the current conduit.
//
// When any of these is #undef, the implementation is still permitted to
// support the query conditionally.  For instance, the shared-memory transport
// may support an Optional query that is not supported for AMs travelling
// outside of the shared-memory nbrhd, or vice-versa.

#define GASNET_SUPPORTS_TI_SRCRANK      1
#define GASNET_SUPPORTS_TI_EP           1
#define GASNET_SUPPORTS_TI_ENTRY        1 or #undef
#define GASNET_SUPPORTS_TI_IS_REQ       1 or #undef
#define GASNET_SUPPORTS_TI_IS_LONG      1 or #undef

// Takes a token, address of client-allocated gex_Token_Info_t, and a mask.
// The mask is a bit-wise OR of GEX_TI_* constants, which indicates which
// fields of the gex_Token_Info_t should be set by the call.
//
// The return value is of the same form as the mask.
// The implementation is permitted to set fields not requested by the
// caller to valid or *invalid* values.  The returned mask will indicate
// which fields contain valid results, and may include bits not present
// in the mask.
//
// Each GEX_TI_* corresponds to either a Required or Optional query.
// When a client requests a Required query, a conforming implementation
// MUST set these fields and the corresponding bit in the return value.
// An Optional query may not be implemented on all conduits or all
// configurations, or even under various conditions (e.g. may not be
// supported in a Reply handler).  If the client makes an Optional request
// the presence of the corresponding bit in the return value is the only
// indication that the struct field is valid.
extern gex_TI_t gex_Token_Info(
                gex_Token_t         token,
                gex_Token_Info_t    *info,
                gex_TI_t            mask);
```

```
//
// Fixed-payload AM APIs
//

// NOTE 0: Prototypes in this section are "patterns"
//
//   These API instantiate the "[M]" at the end of each prototype with
//   the integers 0 through gex_AM_MaxArgs(), inclusive.
//   The '[,arg0, ... ,argM-1]' then represent the arguments
//   (each of type gex_AM_Arg_t).
//   Additionally, on compilers supporting the __VA_ARG__ preprocessor feature
//   (added in C99 and C++11) the "[M]" may optionally be omitted entirely and
//   is inferred based on the argument count.
//
// NOTE 1: Return value
//
//   An AM Request or Reply call is a "no op" IF AND ONLY IF the value
//   GEX_FLAG_IMMEDIATE is included in the 'flags' argument AND the
//   conduit could determine that it would need to block temporarily to
//   obtain the necessary resources.  This case is distinguished by a
//   non-zero return.  In all other cases the return value is zero.
//
//   In the "no op" case no communication has been performed and the
//   contents of the location named by the 'lc_opt' argument (if any) is
//   undefined.
//
// NOTE 2: The 'lc_opt' argument for local completion
//
//   The AM interfaces never detect or report remote completion, but do
//   have selectable behavior with respect to local completion (which
//   means that the source buffer may safely by written, free()ed, etc).
//
//   Short AMs have no payload and therefore have no 'lc_opt' argument.
//
//   The Medium and Long Requests accept the pre-defined constant values
//   GEX_EVENT_NOW and GEX_EVENT_GROUP, and pointers to variables of type
//   'gex_Event_t' (note that GEX_EVENT_DEFER is prohibited).
//   The NOW constant requires that the Request call not
//   return until after local completion.  The GROUP constant allows the
//   Request call to return without delaying for local completion and adds
//   the AM operation to the set of operations for which
//   gex_NBI_{Test,Wait}() call may check local completion when passed
//   GEX_EC_AM.  Use of a pointer to a variable of type 'gex_Event_t'
//   allows the call to return without delay, and requires the client to later
//   check local completion of this root event using gex_Event_{Test,Wait}*().
//
//   The 'lc_opt' argument to Medium and Long Reply calls behave as for the
//   Requests with the exception that GEX_EVENT_GROUP is *not* permitted.
//   It is also important to note that it is not legal to "test", or
//   "wait" on a 'gex_Event_t' in AM handler context.
//   [TBD: we *could* allow handlers to make bounded calls to "test", which
//   does not Poll, if we wanted to.]
//
// NOTE 3: The 'flags' argument for segment disposition [UNIMPLEMENTED]
//
//   The 'flags' argument to Medium and Long Request/Reply calls may include
//   GEX_FLAG_SELF_SEG_* flags to assert segment disposition properties of the
//   address range described by [source_addr..(source_addr+nbytes-1)]. Any such
//   assertions must remain true until local completion is signalled (see above).
//
//   The 'flags' argument to Long Request/Reply calls may include
//   GEX_FLAG_PEER_SEG_* flags to assert segment disposition properties of the
//   address range described by [dest_addr..(dest_addr+nbytes-1)]. Any such
//   assertions must remain true until the AM handler begins execution at the target.
//
//
```

```
// NOTE 4: Overlap
//   Within a single gex_AM_*Long* operation, if the specified source and destination
//   memory regions overlap, behavior is undefined. High-quality implementations
//   may choose to diagnose such errors.
//
// NOTE 5: Longs and Bound Segments
//   In general, the 'tm' and 'rank' arguments to Long Request/Reply calls must
//   name a destination EP with a bound segment known to the initiator, where
//   "known" means the segment was created with gex_Segment_Attach() or was
//   published to the initiator.  However, for the special case of 'nbytes == 0`,
//   no bound segment is required.
//
// Other arguments behave as in the analogous GASNet-1 functions.
// Misc semantic strengthening:
// * dest_addr for Long is guaranteed to be passed to the handler as provided
//   by the initiator, even for the degenerate case when nbytes==0
// Long
int gex_AM_RequestLong[M](
           gex_TM_t tm,                // Names a local context ("return address")
           gex_Rank_t rank,            // Together with 'tm', names a remote context
           gex_AM_Index_t handler,     // Index into handler table of remote context
           const void *source_addr,    // Payload address (or OFFSET)
           size_t nbytes,              // Payload length
           void *dest_addr,            // Payload destination address (or OFFSET)
           gex_Event_t *lc_opt,        // Local completion control (see above)
           gex_Flags_t flags           // Flags to control this operation
           [,arg0, ... ,argM-1])       // Handler argument list, each of type gex_AM_Arg_t
int gex_AM_ReplyLong[M](
           gex_Token_t token,          // Names local and remote contexts
           gex_AM_Index_t handler,
           const void *source_addr,
           size_t nbytes,
           void *dest_addr,
           gex_Event_t *lc_opt,
           gex_Flags_t flags
           [,arg0, ... ,argM-1]);
// Medium
int gex_AM_RequestMedium[M](
           gex_TM_t tm,
           gex_Rank_t rank,
           gex_AM_Index_t handler,
           const void *source_addr,
           size_t nbytes,
           gex_Event_t *lc_opt,
           gex_Flags_t flags
           [,arg0, ... ,argM-1]);
int gex_AM_ReplyMedium[M](
           gex_Token_t token,
           gex_AM_Index_t handler,
           const void *source_addr,
           size_t nbytes,
           gex_Event_t *lc_opt,
           gex_Flags_t flags
           [,arg0, ... ,argM-1]);
// Short
int gex_AM_RequestShort[M](
           gex_TM_t tm,
           gex_Rank_t rank,
           gex_AM_Index_t handler,
           gex_Flags_t flags
           [,arg0, ... ,argM-1]);
int gex_AM_ReplyShort[M](
           gex_Token_t token,
           gex_AM_Index_t handler,
           gex_Flags_t flags
           [,arg0, ... ,argM-1]);
```

```
//
// Negotiated-payload AM APIs (aka "NPAM")
//

// The fixed-payload APIs for Active Message Mediums and Longs (brought
// forward from GASNet-1) allow sending any payload up to defined maximum
// lengths.  However, this comes with the potential costs of extra in-memory
// copies of the payload and/or conservative maximum lengths.  Use of the
// negotiated-payload APIs can overcome these limitations to yield performance
// improvements in two important cases.  First, when the client can begin the
// negotiation before the payload is assembled (for instance concatenation of
// a client-provided header and application-provided data) payload negotiation
// can ensure that the GASNet conduit will not need to make an additional
// in-memory copy to prepend its own header, or to send from pre-registered
// memory.  Second, when the client has a need for fragmentation and
// reassembly (due to a payload exceeding the maximums) use of negotiated
// payload may permit a smaller number of fragments by taking advantage of
// transient conditions (for instance in GASNet's buffer management) that
// allow sending AMs with a larger payload than can be guaranteed in general.
//
// The basis of negotiated-payload AMs is a split-phase interface: "Prepare"
// and "Commit".  The first phase is a Prepare function to which the client
// passes an optional source buffer address, the minimum and maximum lengths
// it is willing to send, and many (but not all) of the other parameters
// normally passed when injecting an Active Message.  In this phase, GASNet
// determines how much of the payload can be sent.
//
// The return from the Prepare call provides the client with an address and a
// length.  The length is in the range defined by the minimum and maximum
// lengths.  When the client_buf argument is non-NULL, the address provided by
// the return will be exactly that value.  Otherwise the address will be a
// GASNet-allocated buffer of the indicated length, suitably aligned to hold any
// data type.
//
// It is important to note that passing NULL for the client_buf argument to a
// Prepare call requires GASNet to allocate buffer space of size no smaller
// than least_payload.  Use of gex_AM_Max{Request,Reply}{Medium,Long}() with
// the GEX_FLAG_AM_PREPARE_LEAST_ALLOC flag gives the limits on the space GASNet
// is required to allocate.  Larger values of least_payload are erroneous.
//
// Between the Prepare and the Commit calls the client is responsible for
// assembling its payload (or the prefix of the given length) at the selected
// address (potentially a no-op).  The client may send a length shorter than
// the value returned from the Prepare, for instance rounding down to some
// natural boundary.  The client may also defer until the Prepare-Commit
// interval its selection of the AM handler and arguments, which might depend
// on the address and length returned by the Prepare call (though the number
// of args must be fixed at Prepare).  In the case of a Long, the client may
// also defer selecting the destination address.  These various parameters are
// passed to the Commit function which performs the actual AM injection.
//
// It is important to note that in the interval between a Prepare and Commit,
// the client is bound by the same restrictions as in an Active Message Reply
// handler (ie all communication calls are prohibited).  Prepare/commit pairs
// do not nest.  Additionally, the Prepare returns a thread-specific object
// that must be consumed (exactly once) by a Commit in the same thread. Calls
// to Prepare are permitted in the same places as the corresponding
// fixed-payload AM injection call.
//
// Currently the semantics of the least_payload==0 case are unspecified.
// We advise avoiding that case until a later release has resolved this.
```

```
// Opaque type for AM Source Descriptor
// Used in negotiated-payload AM calls:
//   Produced by (returned from) gex_AM_Prepare*()
//   Consumed by (passed to) gex_AM_Commit*()

typedef ... gex_AM_SrcDesc_t;

// Predefined value of type gex_AM_SrcDesc_t
// Guaranteed to be zero.
// May be returned by gex_AM_Prepare*() when the GEX_FLAG_IMMEDIATE flag
// was passed, but required resources are not available.
// Must not be passed to gex_AM_Commit*() calls or the
// gex_AM_SrcDesc*() queries.

#define GEX_AM_SRCDESC_NO_OP ((gex_AM_SrcDesc_t)0)

// Query the address component of a gex_AM_SrcDesc_t
//
// Will be identical to the 'client_buf' passed to the Prepare call if that
// value was non-NULL, and otherwise will be GASNet-allocated memory suitably
// aligned to hold any data type.

void *gex_AM_SrcDescAddr(gex_AM_SrcDesc_t sd);

// Query the length component of a gex_AM_SrcDesc_t
//
// Indicates the maximum length of the buffer located at gex_AM_SrcDescAddr()
// that can be sent in the Commit call.
// Will be between the 'least_payload' and 'most_payload' passed
// to the Prepare call (inclusive).

size_t gex_AM_SrcDescSize(gex_AM_SrcDesc_t sd);

// Native implementation indicators for negotiated-payload active messages
// GASNET_NATIVE_NP_ALLOC_{REQ,REP}_{MEDIUM,LONG} symbols are defined to 1 or
// #undef to indicate whether (or not, respectively) the implementation
// of negotiated-payload AM Request/Reply Medium/Long (with a GASNet-allocated
// source buffer, i.e., initiated with client_buf == NULL) for the network
// transport of the current conduit are "native". This is a performance hint
// to clients, and does not affect correctness or normative behavior.
// The native designation implies that AM injection using these calls can avoid
// one or more payload copies relative to the corresponding fixed-payload AM
// call under the right conditions (which may be implementation dependent).
// Note that in configurations providing GASNet shared-memory bypass for AM
// to intra-nbrhd peers (activated by --enable-pshm, enabled by default),
// these only denote the behavior of the network transport (AM to peers outside
// the caller's nbrhd). The shared-memory transport for all conduits always
// provides native behavior for Medium requests and replies.

#define GASNET_NATIVE_NP_ALLOC_REQ_MEDIUM  1 or #undef
#define GASNET_NATIVE_NP_ALLOC_REP_MEDIUM  1 or #undef
#define GASNET_NATIVE_NP_ALLOC_REQ_LONG    1 or #undef
#define GASNET_NATIVE_NP_ALLOC_REP_LONG    1 or #undef

//
// gex_AM_Prepare calls
//
// RETURNS: gex_AM_SrcDesc_t
//   + An opaque scalar type (with accessors) described above
//   + This is thread-specific value
//   + This object is "consumed" by (cannot be used after) the
//     Commit call
// ARGUMENTS:
//  gex_TM_t tm, gex_Rank_t rank [REQUEST ONLY]
//   + These arguments name the destination of an AMRequest
```

```
//  gex_Token_t token [REPLY ONLY]
//   + This argument identifies (implicitly) the destination of
//     an AMReply
//  const void *client_buf
//   + If non-NULL the client is offering this buffer as a
//     source_addr
//   + If NULL, the client is requesting a GASNet-allocated source
//     buffer to populate
//  size_t least_payload
//   + This is the minimum length that the Prepare call may
//     return on success - ie the least-sized payload the
//     client is willing to send at this time.
//   + The value must not exceed the value of the
//     gex_AM_Max[...]() call with the analogous Prepare arguments
//  size_t most_payload
//   + This is the maximum length that the Prepare call may
//     return on success - ie a (not necessarily tight) upper
//     bound on the payload size the client is willing to send at
//     this time.
//   + The value must not be less than least_payload (but they may
//     be equal).
//   + The value *may* exceed the corresponding gex_AM_Max[...]().
//  void *dest_addr [LONG ONLY]
//   + If this value is non-NULL then GASNet may use this value
//     (and flags in the GEX_FLAG_PEER_SEG_* family) to guide its
//     choice of outputs (addr and size)
//   + If this value is non-NULL then the client is required to
//     pass the same value to the Commit call.
//   + May be NULL to request conservative behavior
//   + In all cases the actual dest_addr is supplied at Commit.
//  gex_Event_t *lc_opt
//   + If client_buf is NULL, this argument must also be NULL.
//   + If client_buf is non-NULL, this argument operates in the same
//     manner as the 'lc_opt' argument to the fixed-payload AM calls.
//     Between Prepare and Commit, the contents of the gex_Event_t
//     referenced by lc_opt, if any, is indeterminate.  Only after
//     return from the Commit call may such a value be used by the
//     caller.
//  gex_Flags_t flags
//   + Bitwise OR of flags valid for the corresponding
//     fixed-payload AM injection
//   + GEX_FLAG_IMMEDIATE: the Prepare call may return
//     GEX_AM_SRCDESC_NO_OP==0 if injection resources (in
//     particular a buffer of size least_payload or longer) cannot
//     be obtained.
//     The Commit-time behavior is unaffected by this flag.
//   + [UNIMPLEMENTED] GEX_FLAG_SELF_SEG_OFFSET: is prohibited
//   + [UNIMPLEMENTED] GEX_FLAG_SELF_SEG_*: these flags may only be
//     passed if client_buf is non-NULL, and assert segment disposition
//     properties for the range [client_buf..(client_buf+most_payload-1)]
//     that must be true upon entry to Prepare. If gex_AM_SrcDescAddr()
//     on the Prepare result is equal to client_buf, then the assertion
//     must remain true until after local completion is signalled via `lc_opt`.
//   + [UNIMPLEMENTED] GEX_FLAG_PEER_SEG_*: [LONG ONLY] if `dest_addr` is
//     non-NULL, these flags assert segment disposition properties for the
//     range [dest_addr..(dest_addr+most_payload-1)] that must be true upon
//     entry to Prepare and remain true until entry to the AM handler at
//     the target. If `dest_addr` is NULL at Prepare and non-NULL at Commit,
//     these flags assert segment disposition properties for the Commit-time
//     range [dest_addr..(dest_addr+nbytes-1)] that must be true upon
//     entry to Commit and remain true until entry to the AM handler at
//     the target.
//  unsigned int numargs
//   + The number of arguments to be passed to the Commit call
//
//
```

```
extern gex_AM_SrcDesc_t gex_AM_PrepareRequestMedium(
                gex_TM_t       tm,
                gex_Rank_t     rank,
                const void     *client_buf,
                size_t         least_payload,
                size_t         most_payload,
                gex_Event_t    *lc_opt,
                gex_Flags_t    flags,
                unsigned int   numargs);
extern gex_AM_SrcDesc_t gex_AM_PrepareReplyMedium(
                gex_Token_t    token,
                const void     *client_buf,
                size_t         least_payload,
                size_t         most_payload,
                gex_Event_t    *lc_opt,
                gex_Flags_t    flags,
                unsigned int   numargs);
extern gex_AM_SrcDesc_t gex_AM_PrepareRequestLong(
                gex_TM_t       tm,
                gex_Rank_t     rank,
                const void     *client_buf,
                size_t         least_payload,
                size_t         most_payload,
                void           *dest_addr,
                gex_Event_t    *lc_opt,
                gex_Flags_t    flags,
                unsigned int   numargs);
extern gex_AM_SrcDesc_t gex_AM_PrepareReplyLong(
                gex_Token_t    token,
                const void     *client_buf,
                size_t         least_payload,
                size_t         most_payload,
                void           *dest_addr,
                gex_Event_t    *lc_opt,
                gex_Flags_t    flags,
                unsigned int   numargs);

//
// gex_AM_Commit calls
//
// NOTE: Prototypes in this section are "patterns"
//   These API instantiate the "[M]" at the end of each prototype with
//   the integers 0 through gex_AM_MaxArgs(), inclusive.
//   The '[,arg0, ... ,argM-1]' then represent the arguments
//   (each of type gex_AM_Arg_t).
//   Additionally, on compilers supporting the __VA_ARG__ preprocessor feature
//   (added in C99 and C++11) the "[M]" may optionally be omitted entirely and
//   is inferred based on the argument count.
//
// RETURNS: void
// ARGUMENTS:
//  gex_AM_SrcDesc sd
//   + The value returned by the immediately preceding Prepare
//     call on this thread.
//  gex_AM_Index_t handler
//   + The index of the AM handler to run at the destination
//  size_t nbytes
//   + The client's payload length
//   + Must be in the range: [0 .. gex_AM_SrcDescSize(sd)]
//   + The base address of the source payload buffer is implicitly
//     specified by gex_AM_SrcDescAddr(sd)
//  void *dest_addr [LONG ONLY]
//   + The destination address for transfer of Long payloads
//   + If non-NULL dest_addr was passed to Prepare, this must
//     be the same value
//
```

```
extern void gex_AM_CommitRequestMedium[M](
                gex_AM_SrcDesc_t sd,
                gex_AM_Index_t   handler,
                size_t           nbytes
                [,arg0, ... ,argM-1]);
extern void gex_AM_CommitReplyMedium[M](
                gex_AM_SrcDesc_t sd,
                gex_AM_Index_t   handler,
                size_t           nbytes
                [,arg0, ... ,argM-1]);
extern void gex_AM_CommitRequestLong[M](
                gex_AM_SrcDesc_t sd,
                gex_AM_Index_t   handler,
                size_t           nbytes,
                void             *dest_addr
                [,arg0, ... ,argM-1]);
extern void gex_AM_CommitReplyLong[M](
                gex_AM_SrcDesc_t sd,
                gex_AM_Index_t   handler,
                size_t           nbytes,
                void             *dest_addr
                [,arg0, ... ,argM-1]);


//
// Extended API
//

// NOTE 1: Return value
//
//   An Extended API initiation call is a "no op" IF AND ONLY IF the value
//   GEX_FLAG_IMMEDIATE is included in the 'flags' argument AND the
//   conduit could determine that it would need to block temporarily to
//   obtain the necessary resources.  The blocking and NBI calls return a
//   non-zero value *only* in the "no op" case, while the NB calls return
//   GEX_EVENT_NO_OP.
//
//   In the "no op" case no communication has been performed and the
//   contents of the location named by the 'lc_opt' argument (if any) is
//   undefined.
//
// NOTE 2a: The 'lc_opt' argument for local completion (NBI case)
//
//   Implicit-event non-blocking Puts have an 'lc_opt' argument which
//   controls the behavior with respect to local completion.  The value can
//   be the pre-defined constants GEX_EVENT_NOW, GEX_EVENT_DEFER, or
//   GEX_EVENT_GROUP.  The NOW constant requires that the call not
//   return until the operation is locally complete.  The DEFER constant
//   permits the call to return without delaying for local completion,
//   which may occur as late as in the call which syncs (retires) the
//   operation (could be an explicit-event call if using an NBI access
//   region).  The GROUP constant allows the call to return without
//   delaying for local completion and adds the operation to the set for
//   which gex_NBI_{Test,Wait}() call may check local completion when
//   passed GEX_EC_LC.
//
// NOTE 2b: The 'lc_opt' argument for local completion (NB case)
//
//   Explicit-event non-blocking Puts have an 'lc_opt' argument which
//   controls the behavior with respect to local completion.  The value can
//   be the pre-defined constants GEX_EVENT_NOW or GEX_EVENT_DEFER, or
//   a pointer to a variable of type 'gex_Event_t'.  The NOW
//   constant requires that the call not return until the operation is
//   locally complete.  The DEFER constant permits the call to return
//   without delaying for local completion, which may occur as late as in
//   the call which syncs (retires) the returned event.  Use of a pointer
```

```
//   to a variable of type 'gex_Event_t' allows the call to return without
//   delay, and allows the client to check local completion using
//   gex_Event_{Test,Wait}*().
//
// NOTE 3: Local addressing
//
//   Let "the local endpoint" refer to the endpoint associated with 'tm'.
//
//   Let "device segment" denote a segment created using gex_Segment_Create()
//   with a 'kind' argument other than GEX_MK_HOST.
//
//   Let "in the local bound segment" mean that a given range of addresses
//   lies entirely within the range of the segment as might be determined by
//   applying gex_Segment_QueryAddr() and gex_Segment_QuerySize() to the
//   segment bound to the local endpoint.
//
//   The local address (src of a Put, dest of a Get) is interpreted and
//   constrained as follows:
//   + [UNIMPLEMENTED] In the presence of GEX_FLAG_SELF_SEG_OFFSET in 'flags'
//     the address argument is interpreted as an unsigned offset in bytes from
//     the start address of the local endpoint's (required) bound segment.
//     The memory so named must be in the local bound segment.
//   + In the absence of GEX_FLAG_SELF_SEG_OFFSET in 'flags':
//     - If the local endpoint has a bound device segment, then the address is
//       a device address and the memory so named must be in the local bound
//       segment.
//     - Otherwise the address is a host address, and the named memory is not
//       constrained to lie within the local bound segment (if any).
//
// NOTE 4: Remote addressing
//
//   Let "the remote endpoint" refer to the endpoint named by '(tm,rank)'.
//
//   Let "device segment" denote a segment created using gex_Segment_Create()
//   with a 'kind' argument other than GEX_MK_HOST.
//
//   Let "in the remote bound segment" mean that a given range of addresses
//   lies entirely within the range of the segment as might be determined from
//   the owneraddr and size properties obtained using gex_Segment_QueryBound()
//   or gex_EP_QueryBoundSegmentNB() applied to the '(tm,rank)' tuple.
//
//   The remote address (dest of a Put, src of a Get) is interpreted as
//   follows:
//   + [UNIMPLEMENTED] In the presence of GEX_FLAG_PEER_SEG_OFFSET in 'flags'
//     the address argument is interpreted as an unsigned offset in bytes from
//     the start address of the remote endpoint's bound segment.
//   + In the absence of GEX_FLAG_PEER_SEG_OFFSET in 'flags':
//     - If the bound segment of the remote endpoint is a device segment, then
//       the address is a device address.
//     - Otherwise the address is a host address.
//   In all cases, the remote memory must be in the remote bound segment.
//
// NOTE 5: Overlap
//
//   Within a single gex_RMA_* operation, if the specified source and destination
//   memory regions overlap, behavior is undefined. High-quality implementations
//   may choose to diagnose such errors.

// Put
int gex_RMA_PutBlocking(
           gex_TM_t tm,                // Names a local context ("return address")
           gex_Rank_t rank,            // Together with 'tm', names a remote context
           void *dest,                 // Remote (destination) address (or OFFSET)
           const void *src,            // Local (source) address (or OFFSET)
           size_t nbytes,              // Length of xfer
           gex_Flags_t flags);         // Flags to control this operation
```

```
int gex_RMA_PutNBI(
          gex_TM_t tm,
          gex_Rank_t rank,
          void *dest,
          const void *src,
          size_t nbytes,
          gex_Event_t *lc_opt,            // Local completion control (see above)
          gex_Flags_t flags);
gex_Event_t gex_RMA_PutNB(
          gex_TM_t tm,
          gex_Rank_t rank,
          void *dest,
          const void *src,
          size_t nbytes,
          gex_Event_t *lc_opt,
          gex_Flags_t flags);
// Get
int gex_RMA_GetBlocking( // Returns non-zero *only* in "no op" case (IMMEDIATE flag)
          gex_TM_t tm,                    // Names a local context ("return address")
          void *dest,                     // Local (destination) address (or OFFSET)
          gex_Rank_t rank,                // Together with 'tm', names a remote context
          void *src,                      // Remote (source) address (or OFFSET)
          size_t nbytes,                  // Length of xfer
          gex_Flags_t flags);             // Flags to control this operation
int gex_RMA_GetNBI( // Returns non-zero *only* in "no op" case (IMMEDIATE flag)
          gex_TM_t tm,
          void *dest,
          gex_Rank_t rank,
          void *src,
          size_t nbytes,
          gex_Flags_t flags);
gex_Event_t gex_RMA_GetNB(
          gex_TM_t tm,
          void *dest,
          gex_Rank_t rank,
          void *src,
          size_t nbytes,
          gex_Flags_t flags);
// Value-based payloads
gex_RMA_Value_t gex_RMA_GetBlockingVal(
          gex_TM_t tm,
          gex_Rank_t rank,
          void *src,
          size_t nbytes,
          gex_Flags_t flags);
int gex_RMA_PutBlockingVal(
          gex_TM_t tm,
          gex_Rank_t rank,
          void *dest,
          gex_RMA_Value_t value,
          size_t nbytes,
          gex_Flags_t flags);
int gex_RMA_PutNBIVal(
          gex_TM_t tm,
          gex_Rank_t rank,
          void *dest,
          gex_RMA_Value_t value,
          size_t nbytes,
          gex_Flags_t flags);
gex_Event_t gex_RMA_PutNBVal(
          gex_TM_t tm,
          gex_Rank_t rank,
          void *dest,
          gex_RMA_Value_t value,
          size_t nbytes,
          gex_Flags_t flags);
```

```
// NBI Access regions:
// These are interoperable with, and have the same semantics as,
// gasnet_{begin,end}_nbi_accessregion()
// flags are reserved for future use and must currently be zero.

void gex_NBI_BeginAccessRegion(gex_Flags_t flags);
gex_Event_t gex_NBI_EndAccessRegion(gex_Flags_t flags);


// Event test/wait operations
// The operation is indicated by the suffix
//  + _Test: no Poll call is made, returns zero on success, and non-zero otherwise.
//  + _Wait: Polls until success, void return
//
// In general it is not permitted to test or wait on a leaf event after
// synchronization of its corresponding root event because synchronization of a
// root event *implicitly* synchronizes any/all leaves.
// However, when using NB event array APIs gex_Event_{Test,Wait}{Some,All}()
// one may mix leaf events with their corresponding root events in the same
// array without concern for their relative order.  In other words, placement
// of a leaf event later in the array than its corresponding root event is
// not "test or wait on a leaf event after synchronization of its corresponding
// root event".

// Completion of a single NB event
// Success is defined as when the passed event is complete.
int  gex_Event_Test (gex_Event_t event);
void gex_Event_Wait (gex_Event_t event);

// Completion of an NB event array - "some"
// Success is defined as one or more events have been completed, OR
// the input array contains only GEX_EVENT_INVALID (which are otherwise ignored).
// Completed events, if any, are overwritten with GEX_EVENT_INVALID.
// These are the same semantics as gasnet_{try,wait}_syncnb_some(),
// except that "Test" does not AMPoll as "try" does.
// flags are reserved for future use and must currently be zero.
int  gex_Event_TestSome (gex_Event_t *pevent, size_t numevents, gex_Flags_t flags);
void gex_Event_WaitSome (gex_Event_t *pevent, size_t numevents, gex_Flags_t flags);

// Completion of an NB event array - "all"
// Success is defined as all passed events have been completed, OR
// the input array contains only GEX_EVENT_INVALID (which are otherwise ignored).
// Completed events, if any, are overwritten with GEX_EVENT_INVALID.
// These are the same semantics as gasnet_{try,wait}_syncnb_all(),
// except that "Test" does not AMPoll as "try" does.
// flags are reserved for future use and must currently be zero.
int  gex_Event_TestAll (gex_Event_t *pevent, size_t numevents, gex_Flags_t flags);
void gex_Event_WaitAll (gex_Event_t *pevent, size_t numevents, gex_Flags_t flags);

// Identifiers to name Event Categories (such as local completion from NBI Puts)
// TODO: will eventually include categories for collectives, VIS metadata, ...
typedef [some integer type] gex_EC_t;
#define GEX_EC_ALL   ((gex_EC_t)???)
#define GEX_EC_GET   ((gex_EC_t)???)
#define GEX_EC_PUT   ((gex_EC_t)???)
#define GEX_EC_AM    ((gex_EC_t)???)
#define GEX_EC_LC    ((gex_EC_t)???)
#define GEX_EC_RMW   ((gex_EC_t)???)

// Sync of specified subset of NBI operations
// The 'event_mask' argument is bitwise-OR of GEX_EC_* constants
// flags are reserved for future use and must currently be zero.
int  gex_NBI_Test(gex_EC_t event_mask, gex_Flags_t flags);
void gex_NBI_Wait(gex_EC_t event_mask, gex_Flags_t flags);
```

```
// Extract a leaf event from the root event
// NOTE: name is subject to change
//
// The 'root' argument must be a valid root event, such as returned by an
// NB initiation function (gex_*NB()) or gex_NBI_EndAccessRegion.
// It is permitted to be GEX_EVENT_INVALID (but not GEX_EVENT_NO_OP).
// The 'event_category' argument is an GEX_EC_<x> constant.
// It cannot be a bitwise-OR of multiple such values, nor GEX_EC_ALL.
//
// There are additional validity constraints to be documented, such as one
// cannot ask for an event that was "suppressed" by passing EVENT_NOW or EVENT_DEFER.
// Violating those constraints give undefined results (though we want a debug
// build to report the violation).
//
// For root==GEX_EVENT_INVALID, or equivalently for
// an event that has "already happened" the implementation may return
// either GEX_EVENT_INVALID or a valid event that tests as done.  The
// implementation is not constrained to pick consistently between these two
// options (and in the extreme could choose between them at random).
//
// This is a *query* and does not instantiate a new object, and so multiple
// calls with the same argument (that don't return INVALID_HANDLE) must return
// the *same* event.
gex_Event_t gex_Event_QueryLeaf(
        gex_Event_t root,
        gex_EC_t event_category);

//
// Neighborhood and Host:
//
// A "neighborhood" is defined as a set of GEX processes that can share
// memory via the GASNet PSHM feature, and is abbreviated to Nbrhd.
//
// A "host" is an abstract boundary in the system hierarchy that is guaranteed
// to be a superset of the neighborhood, but the exact definition may be
// system-specific.  Generally it encompasses processing resources associated
// with a single physical address space and OS kernel image.  When using
// GASNet-Tools from the same release, it is guaranteed that the definition
// for "host" is consistent with the following:
//   gasnett_cpu_count(), gasnett_getPhysMemSz()
// However, there is no guarantee of correspondence to gasnett_gethostname().
//
// As with all functions in the gex_System_*() namespace, the following queries
// return information about the global GASNet job, independent of any
// particular client, team or endpoint.

// Const-qualified struct type for describing a member of a neighborhood
typedef const struct {
    gex_Rank_t gex_jobrank; // the Job Rank (as defined above)
    // Reserved for future expansion and/or internal-use fields
} gex_RankInfo_t;

// Query information about the neighborhood of the calling process.
//
// All arguments are pointers to locations for outputs, each of which
// may be NULL if the caller does not need a particular value.
// info_p:
//        Receives the address of an array with elements of type
//        gex_RankInfo_t (defined above), which includes one entry
//        for each process in the neighborhood of the calling process.
//        Entries are sorted by increasing gex_jobrank.
//        The storage of this array is owned by GASNet and must not be
//        written to or free()ed.
//        High-quality implementations will store this array in shared memory
//        to reduce memory footprint.  Therefore, clients should consider using
//        it in-place to avoid creating a less-scalable copy per process.
```

```
// info_count_p:
//         Receives the number of processes in the neighborhood of the calling
//         process.  This includes the caller, and is therefore always non-zero.
// my_info_index_p:
//         Receives the 0-based index of the calling process relative to its
//         neighborhood.  In particular, the following formula holds:
//         (*info_p)[*my_info_index_p].gex_jobrank == gex_System_QueryJobRank()
//
// Semantics in a resilient build will be defined in a later release.
extern void gex_System_QueryNbrhdInfo(
            gex_RankInfo_t         **info_p,
            gex_Rank_t             *info_count_p,
            gex_Rank_t             *my_info_index_p);

// Query information about the Host of the calling process.
//
// Operates analogously to gex_System_QueryNbrhdInfo, except that instead of
// querying information about the neighborhood, this function instead queries
// information about the "host" enclosing the calling process and its
// neighborhood.
//
// Argument semantics are identical to gex_System_QueryNbrhdInfo with
// "neighborhood" replaced with "host".
extern void gex_System_QueryHostInfo(
            gex_RankInfo_t         **info_p,
            gex_Rank_t             *info_count_p,
            gex_Rank_t             *my_info_index_p);

// Query information about the sets of Neighborhoods and Hosts
//
// All arguments are pointers to locations for outputs, each of which
// may be NULL if the caller does not need a particular value.
// nbrhd_set_size_p:
//         Receives the number of neighborhoods in the job.
// nbrhd_set_rank_p:
//         Receives the 0-based rank of the caller's neighborhood within
//         the set of neighborhoods in the job (a value between 0 and
//         nbrhd_set_size-1, inclusive).
// host_set_size_p:
//         Receives the number of hosts in the job.
// host_set_rank_p:
//         Receives the 0-based rank of the caller's host within the
//         set of host in the job (a value between 0 and host_set_size-1,
//         inclusive).
//
// In a non-resilient build, the values returned by this query are constant for
// any given caller over the lifetime of the job.  Semantics in a resilient
// build will be defined in a later release.
//
// Information returned by this query is guaranteed to be self consistent:
// (where the value received into the variable referenced by "PROPERTY_p"
//  is referred to below as "PROPERTY")
//   + All callers receive identical nbrhd_set_size.
//   + Callers in the same neighborhood receive identical nbrhd_set_rank.
//   + Callers in distinct neighborhoods receive distinct nbrhd_set_rank.
//   + All callers receive identical host_set_size.
//   + Callers on the same host receive identical host_set_rank.
//   + Callers on distinct hosts receive distinct host_set_rank.
// Other than these rules, and the [0,set_size) ranges, there are no other
// guarantees as to how the ranks are assigned.

extern void gex_System_QueryMyPosition(
            gex_Rank_t *nbrhd_set_size_p,
            gex_Rank_t *nbrhd_set_rank_p,
            gex_Rank_t *host_set_size_p,
            gex_Rank_t *host_set_rank_p);
```

```
//
// Handler-safe locks (HSLs)
// Lock semantics are identical to those in GASNet-1
//

// Type for an HSL
// This type interoperable with gasnet_hsl_t
typedef {...} gex_HSL_t;

// Static-initializer for an HSL
// Synonymous with GASNET_HSL_INITIALIZER
#define GEX_HSL_INITIALIZER {...}

// The following operations on HSLs are semantically identical
// to the corresponding gasnet_hsl_* functions:
void gex_HSL_Init   (gex_HSL_t *hsl);
void gex_HSL_Destroy(gex_HSL_t *hsl);
void gex_HSL_Lock   (gex_HSL_t *hsl);
void gex_HSL_Unlock (gex_HSL_t *hsl);
int  gex_HSL_Trylock(gex_HSL_t *hsl);

//
// Common types for atomics and reductions
//

// Data types for atomics and reductions
//
// GASNet-EX defines (as preprocess-time constants) at least the following
// data types codes for use with remote atomic and reduction operations.
// These are known as the "built-in data types".
//
//     GEX Constant  C Data Type
//     ------------  -----------
// Integer types:
//     GEX_DT_I32    int32_t
//     GEX_DT_U32    uint32_t
//     GEX_DT_I64    int64_t
//     GEX_DT_U64    uint64_t
// Floating-point types:
//     GEX_DT_FLT    float
//     GEX_DT_DBL    double
//
// In addition to the built-in data types, the following is used to denote an
// opaque user-defined data type in the context of a reduction operation.
//     GEX_DT_USER
//
// It is guaranteed that all GEX_DT_* values are represented by disjoint non-zero bits.
//
// Currently, Remote Atomics support all built-in data types listed above.
// Currently, Reductions support all data types (built-in and user-defined)
// listed above.
//
// Note that GASNet-EX supports signed and unsigned exact-width integer types.
// Any mapping to types such as 'int', 'long' and 'long long' is the
// responsibility of the client.

typedef [some integer type] gex_DT_t;
#define GEX_DT_??? ((gex_DT_t)???) // For each GEX_DT_* above

// Operation codes (opcodes) for atomics and reductions
//
// GASNet-EX defines (as preprocess-time constants) at least the following
// operation codes for use with atomic and reduction operations.  Not all
// operations are valid in all contexts, as indicated below.
// See documentation for the atomic and reduction operations for more details.
```

```
//
// The following apply to the operation definitions which follow:
//   For atomics:
//     'op0' denotes the value at the target location prior to the operation
//     'op1' and 'op2' denote the value of the corresponding function arguments
//     'expr' denotes the value of the target location after the operation
//     Fetching operations always return 'op0'
//   For reductions:
//     'op0' represents the "left" (first) reduction operand
//     'op1' represents the "right" (second) reduction operand
//     'expr' denotes the value of the result of the pairwise reduction
//
// Except where otherwise noted, the expressions below are evaluated according
// to C language rules.
//
// The following are known as the "built-in operations":
// + Non-fetching Operations
//    - Binary Arithmetic Operations
//      Valid for Atomics and Reductions
//      Valid for all built-in data types
//        GEX_OP_ADD   expr = (op0 + op1)
//        GEX_OP_MULT  expr = (op0 * op1)
//        GEX_OP_MIN   expr = ((op0 < op1) ? op0 : op1)
//        GEX_OP_MAX   expr = ((op0 > op1) ? op0 : op1)
//    - Non-commutative Binary Arithmetic Operations
//      Valid only for Atomics
//      Valid for all built-in data types
//        GEX_OP_SUB   expr = (op0 - op1)
//    - Unary Arithmetic Operations
//      Valid only for Atomics
//      Valid for all built-in data types
//        GEX_OP_INC   expr = (op0 + 1)
//        GEX_OP_DEC   expr = (op0 - 1)
//    - Bit-wise Operations
//      Valid for Atomics and Reductions
//      Valid only for Integer built-in types
//        GEX_OP_AND   expr = (op0 & op1)
//        GEX_OP_OR    expr = (op0 | op1)
//        GEX_OP_XOR   expr = (op0 ^ op1)
// + Fetching Operations
//   Valid only for Atomics
//   Each GEX_OP_Fxxx performs the same operation as GEX_OP_xxx, above,
//   and is valid for the same types.
//   Additionally these operations fetch 'op0' as the result of the atomic.
//    - Binary Arithmetic Operations
//        GEX_OP_FADD
//        GEX_OP_FMULT
//        GEX_OP_FMIN
//        GEX_OP_FMAX
//    - Non-commutative Binary Arithmetic Operations
//        GEX_OP_FSUB
//    - Unary Arithmetic Operations
//        GEX_OP_FINC
//        GEX_OP_FDEC
//    - Bit-wise Operations
//        GEX_OP_FAND
//        GEX_OP_FOR
//        GEX_OP_FXOR
// + Accessor Operations
//   Valid only for Atomics
//   Valid for all built-in data types
//    - Non-fetching Accessor
//        GEX_OP_SET   expr = op1  (writes 'op1' to the target location)
//        GEX_OP_CAS   expr = ((op0 == op1) ? op2 : op0)
//                     With a guarantee to be free of spurious failures as from
//                     cache events.
```

```
//    - Fetching Accessors (fetch 'op0' as the result of the atomic)
//        GEX_OP_GET   expr = op0  (does not modify the target location)
//        GEX_OP_SWAP  expr = op1  (swaps 'op1' with the target location)
//        GEX_OP_FCAS  Fetching variant of GEX_OP_CAS
//
// NOTE: GEX_OP_CSWAP is a deprecated alias for GEX_OP_FCAS
//
// In addition to the built-in operations, the following constants are defined:
// + User-defined Operations
//   Valid only for Reductions
//   Valid for all built-in data types and GEX_DT_USER
//   The client code, not this specification, determines the operation.
//    - Commutative User-Defined Reduction Operation
//        GEX_OP_USER
//    - Non-commutative User-Defined Reduction Operation
//        GEX_OP_USER_NC
//
// It is guaranteed that all GEX_OP_* values are represented by disjoint non-zero bits.

typedef [some integer type] gex_OP_t;
#define GEX_OP_??? ((gex_OP_t)???) // For each GEX_OP_* above

// Opcode conversion
//
// The macro GEX_OP_TO_FETCHING(op) takes a non-fetching opcode as an argument
// and returns the corresponding fetching opcode.  The value of 'op' must be
// GEX_OP_SET, GEX_OP_CAS or an opcode listed under "Non-fetching Operations",
// above.  All other values return undefined results.
//
// The macro GEX_OP_TO_NONFETCHING(op) takes a fetching opcode as an argument
// and returns the corresponding non-fetching opcode.  The value of 'op' must be
// GEX_OP_SWAP, GEX_OP_FCAS or an opcode listed under "Fetching Operations",
// above.  All other values return undefined results.
//
// In addition to the natural result when applied to the arithmetic opcodes and
// (F)CAS, SWAP/SET are considered to be a fetching/non-fetching pair:
//    GEX_OP_TO_FETCHING(GEX_OP_SET)     == GEX_OP_SWAP
//    GEX_OP_TO_NONFETCHING(GEX_OP_SWAP) == GEX_OP_SET

#define GEX_OP_TO_FETCHING(op)    ???
#define GEX_OP_TO_NONFETCHING(op) ???

//-----------------------------------------------------------------------
//
// Remote Atomic Operations
// APIs in this section are provided by gasnet_ratomic.h
//

//
// Atomic Domains
//
// An "Atomic Domain" is an opaque scalar type.
//
// Just as all point-to-point RMA calls take a gex_TM_t argument, calls to
// initiate Remote Atomic operations take a gex_AD_t, where "AD" is short for
// "Atomic Domain".
//
// + Creation of an AD associates it with a specific gex_TM_t.
//
//   This association defines the memory locations which can be accessed using
//   the AD.  Only memory within the address space of a process hosting an
//   endpoint that is a member of this team may be accessed by atomic
//   operations which pass a given AD.
//
//   Currently, there is an additional constraint that target locations must
//   lie within the bound segments of the team's endpoints.
```

```
//
// + Creation of an AD associates with it one data type and a set of operations.
//
//   This permits selection of the best possible implementation which can
//   provide correct results for the given set of operations on the given data
//   type.  This is important because the best possible implementation of a
//   operation "X" may not be compatible with operation "Y".  So, this best
//   "X" can only be used when it is known that "Y" will not be used.  This
//   issue arises because a NIC may offload "X" (but not "Y") and use of a
//   CPU-based implementation of "Y" would not be coherent with the NIC
//   performing a concurrent "X" operation.
//
// + Use of an AD is conceptually tied to specific data and time.
//
//   Correct operation of gex_AD_Op*() APIs is only assured if the client code
//   can ensure that there are no other accesses to the same target locations
//   concurrent with the operations on a given AD.
//
//   The prohibition against concurrent access applies to all access by CPUs,
//   GPUs and any other hardware that references memory; and to all GASNet-EX
//   operations other than the atomic accesses defined in this section.  The
//   write by a fetching remote atomic operation to an output location on the
//   initiator is NOT an atomic access for the purposes of this prohibition.
//
//   Prohibited accesses by CPUs and GPUs include not only load/store, but
//   also any atomic operations provided by languages such as C11 and C++11,
//   by compiler intrinsics, operating system facilities, etc.
//
//   This prohibition also extends to concurrent access via multiple ADs, even
//   if created with identical arguments.  However, this specification does
//   not prohibit concurrent access to distinct (non overlapping) data using
//   distinct ADs.
//
//   GASNet-EX does not provide any mechanisms to detect violations of the
//   prohibitions described above.
//
// + Atomic Access Phases [INCOMPLETE / OPEN ISSUE]
//
//   It is the intent of this specification to permit access to the same data
//   using remote atomics and other (non-atomic) mechanisms, and to the same
//   data using multiple atomics domains.  However, such different accesses
//   must be NON-concurrent.  This separation is into what we will call
//   "atomic access phases":
//     During a given atomic access phase, any given byte in the memory of any
//     GASNet process shall NOT be accessed by more than ONE of:
//      (1) gex_AD_Op*() calls that reference that byte as part of the target
//          object.
//      (2) any means except for (1).
//     Furthermore, during a given atomic access phase, all gex_AD_Op*() calls
//     accessing a given target byte shall use the same AD object.
//
//   Note that the byte-granularity of this definition has consequences for
//   the use of union types and of type-punning, either of which may result in
//   a given byte being considered part of multiple C objects.
//
//   The means for a transition between atomic access phases has not yet been
//   fully specified.  We do NOT expect that the resolution to this open issue
//   will invalidate any interface defined in this current specification.
//   However, when implementations of remote atomics are introduced with
//   properties such as caching, it may become necessary for clients using
//   remote atomics to take additional steps to transition between atomic
//   access phases.
//
//   FOR *THIS* RELEASE we believe it is sufficient to separate atomic access
//   phases by a barrier synchronization.  However, it is necessary to ensure
//   that any GASNet-EX accesses which may conflict have been completed
```

```
//   (synced) prior to the barrier.  This includes completing all remote atomic
//   operations before a transition to non-atomic access or accesses by a
//   different atomic domain; and completing all other GASNet data-movement
//   operations (RMA, Collective, etc.) before a transition to atomic access.
//
// + Memory Ordering/Fencing/Consistency
//
//   By default calls to the gex_AD_Op*() APIs are not guaranteed to be ordered
//   with respect to other memory accesses.  However, one can request Acquire
//   or Release fencing through the use of 'flags' as described in more detail
//   with the description of gex_AD_Op*().  The definitions given below for
//   Acquire and Release are intended to be compatible with the same concepts
//   in the C11 and C++ language specifications for atomic operations.

// Opaque scalar type for Atomic Domain
typedef ... gex_AD_t;

// Pre-defined constant, guaranteed to be zero
#define GEX_AD_INVALID      ((gex_AD_t)0)

// Create an Atomic Domain
//
// This is a collective call over the team named by the 'tm' argument that
// creates an atomic domain for the operations in the 'ops' argument performed
// on data type 'dt'.
//
// The 'ad_p' is an OUT parameter that receives a reference to the
// newly created atomic domain.
//
// The 'dt' and 'ops' arguments define the type and operations.
//  + 'dt' is a value of type gex_DT_t
//  + 'ops' is a bitwise-OR of one or more GEX_OP_* constants of type gex_OP_t.
// If 'dt' and 'ops' do not define only valid combinations (as described in the
// definitions of gex_OP_t), then the behavior is undefined.
//
// The 'flags' argument provides additional control over the created domain.
//  + GEX_FLAG_AD_FAVOR_{MY_RANK,MY_NBRHD,REMOTE}
//    This family of mutually-exclusive flags are hints to influence the
//    selection of implementation to favor PERFORMANCE of accesses initiated
//    for target locations having certain locality properties.  Presence or
//    absence of these flags will never impact correctness.
//    - GEX_FLAG_AD_FAVOR_MY_RANK:
//      Favor calls with the initiating and target endpoint being the same.
//      (e.g use of GEX_FLAG_AD_MY_RANK would be legal at initiation).
//    - GEX_FLAG_AD_FAVOR_MY_NBRHD:
//      Favor calls with the initiating and target endpoints belonging to
//      processes in the same "Neighborhood", as defined previously.  (e.g.
//      use of GEX_FLAG_AD_MY_NBRHD would be legal at initiation).
//    - GEX_FLAG_AD_FAVOR_REMOTE:
//      Favor calls with the initiating and target endpoints belonging to
//      distinct Neighborhoods.
//   If a call to gex_AD_Create does not include any flag from this group, the
//   behavior is not required to correspond to any of the behaviors described
//   above.  A high-quality implementation should examine the composition of
//   'tm' and when possible favor either RANK (TM with a single member) or
//   NBRHD (TM with all members in the same Neighborhood).
//
// The 'dt', 'ops' and 'flags' parameters are each single-valued.
//
void gex_AD_Create(
            gex_AD_t                *ad_p,          // Output
            gex_TM_t                tm,             // The team
            gex_DT_t                dt,             // The data type
            gex_OP_t                ops,            // OR of operations
            gex_Flags_t             flags);         // flags
```

```
// Destroy an Atomic Domain
//
// This is a collective call over the team named at creation of the 'ad'
// argument that destroys an atomic domain.
//
// All operations initiated on the atomic domain must be complete prior to any
// rank making this call (or the behavior is undefined).  In practice, this
// means completing (syncing) all atomic operation at their initiators,
// followed by a barrier prior to calling this function.
//
// [INCOMPLETE / OPEN ISSUE]
// Once this specification includes a complete specification of atomic access
// phases, this call will provide and/all aspects of division between such
// phases which are stronger than the quiescence pre-condition.  We do NOT
// expect that the resolution to this open issue will invalidate this API's
// specification.
//
// Though this function is collective, it does not guarantee barrier
// synchronization.
//
void gex_AD_Destroy(gex_AD_t ad);

//
// Query operations on gex_AD_t
//

// Query the parameters passed when atomic domain was created

gex_Flags_t  gex_AD_QueryFlags(gex_AD_t ad);
gex_TM_t     gex_AD_QueryTM(gex_AD_t ad);
gex_DT_t     gex_AD_QueryDT(gex_AD_t ad);
gex_OP_t     gex_AD_QueryOps(gex_AD_t ad);

// Client-Data (CData) support for gex_AD_t
// These calls provide the means for the client to set and retrieve one void*
// field of client-specific data for each AD, which is NULL for a newly
// created AD.

void  gex_AD_SetCData(gex_AD_t ad, const void *val);
void* gex_AD_QueryCData(gex_AD_t ad);


//
// Remote Atomic Operations
//


// Remote atomic operations are point-to-point communication calls that
// perform read-modify-write and accessor operations on typed data (the
// "target location") in the address space of a process hosting an endpoint
// that is a member of the team passed to gex_AD_Create().
//
// These operations are guaranteed to be atomic with respect to all other
// accesses to the same target location made using the same AD, from any rank.
// When using a thread-safe endpoint this includes atomicity of concurrent
// access by multiple threads within a rank.  No other atomicity guarantees
// are provided.  [Currently all endpoints are "thread-safe" when using a
// GASNET_PAR build, and no endpoints are thread-safe otherwise.]
//
// Despite "Remote" in the name, it is explicitly permitted to apply these
// operations to the caller's own memory (and a high-quality implementation
// will optimize this case when possible).
//
// Additional semantics are described following the "Argument synopsis".
//
//
```

```
// Return value:
//
// Atomic operations are available with explicit-event (NB) or implicit-event
// (NBI) completion, with different return types:
//
//  + The gex_AD_OpNB_*() APIs return a gex_Event_t.
//    If (and only if) the GEX_FLAG_IMMEDIATE flag is passed to remote atomic
//    initiation, these calls are *permitted* to return GEX_EVENT_NO_OP to
//    indicate that no operation was initiated.  Otherwise, the return value
//    is an event to be used in calls to gex_NBI_{Test,Wait}() to check for
//    completion.  These calls may return GEX_EVENT_INVALID if the operation
//    was completed synchronously.
//
//  + The gex_AD_OpNBI_*() APIs return an integer.
//    If (and only if) the GEX_FLAG_IMMEDIATE flag is passed to remote atomic
//    initiation, these calls are *permitted* to return non-zero to indicate
//    that no operation was initiated.  Otherwise, the return value is zero
//    and a gex_NBI_{Test,Wait}() call must be used to check completion.  For
//    the opcodes GEX_OP_SET and GEX_OP_GET, one should use GEX_EC_PUT and
//    GEX_EC_GET, respectively, to check completion.  All other opcodes
//    correspond to an event category of GEX_EC_RMW.
//
// Data types and prototypes:
//   The APIs for remote atomic initiation are typed.  Therefore, descriptions
//   and prototypes below use "[DATATYPE]" to denote the tokens corresponding
//   to the "???" in each supported GEX_DT_???, and "[TYPE]" to denote the
//   corresponding C type.  There is an instance of each function (NB and NBI)
//   for each supported data type.
//   See the "Data types for atomics and reductions" section for which data
//   types are supported for remote atomics, and their corresponding C types.
//
// "Fetching":
//   Text below uses "fetching" to denote operations that write to an output
//   location ('*result_p') at the initiator (and "non-fetching" for all
//   others).
//   See the "Operation codes (opcodes) for atomics and reductions" section
//   for which opcodes are fetching vs non-fetching.
//
// Endpoints:
//   Let 'tm' denote the corresponding argument passed to gex_AD_Create().
//   The endpoint associated with 'tm' is known as the "initiating endpoint".
//   The endpoint named by (tm, tgt_rank) is known as the "target endpoint".
//
// Argument synopsis:
//   gex_AD_t       ad
//     + The Atomic Domain for this operation.
//   [TYPE] *       result_p
//     + Address (or offset) of the output location for fetching operations.
//       Ignored for non-fetching operations.
//   gex_Rank_t     tgt_rank
//     + Rank of the target location
//   void *         tgt_addr
//     + Address (or offset) of the target location
//   gex_OP_t       opcode
//     + Indicates the operation to perform atomically.
//       Operations are described with the definition of gex_OP_t.
//   [TYPE]         operand1
//     + First operand, if any.
//       Ignored if the given opcode takes no operands.
//   [TYPE]         operand2
//     + Second operand, if any.
//       Ignored if the given opcode takes fewer than two operands.
//   gex_Flags_t    flags
//     + Per-operation flags
//       A bitwise OR of zero or more of the GEX_FLAG_* constants.
//
```

```
// Semantics of gex_AD_Op*():
//
// + Successful synchronization of a fetching remote atomic operation means
//   that the local output value (at *result_p) is ready to be examined, and
//   will contain a value that was held at the target location at some time in
//   the interval between the call to the initiation function and the
//   successful completion of the synchronization.  This value will be the one
//   present at the start of the atomic operation and denoted as 'op0' in the
//   definition of the applicable opcode.
//   [THIS PARAGRAPH IS NOT INTENDED TO BE A FORMAL MODEL.
//    HOWEVER, ONE IS FORTHCOMING.]
//
// + Successful synchronization of any remote atomic operation means the
//   operation has been performed atomically (including any constituent Read
//   and Write access to the target location) and any remote atomic issued
//   subsequently by any thread on any rank with the same AD and target
//   location will observe the Write, if any (assuming no intervening updates).
//   [THIS PARAGRAPH IS NOT INTENDED TO BE A FORMAL MODEL.
//    HOWEVER, ONE IS FORTHCOMING.]
//
// + Atomicity guarantees apply only to "target locations".  They do not apply
//   to the output of a fetching operation.  Therefore, clients must check for
//   operation completion before the output value of a fetching operation can
//   safely be read (analogous to the destination of an gex_RMA_Get*()).
//   Additionally, a given 'result_p' location must not be used as the target
//   location of remote atomic operations in the same atomic access phase.
//   (see "Atomic Access Phases").
//
// + If two target objects accessed by gex_AD_Op*() overlap (partially or
//   completely) those accesses are subject to the restrictions documented in
//   "Atomic Access Phases" above.  In particular, such accesses are
//   permitted during the same atomic access phase *only* if the accessed
//   bytes exactly coincide and the calls use the same AD object.
//
// + Currently, the target location must be contained entirely within the
//   bound segment of the target endpoint (though this may eventually be
//   relaxed).
//
// + The data type associated with 'ad' and that of the gex_AD_Op*() call must
//   be equal.
//
// + The 'result_p' argument to fetching operations must be a valid pointer to
//   an object of the given type [TYPE] on the initiator.  (See also the
//   description of the [UNIMPLEMENTED] GEX_FLAG_SELF_SEG_OFFSET flag, below.)
//
// + The 'result_p' argument to non-fetching operations is ignored.
//
// + The 'tgt_rank' argument names the target endpoint.  By default, this
//   argument must be a valid rank relative to the team associated with the AD
//   at its creation.  However, in the presence of GEX_FLAG_RANK_IS_JOBRANK,
//   this argument instead names the target endpoint by a valid rank in the
//   primordial team, created by gex_Client_Init().  In this latter case the
//   named endpoint must be a member of the team associated with the AD at its
//   creation.
//
// + The 'tgt_addr' argument names the target location, which must be properly
//   aligned for its data type [TYPE] and (for any operation except
//   GEX_OP_SET) must contain an object with compatible effective type,
//   including a qualified version of [TYPE], and (for integer types only)
//   including signed or unsigned variants.  (See also the description of the
//   [UNIMPLEMENTED] GEX_FLAG_PEER_SEG_OFFSET flag, below.)
//
// + The 'opcode' argument gives the operation to be performed atomically.
//   See the "Operation codes (opcodes) for atomics and reductions" section
//   for definitions of each operation.
//
```

```
// + The 'opcode' must be a single GEX_OP_* value, not a bitwise OR of two or
//   more GEX_OP_* values.
//
// + The 'opcode' must be a member of the set of opcodes passed to
//   gex_AD_Create().
//
// + Operations on floating-point data types are not guaranteed to obey all
//   rules in the IEEE 754 standard even when the C float and double types
//   otherwise do conform.  Deviations from IEEE 754 include (at least):
//     - Operations on signalling NaNs have undefined behavior.
//     - (F)CAS *may* be performed as if on integers of the same width.
//       This could result in non-conforming behavior with quiet NaNs
//       or negative zero.
//     - MIN, MAX, FMIN and FMAX *may* be performed as if on "sign
//       and magnitude representation integers" of the same width.
//       This could result in non-conforming behavior with quiet NaNs.
//       (see https://en.wikipedia.org/wiki/IEEE_754-1985, and especially
//       the section Comparing_floating-point_numbers)
//   [THIS PARAGRAPH MAY NOT BE A COMPLETE LIST OF NON-IEEE BEHAVIORS]
//
//  + If the given opcode requires one or more operands, the 'operand1'
//    argument provides the first ('op1' in the gex_OP_t documentation).
//    Otherwise, 'operand1' is ignored.
//
//  + If the given opcode requires two operands, the 'operand2' argument
//    provides the second ('op2' in the gex_OP_t documentation).
//    Otherwise, 'operand2' is ignored.
//
//  + The 'flags' argument must either be zero, or a bitwise OR of one or more
//    of the following flags.
//     - GEX_FLAG_IMMEDIATE: the call is permitted (but not required) to
//       return a distinguishing value without initiating any communication if
//       the conduit could determine that it would need to block temporarily
//       to obtain the necessary resources.  The NBI calls return a non-zero
//       value (only) in this "no op" case, while the NB calls will return
//       GEX_EVENT_NO_OP.
//     - At most one flag from the following mutually-exclusive group:
//       - GEX_FLAG_AD_MY_RANK: asserts that the initiating endpoint and target
//         endpoint are the same endpoint.  This may allow the implementation to
//         perform the operation more efficiently.
//         The precise definition of the assertion is:
//           (tgt_rank == gex_TM_QueryRank(gex_AD_QueryTM(ad))).
//       - GEX_FLAG_AD_MY_NBRHD: asserts that the target EP belongs to
//         a process within the "Neighborhood" (defined earlier in this
//         document) of the calling process.  This may allow the
//         implementation to perform the operation more efficiently.
//     - GEX_FLAG_AD_REL: this atomic operation shall perform a "release".
//       Within the thread that initiates this operation, memory accesses by
//       the processor, issued before the initiation call, shall not be
//       reordered after that call.  Additionally, this includes accesses to
//       memory by any GASNet operations synchronized by that thread before
//       initiation.  However, there is no ordering with respect to other
//       GASNet operations.
//     - GEX_FLAG_AD_ACQ: this atomic operation shall perform an "acquire".
//       Within the thread that synchronizes this operation, memory accesses by
//       the processor, issued after the synchronization call, shall not be
//       reordered before that call.  Additionally, this includes accesses to
//       memory by any GASNet operations initiated by that thread after
//       synchronization.  However, there is no ordering with respect to other
//       GASNet operations.
//     - GEX_FLAG_RANK_IS_JOBRANK: this flag indicates that the 'tgt_rank'
//       argument is a jobrank (rank in the primordial team created by
//       gex_Client_Init()), rather than the rank in the team associated with
//       the AD at its creation.
//     - [UNIMPLEMENTED] GEX_FLAG_SELF_SEG_OFFSET: 'result_p' is to be
//       interpreted as an offset relative to the bound segment of the
```

```
//        initiating endpoint (instead of as a virtual address).
//        Ignored for non-fetching operations.
//      - [UNIMPLEMENTED] GEX_FLAG_PEER_SEG_OFFSET: 'tgt_addr' is to be
//        interpreted as an offset relative to the bound segment of the target
//        endpoint (instead of as a virtual address).
//

gex_Event_t gex_AD_OpNB_[DATATYPE](
            gex_AD_t      ad,          // The atomic domain
            [TYPE] *      result_p,    // Output location, if any, else ignored
            gex_Rank_t    tgt_rank,    // Rank of target endpoint
            void *        tgt_addr,    // Address (or OFFSET) of target location
            gex_OP_t      opcode,      // The operation (GEX_OP_*) to perform
            [TYPE]        operand1,    // First operand, if any, else ignored
            [TYPE]        operand2,    // Second operand, if any, else ignored
            gex_Flags_t   flags);      // Flags to control this operation
int gex_AD_OpNBI_[DATATYPE](
            gex_AD_t      ad,
            [TYPE] *      result_p,
            gex_Rank_t    tgt_rank,
            void *        tgt_addr,
            gex_OP_t      opcode,
            [TYPE]        operand1,
            [TYPE]        operand2,
            gex_Flags_t   flags);

// End of section describing APIs provided by gasnet_ratomic.h
//-----------------------------------------------------------------------

//
// Vector/Indexed/Strided (VIS)
//
// APIs in this section are provided by gasnet_vis.h

// This API is an updated and expanded version of the VIS prototype offered
// in GASNet-1, which is documented here: https://gasnet.lbl.gov/pubs/upc_memcpy_gasnet-2.0.pdf

// The following semantics apply to all VIS functions, superseding the above document:

// For NB variants, return type for all functions in this section is gex_Event_t.
// For NBI/Blocking variants, the return type is int which is non-zero *only* in the
// "no op" case (IMMEDIATE flag), exactly analogous to the gex_RMA_{Put,Get}*() functions.
//
// By default, local completion of all client-owned input buffers (ie payload
// buffers and metadata arrays) passed to non-blocking initiation functions
// can occur as late as operation completion, and thus must remain valid until
// that time (as in GASNet-1).
// As an exception, the metadata arrays passed to Strided variants ({src,dst}strides[] and count
   [])
// are guaranteed to be consumed synchronously before return from initiation.
// gex_VIS_*Put{NB,NBI} optionally expose local completion of data payload buffers -
// this functionality must be requested using the GEX_FLAG_ENABLE_LEAF_LC flag (see below).
//
// A future revision may expose other intermediate completion events [UNIMPLEMENTED]
//
// Within a single VIS operation, if any destination location overlaps a source location
// or another destination location, then behavior is undefined.
// Source locations are permitted to overlap with each other.
//
// The 'flags' argument must either be zero, or a bitwise OR of one or more
// of the following flags:
// - GEX_FLAG_IMMEDIATE: the call is permitted (but not required) to return a
//   distinguishing value without initiating any communication if the conduit
//   could determine that it would need to block temporarily to obtain the
//   necessary resources.  The Blocking and NBI calls return a non-zero value
//   (only) in this "no op" case, while the NB calls will return GEX_EVENT_NO_OP.
```

```
// - GEX_FLAG_ENABLE_LEAF_LC: (gex_VIS_*Put{NB,NBI} only) This flag requests
//   asynchronous local completion indication for the local data payload buffers
//   comprising the source region(s) of the VIS Put operation. Without this flag,
//   local completion behaves as GEX_EVENT_DEFER, i.e. folded into operation completion.
//   When this flag is passed to gex_VIS_*PutNBI, asynchronous local completion indication
//   behaves as specified in sec:`Extended API` for lc_opt=GEX_EVENT_GROUP.
//   When this flag is passed to gex_VIS_*PutNB, asynchronous local completion indication
//   behaves as specified in sec:`Extended API` for lc_opt=&(gex_Event_t variable).
//   In the latter case, the client should retrieve the gex_Event_t corresponding to
//   local completion by passing the root gex_Event_t returned by the Put initiation
//   call to gex_Event_QueryLeaf(), for example:
//     gex_Event_t VISput_RC = gex_VIS_VectorPutNB(..., GEX_FLAG_ENABLE_LEAF_LC);
//     gex_Event_t VISput_LC = gex_Event_QueryLeaf(VISput_RC, GEX_EC_LC);
//   The second call is only valid when GEX_FLAG_ENABLE_LEAF_LC was passed to the _VIS_*PutNB()
//   call, and otherwise has undefined behavior.

// NOTE: All of the (void *) types in this API will eventually be gex_Addr_t [UNIMPLEMENTED]

//
// Vector and Indexed Puts and Gets
//

// These operate analogously to those in the GASNet-1 prototype gasnet_{put,get}[vi]_* API

{gex_Event_t,int} gex_VIS_VectorGet{NB,NBI,Blocking}(
        gex_TM_t tm,                                  // Names a local context
        size_t dstcount, gex_Memvec_t const dstlist[], // Local destination data description
        gex_Rank_t srcrank,                           // Together with 'tm', names a remote
            context
        size_t srccount, gex_Memvec_t const srclist[], // Remote source data description
        gex_Flags_t flags);                           // Flags to control this operation
{gex_Event_t,int} gex_VIS_VectorPut{NB,NBI,Blocking}(
        gex_TM_t tm, gex_Rank_t dstrank,
        size_t dstcount, gex_Memvec_t const dstlist[],
        size_t srccount, gex_Memvec_t const srclist[],
        gex_Flags_t flags);

{gex_Event_t,int} gex_VIS_IndexedGet{NB,NBI,Blocking}(
        gex_TM_t tm,
        size_t dstcount, void * const dstlist[], size_t dstlen,
        gex_Rank_t srcrank,
        size_t srccount, void * const srclist[], size_t srclen,
        gex_Flags_t flags);
{gex_Event_t,int} gex_VIS_IndexedPut{NB,NBI,Blocking}(
        gex_TM_t tm, gex_Rank_t dstrank,
        size_t dstcount, void * const dstlist[], size_t dstlen,
        size_t srccount, void * const srclist[], size_t srclen,
        gex_Flags_t flags);

//
// Strided Puts and Gets
//

// These operate similarly to the GASNet-1 prototype gasnet_{put,get}s_* API,
// but the metadata format is changing slightly in EX.  Notable changes:
// + The stride arrays change type from (const size_t[]) to (const ptrdiff_t[])
// + The 'count[0]' datum moves to a new parameter 'elemsz', and the subsequent
//   elements 'count[1..stridelevels]' "slide down", meaning 'count' now references
//   an array with 'stridelevels' entries (down from 'stridelevels+1').
// Note that 'elemsz' need not match the "native" element size of the underlying
// datastructure, it just needs to indicate a size of contiguous data chunks
// (eg, it could be the length of an entire row of doubles stored contiguously).
// These interface changes enable the Strided interface to support more generalized
// strided data movements (specifically, transpose and reflection).
//
//
```

```
// Degenerate cases:
// * If elemsz == 0:
//   the operation is a no-op and all other arguments are ignored
// * If stridelevels == 0:
//   the operation is a contiguous copy of elemsz bytes, and the
//   srcstrides, dststrides, count arguments are all ignored
// * If any entry in count[0..stridelevels-1] == 0:
//   the operation is a no-op and tm, rank, srcaddr, dstaddr are ignored
//   ({src,dst}strides must still reference valid arrays)

{gex_Event_t,int} gex_VIS_StridedGet{NB,NBI,Blocking}(
        gex_TM_t tm,
        void *dstaddr, const ptrdiff_t dststrides[],
        gex_Rank_t srcrank,
        void *srcaddr, const ptrdiff_t srcstrides[],
        size_t elemsz, const size_t count[], size_t stridelevels,
        gex_Flags_t flags);
{gex_Event_t,int} gex_VIS_StridedPut{NB,NBI,Blocking}(
        gex_TM_t tm, gex_Rank_t dstrank,
        void *dstaddr, const ptrdiff_t dststrides[],
        void *srcaddr, const ptrdiff_t srcstrides[],
        size_t elemsz, const size_t count[], size_t stridelevels,
        gex_Flags_t flags);

// VIS Put Peer Completion [DEPRECATED]
//
// The following call "arms" a peer completion callback that will
// signal completion of the next VIS operation initiated by the current thread
// to the (possibly remote) peer endpoint. When the selected VIS data movement
// operation is complete with respect to the peer, an Active Message
// (with some restrictions defined below) is delivered to the peer endpoint.
// Currently this feature is only supported for VIS Put operations (not Gets).
//
// Argument synopsis:
//   gex_AM_Index_t  handler
//     + The AM handler index to invoke at the peer endpoint.
//   const void *    source_addr
//   size_t          nbytes
//     + The local source address and length of an optional client-provided payload
//       to be delivered in the AM notification.
//   gex_Flags_t    flags
//     + Unused in the current release, should be set to zero.
//
// + This call "arms" a peer completion handler and binds it to the next VIS operation
//   successfully initiated by the current thread. Only the next such operation is affected, after
//   which the peer completion binding for this thread is automatically "disarmed".
//   In this release, the VIS operation in question may not be passed GEX_FLAG_IMMEDIATE.
//   [this restriction may be relaxed in a future release]
// + If `handler == 0` then any previous gex_VIS_SetPeerCompletionHandler() call from
//   this thread (if any) is "disarmed" and cancelled.
// + Otherwise, `handler` specifies a 0-argument AM Medium Reply handler to invoke at the peer
//   endpoint selected by the VIS initiation call. The handler is invoked after the data
//   movement associated with the VIS operation is complete with respect to the peer process.
// + The selected AM handler must have been registered at the peer endpoint using
   gex_EP_RegisterHandlers()
//   with gex_flags == GEX_FLAG_AM_MEDIUM | GEX_FLAG_AM_REPLY and gex_nargs == 0
//   [this restriction may be relaxed in a future release]
//   and must adhere to the signature and restrictions of a 0-argument AM Medium Reply handler.
// + If `nbytes == 0`, then `source_addr` is ignored.
// + Otherwise, `nbytes` must be no greater than GEX_VIS_MAX_PEERCOMPLETION.
// + If `nbytes > 0`, the specified source memory must remain valid and unchanged starting from
   the
//   call to gex_VIS_SetPeerCompletionHandler and lasting until the earlier of either operation
   completion
//   or local completion (if enabled) of the VIS operation is signalled to the initiating rank.
// + The specified payload is delivered to the invoked AM Medium handler as usual.
```

```
// + The thread running the peer completion AM handler is guaranteed to observe the results
//   of the VIS data movement operation upon which it depends. However if it wishes to hand-off
//   completion notification to other local threads it should use normal cross-thread
//   synchronization mechanisms (including issuing a write memory barrier on most architectures)
//   to ensure other cores also observe the payload delivery.

void gex_VIS_SetPeerCompletionHandler(gex_AM_Index_t handler,
        const void *source_addr, size_t nbytes, gex_Flags_t flags);

// The largest permissible size (in bytes) for a client payload in a VIS peer completion handler.
// Guaranteed to be at least 127 bytes.

#define GEX_VIS_MAX_PEERCOMPLETION ((size_t)???)

// End of section describing APIs provided by gasnet_vis.h
//---------------------------------------------------------------------

//
// Collectives (Coll)
//
// With the exception of gex_Coll_BarrierNB(), APIs in this section are provided
// by gasnet_coll.h
//

// This API is an updated and expanded version of the collectives prototype
// offered in GASNet-1, and previously documented in docs/collective_notes.txt.
// As these APIs are fully specified and implemented, the corresponding
// portions of the GASNet-1 collectives prototype will be removed from
// gasnet_coll.h and replaced with GEX variants.  The GASNet-1 collectives API
// signatures will not be supported in future releases.

// The following semantics apply to all Coll functions:

// For NB variants, return type for all functions in this section is gex_Event_t.
// There are no NBI or Blocking variants at this time.

// All functions in this section are "Collective Calls" as defined in the
// Glossary.

// Multiple collective operations from this section may be active concurrently,
// over multiple teams or over a single team.  There is no longer an exception
// regarding gex_Coll_BarrierNB(), as was the case in an earlier release.

// GASNet-EX collectives and the GASNet-1 barrier must not operate concurrently.
// Specifically, the gasnet_barrier*() family of calls may not operate
// concurrently with any gex_Coll_*() operation (including barriers) over the
// primordial team (created by gex_Client_Init(flags=GEX_FLAG_USES_GASNET1) or
// obtained from a call to gasnet_QueryGexObjects()).
//   - Collective operations over the primordial team issued prior to a
//     GASNet-1 barrier over the same team must be complete/synchronized prior
//     to initiating the barrier.
//   - No collective call may be initiated over the primordial team between the
//     initiation and completion/synchronization of any GASNet-1 barrier over
//     the same team.
// [This restriction may be relaxed in a future release]
//
// Uses of the GASNet-1 barrier APIs which do not violate the restriction above
// are permitted in the same program as GASNet-EX collectives.

// In contrast to the UPC-influenced design of the GASNet-1 collectives, the
// GASNet-EX collectives do not support "NOSYNC" or "ALLSYNC" flags (they
// behave as if IN_MYSYNC|OUT_MYSYNC), nor single-valued address information.
// [A future release may re-introduce single-valued addressing for symmetric
// heaps via offset-based addressing]
// In this respect, the intuition one may hold from MPI-3 non-blocking
// collectives is largely applicable.
```

```
// By default, local completion of all client-owned input buffers ('src'
// arguments) passed to the collective initiation functions can occur as late
// as operation completion, and thus these buffers must remain valid until that time.
//
// A future revision may expose intermediate completion events [UNIMPLEMENTED]

// Upon operation completion (synchronization of the gex_Event_t returned at
// initiation) the following will hold:
// + Any input buffer ('src' argument) will be locally complete (analogous
//   to the source of a gex_RMA_PutNB() with GEX_EVENT_DEFER).
// + Any output buffer ('dst' argument) is ready to be examined by the thread
//   performing the sync of the gex_Event_t (analogous to the destination of
//   a gex_RMA_GetNB()).
// + Unless otherwise noted in the description of a given operation, there are
//   no guarantees regarding the state on other ranks participating in the
//   collective operation nor their associated input and output buffers.

// Unless noted explicitly, no API in this section, other than a Barrier, is
// required to synchronize the calling ranks.  However, the implementation is
// *permitted* to do so in any call in this section.

// NOTE: All of the (void *) types for source and destination buffers in these
// APIs will eventually be gex_Addr_t [UNIMPLEMENTED]


//
// Collectives Part I.  Barrier
//


// Split-phase barrier over a Team
//
// This is a collective call over the team named by the 'tm' argument that
// initiates a split-phase (non-blocking) barrier over the callers.
//
// + The return value is a root event which can be successfully synchronized
//   (return from gex_Event_Wait*() or zero return from gex_Event_Test*())
//   only after all members of the team have issued a corresponding call.
// + This call is non-blocking (may return before other team members have
//   issued a corresponding call).
// + Calls to gex_Coll_BarrierNB() are not "compatible" with calls to
//   gasnet_barrier() or gasnet_barrier_notify() for the purpose of
//   determining collective calling order.
// + The barrier operation provides the following memory ordering behaviors:
//   - Initiating a barrier operation shall perform a "release".
//     Within the thread that initiates the operation, memory accesses by the
//     processor, issued before the initiation call, shall not be reordered
//     after that call.  Additionally, this includes accesses to memory by any
//     GASNet operations synchronized by that thread before initiation.
//     However, there is no ordering with respect to other GASNet operations.
//   - Synchronizing a barrier operation shall perform an "acquire".
//     Within the thread that synchronizes the operation, memory accesses by
//     the processor, issued after the synchronization call, shall not be
//     reordered before that call.  Additionally, this includes accesses to
//     memory by any GASNet operations initiated by that thread after
//     synchronization.  However, there is no ordering with respect to other
//     GASNet operations.
//
// tm:      The call is collective over the associated team.
// flags:   Flags are reserved for future use and must currently be zero
//

gex_Event_t gex_Coll_BarrierNB(gex_TM_t tm, gex_Flags_t flags);
```

```
//
// Collectives Part II.  Data Movement
//
// The following argument descriptions are applicable to all collective data
// movement APIs in this section using arguments with these names.
//
// root:
//      The rank within 'tm' of one distinguished endpoint.  More information
//      on the distinguishing role of the root is provided with the detailed
//      description of each such collective operation.
//      This is always a single-valued parameter.
// src:
//      The local address of the caller's input buffer, if any.
//      This is not a single-valued parameter.
// dst:
//      The local address of the caller's output buffer, if any.
//      This is not a single-valued parameter.
// nbytes:
//      The length in bytes of one element of data.
//      This is a single-valued parameter.
// flags:
//      A bitwise OR of zero or more of permitted GEX_FLAG_* constants.
//      Currently no flags are defined for data-movement collective
//      operations, and the value zero should be passed.
//      However, a future release will support the "segment disposition"
//      flags [UNIMPLEMENTED].
//      Individual flags bits may or may not be single-valued, as will
//      be documented with each supported flag.

// Broadcast
//
// This operation copies 'nbytes' bytes of data starting at 'src' on rank
// 'root' of 'tm', to 'dst' on every rank within the 'tm'.
//
// The value of 'src' is ignored on all ranks other than 'root'.
//
// On the 'root' rank, the data is copied from 'src' to 'dst' except in the
// case these pointers are equal.  However, any other overlap between 'src'
// and 'dst' buffers on the root rank yields undefined behavior.

gex_Event_t gex_Coll_BroadcastNB(
            gex_TM_t        tm,             // The team
            gex_Rank_t      root,           // Root rank (single-valued)
            void *          dst             // Destination (all ranks)
            const void *    src,            // Source (root rank only)
            size_t          nbytes,         // Length of data (single-valued)
            gex_Flags_t     flags);         // Flags (partially single-valued)


//
// Collectives Part III.  Computational
//

// User-Defined Reduction Operations
//
// GASNet provides a set of useful built-in reduction operations.  These
// should be favored whenever possible in performance-critical reductions,
// because using a built-in operator is generally a prerequisite to leveraging
// hardware-offload support for reductions which is available in some network
// hardware.  However for situations where none of the provided built-in
// operations fit client requirements, GASNet also allows clients to provide
// code for their own reduction operation.
//
// Reduction operations which do not correspond to a built-in opcode
// (GEX_OP_*) constant are supported by passing GEX_OP_USER or GEX_OP_USER_NC
// as the 'op' when initiating a reduction.
```

```
// + GEX_OP_USER: denotes a user-defined operation that is both
//   associative and commutative.
// + GEX_OP_USER_NC: denotes a user-defined operation that is
//   associative but NOT commutative. [UNIMPLEMENTED]
//
// The implementation will invoke the user-provided function an unspecified
// number of times to perform the user's operation on a pair of vectors of
// operands.
//
// The user-defined reduction operation is passed to the initiation call using
// a function pointer with type gex_Coll_ReduceFn_t:

    typedef void (*gex_Coll_ReduceFn_t)(
            const void * arg1,         // "Left" operands
            void *       arg2_and_out, // "Right" operands and result
            size_t       count,        // Operand count
            const void * cdata);       // Client-data

// These arguments are defined as follows, with additional semantics below:
//
//   arg1:
//     This is a pointer to memory containing 'count' consecutive operands.
//     These may be caller-provided input values or intermediate results.
//     In the case of a non-commutative operation, these are the operands on
//     the "left-hand side" of the nominal operator.
//     The reduction operation is not permitted to write to this memory.
//   arg2_and_out:
//     This is a pointer to memory containing 'count' consecutive operands.
//     These may be caller-provided input values or intermediate results.
//     In the case of a non-commutative operation, these are the operands on
//     the "right-hand side" of the nominal operator.
//     The reduction operation must write the result(s) to this memory, as
//     described below.
//   count:
//     This is the number of "fields" on which to perform the reduction, and
//     thus the length of the accessible memory at 'arg1' and 'arg2_and_out'
//     is equal to 'count' times the size of each element (passed as 'dt_sz'
//     at initiation of the reduction).
//     Note that this argument may take on any positive value, which may be
//     either smaller or larger than the 'dt_cnt' passed at initiation of
//     the reduction.
//   cdata:
//     This is the value of the 'user_cdata' argument passed locally at
//     initiation of the reduction, and is intended to assist in implementing
//     more than a single data type and/or operation with a common C function.
//
// The function implementing a user-defined reduction operation:
//
// + May use the 'cdata' argument to receive information (such as the
//   operation or data type) not provided by the other arguments.
// + May assume 'arg1' and 'arg2_and_out' do not overlap each other.  However,
//   they can overlap the 'src' buffer (and 'dst' buffer, if any) passed at
//   operation initiation.
// + May perform the element-wise operations in any order, and parallel
//   computation is explicitly permitted.
// + Shall interpret the 'arg1' and 'arg2_and_out' as arrays of length 'count'
//   with an element type corresponding to the data type passed at initiation
//   of the reduction.
// + Shall apply the desired operation element-wise to each of the 'count'
//   pairs of operands, storing the result in the location from which the
//   second (right-hand) operand is retrieved.  Here "element-wise"
//   application can be expressed in pseudo-code as follows, using 'T' to
//   denote the C data type and '(+)' to denote the operator:
//     T* x = (T*)arg1;
//     T* y = (T*)arg2_and_out;
//     For all i in [0..count) do y[i] = x[i] (+) y[i];
```

```
// + Shall not assume 'count' is equal to the 'dt_cnt' passed at operation
//   initiation, since the implementation is free to process the 'dt_cnt'
//   elements passed in at initiation in smaller groups, or to group more
//   than 'dt_cnt' pairs of operands into a single call to the user's
//   function.
// + Shall not block pending any condition the satisfaction of which is
//   dependent on progress in GASNet (whether local or global).
// + Shall not make any GASNet calls other than those enumerated as permitted
//   while holding a handler-safe lock, in the section "Calls from restricted
//   context"
// + Shall not assume that the executing thread was created by the client.
// + Shall allow for the possibility that the implementation invokes the
//   function concurrent with itself, even if the client has not spawned
//   threads.  Use of handler-safe locks or GASNet-Tools atomics are
//   recommended mechanisms to deal with any access to global/persistent
//   state.

// User-Defined Data Types
//
// Reductions on types without a corresponding built-in data type (GEX_DT_*)
// constant are supported by passing GEX_DT_USER as the 'dt' when initiating a
// reduction.  In this case data elements are treated as indivisible byte
// sequences with length given as 'dt_sz' at initiation of the reduction.
// Reduction operations passing GEX_DT_USER for the data type must pass either
// GEX_OP_USER or GEX_OP_USER_NC for the operation.

// Limitations for Built-in Data Types
//
// + Operations on floating-point data types are not guaranteed to obey all
//   rules in the IEEE 754 standard even when the C float and double types
//   otherwise do conform.  Deviations from IEEE 754 include (at least):
//     - Operations on signalling NaNs have undefined behavior.
//     - MIN and MAX *may* be performed as if on "sign and magnitude
//       representation integers" of the same width, resulting in
//       non-conforming behavior with quiet NaNs.
//       (See https://en.wikipedia.org/wiki/IEEE_754-1985, and especially
//       the section "Comparing_floating-point_numbers")
//   [THIS PARAGRAPH MAY NOT BE A COMPLETE LIST OF NON-IEEE BEHAVIORS]

//
// The following argument descriptions are applicable to all computational
// collective APIs in this section using arguments with these names.
//
// src:
//      The local address of the caller's input buffer.
//      This is not a single-valued parameter.
// dst:
//      The local address of the caller's output buffer, if any.
//      This is not a single-valued parameter.
// dt:
//      The data type for the reduction operation.
//      Must be a GEX_DT_* constant documented as valid for Reductions.
//      This is a single-valued parameter.
// dt_sz:
//      The length in bytes of one element of data.
//      When 'dt' is a built-in type, this value must be the size in bytes of
//      the corresponding built-in C type.  When 'dt' is GEX_DT_USER, this
//      value must be the size in bytes of the user-defined data type.
//      This length must be non-zero.
//      This is a single-valued parameter.
// dt_cnt:
//      The per-rank count of data elements to reduce (not a length in bytes).
//      This count must be non-zero.
//      This is a single-valued parameter.
// op:
//      The opcode, of type gex_OP_t, naming the reduction operator.
```

```
//      Must be a GEX_OP_* constant documented as valid for Reductions.
//      This is a single-valued parameter.
// user_op, user_cdata:
//      If 'op' is neither GEX_OP_USER nor GEX_OP_USER_NC, then these two
//      arguments are ignored.  Otherwise 'user_op' is a local function
//      pointer of type gex_Coll_ReduceFn_t (described above), and
//      'user_cdata' is a client data pointer to be passed to each local
//      invocation of 'user_op'.  The 'user_cdata' is treated as opaque by the
//      implementation and therefore is not required to be a pointer to valid
//      memory.  In particular, it may be NULL.
// flags:
//      A bitwise OR of zero or more of permitted GEX_FLAG_* constants.
//      Currently no flags are defined for computational collective
//      operations, and the value zero should be passed.
//      However, a future release will support the "segment disposition"
//      flags [UNIMPLEMENTED].
//      Individual flags bits may or may not be single-valued, as will
//      be documented with each supported flag.

// Common Semantics
//
// All reductions are performed via an unspecified pattern of applications of
// the operator to pairs of operands, under the assumptions that (1) all
// operations are mathematically associative and (2) operations other than
// GEX_OP_USER_NC are mathematically commutative.
//
// The implementation is not required to take measures to accommodate any
// divergence (for instance of IEEE floating-point arithmetic) from the
// assumptions in the preceding paragraph.  Specifically, in the presence of
// such divergence, the implementation is not required to provide equality of
// the results of calls with mathematically equivalent arguments; neither
// between distinct calls in the same execution, nor between the same call in
// distinct executions.  However, high-quality implementations will provide
// reproducibility among calls with the same parameters within a single
// execution (e.g. by applying the operation in a deterministic order).
//
// The implementation is not required to preserve the vector of 'dt_cnt'
// elements as an indivisible unit.  It is permitted not only to break the
// vector into shorter ones, but may also concatenate multiple vectors to
// lessen the number of calls to a user-defined reduction operator.
//
// This specification does not require that a user-defined function applies
// the semantically equivalent operation to every pair of inputs.  Nothing in
// this specification prohibits passing a different user-defined operator on
// each caller, nor does it prohibit a user-defined operator from applying a
// different operation depending on an element's location in the argument
// vectors.  However, both behaviors are strongly discouraged.  Since this
// specification explicitly permits the implementation freedom in the order of
// reductions in both rank and vector dimensions, either of these behaviors
// will result in unpredictable output values.  Nothing in this paragraph is
// intended to prohibit, or discourage use of, user-defined operations with
// behaviors which depend on characteristics encoded in a user-defined data
// type (which may include position in the rank or vector dimensions); the
// intent is to discourage operators which *infer* such position information.


// Reduction to one
//
// This is a collective call over the team named by the 'tm' argument that
// initiates a non-blocking reduction applying the operation denoted by 'op'
// repeatedly to reduce a collection of operands of type denoted by 'dt'.
// Each member of 'tm' provides a 'src' vector of length 'dt_cnt' (in
// elements), and the elements are reduced element-wise such that the i'th
// element of the output vector is the reduction over the i'th elements of the
// 'src' vectors of all team members.  The result is written to the 'dst' of
// one 'root' rank.
```

```
//
// Using `(+)` to represent the nominal reduction operator, and `src_i[j]` to
// denote the i'th element of the 'src' vector passed by rank 'j', the result
// produced on the 'root' rank can be expressed as:
//     dst_i = src_i[0] (+) src_i[1] ... (+) src_i[N-1]
// where `N = gex_TM_QuerySize(tm)`.
//
// This call is non-blocking (may return before other team members have issued
// a corresponding call).
//
// On the 'root' rank the 'dst' buffer has length in bytes of 'dt_sz * dt_cnt'.
// The value of 'dst' is ignored on all other ranks.
//
// On all ranks the 'src' buffer has length in bytes of 'dt_sz * dt_cnt'.
//
// On the 'root' rank, it is permitted that 'src' and 'dst' be equal.
// However, any other overlap between 'src' and 'dst' buffers on the root rank
// yields undefined behavior.
//
// LIMITATIONS of the current release:
//  + The current implementation may limit `dt_sz` for user-defined types to as
//    little as 32KB bytes in some configurations and with default parameters.
//    The precise limit depends on the network, the sizes of the job and team,
//    and the size of the team's collective scratch space.

gex_Event_t gex_Coll_ReduceToOneNB(
            gex_TM_t             tm,            // The team
            gex_Rank_t           root,          // Root rank (single-valued)
            void *               dst,           // NOT single-valued
            const void *         src,           // NOT single-valued
            gex_DT_t             dt,            // Data type (single-valued)
            size_t               dt_sz,         // Data type size (single-valued)
            size_t               dt_cnt,        // Element count (single-valued)
            gex_OP_t             op,            // Operation (single-valued)
            gex_Coll_ReduceFn_t  user_op,       // NOT single-valued
            void *               user_cdata,    // NOT single-valued
            gex_Flags_t          flags);        // Flags (partially single-valued)

// Reduction to all
//
// This is a collective call over the team named by the 'tm' argument that
// initiates a non-blocking reduction applying the operation denoted by 'op'
// repeatedly to reduce a collection of operands of type denoted by 'dt'.
// Each member of 'tm' provides a 'src' vector of length 'dt_cnt' (in
// elements), and the elements are reduced element-wise such that the i'th
// element of the output vector is the reduction over the i'th elements of the
// 'src' vectors of all team members.  The result is written to the 'dst' of
// all ranks.
//
// The definition of the element-wise reduction is the same as was given above
// for gex_Coll_ReduceToOneNB().
//
// This call produces an output in the 'dst' buffer of all ranks.  However,
// the implementation is free to apply associativity (and commutativity for
// operators other than GEX_OP_USER_NC) *differently* in producing the
// multiple outputs.  Therefore, when the operator differs from the assumed
// mathematical properties, the results on different ranks might not be
// identical.
//
// The 'dst' and 'src' buffers have length in bytes of 'dt_sz * dt_cnt'.
//
// It is permitted that 'src' and 'dst' be equal pairwise either on every rank,
// or on none of them.  Any other overlap between 'src' and 'dst' buffers
// yields undefined behavior.  This includes any case in which 'src' and 'dst'
// are equal on at least one rank, but less than all ranks in the team (though
// this last restriction may be relaxed in a future release).
```

```
//
// LIMITATIONS of the current release:
//  + The current implementation may limit `dt_sz` for user-defined types to as
//    little as 32KB bytes in some configurations and with default parameters.
//    The precise limit depends on the network and sizes of the job and team.

gex_Event_t gex_Coll_ReduceToAllNB(
            gex_TM_t             tm,           // The team
            void *               dst,          // NOT single-valued
            const void *         src,          // NOT single-valued
            gex_DT_t             dt,           // Data type (single-valued)
            size_t               dt_sz,        // Data type size (single-valued)
            size_t               dt_cnt,       // Element count (single-valued)
            gex_OP_t             op,           // Operation (single-valued)
            gex_Coll_ReduceFn_t user_op,       // NOT single-valued
            void *               user_cdata,   // NOT single-valued
            gex_Flags_t          flags);       // Flags (partially single-valued)


// End of section describing APIs provided by gasnet_coll.h
//---------------------------------------------------------------------

//
// Memory Kinds (Device memory support)
//

// GASNet-EX features support for communication involving memory segments
// which are associated with various accellerator devices, e.g. HBM onboard GPUs.
//
// The API for this support is currently described in the following document:
//
//   GASNet-EX API: Memory Kinds
//
// which should be available in memory_kinds.pdf as a sibling to this file.

// Implementation status of Memory Kinds support is described in the document:
//
//   GASNet-EX Memory Kinds: Implementation Status
//
// which should be available in memory_kinds_implementation.md as a sibling to this file.

// End of "Memory Kinds" section
//---------------------------------------------------------------------

// vim: syntax=c
```

GASNet-EX API: Memory Kinds

Revision 2024.5.0

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, and AMD GPUs as demonstrated in GASNet-EX 2021.9.0. 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.

Copyright

Legal Disclaimer

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.
    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 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!

GASNet Specification

Version 1.8.1
Released November 2nd, 2006 (corrections: Aug 31st, 2017)
Printed 31 August 2017

Editor: Dan Bonachea gasnet-devel@lbl.gov
http://gasnet.lbl.gov

This the GASNet specification, version 1.8.1.

Selected portions adapted from:

- *A. Mainwaring and D. Culler, "Active Message Applications Programming Interface and Communication Subsystem Organization", U.C. Berkeley Computer Science Technical Report, 1996.*
- *D. Culler et al., "Generic Active Message Interface Specification v1.1", U.C. Berkeley Computer Science Technical Report, Feb, 1995.*

Published by LBNL and U.C. Berkeley

1 Introduction

1.1 Scope

This GASNet specification describes a network-independent and language-independent high-performance communication interface intended for use in implementing the runtime system for global address space languages (such as UPC or Titanium). GASNet stands for "**G**lobal-**A**ddress **S**pace **Net**working".

1.2 Organization

The interface is divided into 2 layers - the GASNet core API and the GASNet extended API:

- The extended API is a richly expressive and flexible interface that provides medium and high-level operations on remote memory and collective operations (basically anything that we could imagine being implemented using hardware support on some NIC's).
- The core API is a narrow interface based on the Active Messages paradigm, which is general enough to implement everything in the extended API.

The core API is the minimum interface that must be implemented on each network when porting to a new system, and we provide a network-independent reference implementation of the extended API which is written purely in terms of the core API to ease porting and quick prototyping. Implementors for NIC's that provide some hardware support for higher-level messaging operations (e.g. support for servicing remote reads/writes on the NIC without involving the main CPU) are encouraged to also implement an appropriate subset of the extended API directly on the network of interest (bypassing the core API) to achieve maximal performance for those operations (but this is an optimization and is not required to have a working system). Most clients will use calls to the extended API functions to implement the bulk of their communication work (thereby ensuring optimal performance across platforms). However the client is also permitted to use the core active message interface to implement non-trivial language-specific or compiler-specific communication operations which would not be appropriate in a language-independent API (e.g. implementing distributed language-level locks, distributed garbage collection, collective memory allocation, etc.).

Note the extended API interface is meant primarily as a low-level compilation target, not a library for hand-written code - as such, the goals of expressiveness and performance generally take precedence over readability and minimality.

1.3 Conventions

- All GASNet entry points are lower-case identifiers with the prefix `gasnet_`
- All constants are upper-case and preceded with the prefix `GASNET_`
- Clients access the GASNet interface by including the header '`gasnet.h`' and linking the appropriate library
- Except where otherwise noted, any of the operations in the GASNet interface could be implemented using macros or inline functions in an actual implementation - they are specified using function declaration syntax below to make the types clear, and all correct client code must type check using the definitions below. In no case should client code assume it can create a "function pointer" to any of these operations, or invoke operations having void return type from within expression context. Any macro implementations will ensure that arguments are evaluated exactly once.
- Implementation-specific values in declarations are indicated using "???"
- Sections marked "Implementor's note" are recommendations to implementors and are not part of the specification

1.4 Definitions

- **node** - An OS-level process which returns from `gasnet_init()`, and its associated local memory space and system resources. The basic unit of control when interfacing with GASNet.
- **thread** - A single thread of control within a GASNet node, which possibly shares a virtual memory space and OS-level process-id with other threads in the node. Clients which may concurrently call GASNet from more than a single thread must compile to the multi-threaded version of the GASNet library. Except where otherwise noted, GASNet makes no distinction between the threads within a multi-threaded node, and all control functions (e.g. barriers) should be executed by a single thread on the node on behalf of all local threads.
- **job** - The collection of nodes making up a parallel execution environment. Nodes often correspond to physical, architectural units, but this need not be the case (e.g. nodes may share a physical CPU/memory/NIC in multiprogrammed systems with sufficient sharable resources - note that some GASNet implementations may limit the number nodes which can run concurrently on a single system based on the number of physical network interfaces)

1.5 Configuration of GASNet

Client code must `#define` exactly one of `GASNET_PAR`, `GASNET_PARSYNC` or `GASNET_SEQ` when compiling the GASNet library and the client code (before including '`gasnet.h`') to indicate the threading environment.

`GASNET_PAR`
: The most general configuration. Indicates a fully multi-threaded and thread-safe environment - the client may call GASNet concurrently from more than one thread. The exact threading system in use is system-specific, although for obvious reasons both GASNet and the client code must agree on the threading system - unless otherwise noted, the default mechanism is POSIX threads.

`GASNET_PARSYNC`
: Indicates a multi-threaded but non-concurrent (non- threadsafe) GASNet environment, where multiple client threads may call GASNet, but their accesses to GASNet are fully serialized (e.g. by some level of synchronization above the GASNet interface). GASNet may safely assume that it will never be called from more than one client thread *concurrently* (and the client must ensure this property holds). Client code must still use GASNet No-Interrupt Sections and Handler-Safe Locks to ensure correct operation.

`GASNET_SEQ`
: Indicates a single-threaded, non-threadsafe environment. GASNet may safely assume that it will only ever be called from one unique client thread. Client code must still use GASNet No-Interrupt Sections and Handler-Safe Locks to ensure correct operation.

Implementor's Note:

- We may be able to make GASNet implementations independent of the threading system by having the client provide a few callback functions (e.g. mutex create/lock/unlock, thread create, threadid query and thread-local- data set/get)
- change the name of gasnet_init based on which mode is selected to ensure correct version is linked
- An implementation of `GASNET_PAR` is sufficient to handle all the configurations - the other configurations just permit certain useful optimizations (such as removing unnecessary locking in the library)
- Interrupt-driven implementations of `GASNET_SEQ` and `GASNET_PARSYNC` using signals must be prepared to handle the case where the thread responding to the signal may not be the thread currently inside a GASNet call. They may also need to use a private lock during HSL release to prevent multiple threads from polling simultaneously

1.6 Errors

Many GASNet core functions return 0 on success (`GASNET_OK`), or else they return errors from the following list, as specified by each function:

```
GASNET_OK = 0 (no error)
GASNET_ERR_RESOURCE
GASNET_ERR_BAD_ARG
GASNET_ERR_NOT_INIT
GASNET_ERR_BARRIER_MISMATCH
GASNET_ERR_NOT_READY
```

Except where otherwise noted, errors that occur during a call to the extended API are fatal.

Many of the core API functions will return `GASNET_ERR_RESOURCE` to indicate a generic failure in the hardware or communications system, `GASNET_ERR_BAD_ARG` to indicate an illegal client argument, or `GASNET_ERR_NOT_INIT` to indicate that `gasnet_attach()` has not been called.

If any node of a GASNet job crashes, aborts, or suffers a fatal hardware error, GASNet should make every attempt to ensure that the remaining nodes of the job are terminated in a timely manner to prevent creation of orphaned processes.

1.6.1 gasnet_ErrorName, gasnet_ErrorDesc

`const char * gasnet_ErrorName` (*int* *errval*)
`const char * gasnet_ErrorDesc` (*int* *errval*)

`gasnet_ErrorName()` and `gasnet_ErrorDesc()` convert the GASNet error number *errval* into a string containing the name or description (respectively) of the given error number. The client must not modify the string returned.

1.7 GASNet Types

`gasnet_node_t`
unsigned integer type representing a unique 0-based node index

`gasnet_handle_t`
an opaque type representing a non-blocking operation in-progress initiated using the extended API

`gasnet_handler_t`
an unsigned integer type representing an index into the core API AM handler table

`gasnet_handlerarg_t`
a 32-bit signed integer type which is used to express the user-provided arguments to all AM handlers. Platforms lacking a native 32-bit type may define this to a 64-bit type, but only the lower 32-bits are transmitted during an AM message send (and sign-extended on the receiver).

`gasnet_token_t`
an opaque type passed to core API handlers which may be used to query message information

`gasnet_register_value_t`
the largest unsigned integer type that can fit entirely in a single CPU register for the current architecture and ABI. `SIZEOF_GASNET_REGISTER_VALUE_T` is a preprocess-time literal integer constant (i.e. not `sizeof()`) indicating the size of this type in bytes

`gasnet_handlerentry_t`
struct type used to negotiate handler registration in `gasnet_attach()`

1.8 Compile-time constants

`GASNET_SPEC_VERSION_MAJOR`
`GASNET_SPEC_VERSION_MINOR`

Integral values corresponding to the major and minor version numbers of the GASNet specification version adhered to by a particular implementation. The minor version is incremented whenever new functionality is added to the specification without breaking backward compatibility. The major version is incremented whenever specification changes require breaking backward compatibility. The title page of this document provides the specification version corresponding to this version of the specification.

`GASNET_RELEASE_VERSION_MAJOR`
`GASNET_RELEASE_VERSION_MINOR`
`GASNET_RELEASE_VERSION_PATCH`

Integral values corresponding to the major, minor and patch version numbers of the release identifiers corresponding to the packaging on an implementation of GASNet. The significance of these values is implementation-defined.

`GASNET_VERSION (deprecated)`

equivalent to GASNET_SPEC_VERSION_MAJOR

`GASNET_CONFIG_STRING`

a string representing any of the relevant GASNet compile-time configuration settings that can be compared using string compare to verify version compatibility. The string is also embedded into the library itself such that it can be scanned for within a binary executable which is statically linked with GASNet.

`GASNET_MAXNODES`

an integer representing the maximum number of nodes supported in a single GASNet job. This value must be representable as a `gasnet_node_t`.

`GASNET_ALIGNED_SEGMENTS`

defined by the GASNet implementation to the value 1 if `gasnet_attach()` guarantees that the remote-access memory segment will be aligned at the same virtual address on all nodes. Defined to 0 otherwise.

`GASNET_PAGESIZE`

a preprocessor constant integer which provides the memory granularity size used for various GASNet parameters which are required to be page-aligned. On many systems this will be the system page size.

1.9 General notes

- All GASNet functions (in the extended *and* core API) support loopback (i.e. a node sending a get or active message to itself), and all functions will still work in the case of single-node jobs (e.g. barriers are basically no-ops in that case)
- GASNet will ensure that stdout/stderr are correctly propagated in a system-specific way (e.g. to the spawning console or possibly to a file or set of files). No guarantees are made about propagation of stdin, although some implementations may choose to deal with this.
- GASNet makes no guarantees about the propagation of external signals across a job - however, see comments in `gasnet_exit`

2 Core API

The core API consists of:

- A job control interface for bootstrapping, job termination and job environment queries
- The active messaging interface for implementing requests, replies and handlers
- An interface which provides handler signal-safety and atomicity control (No-Interrupt Sections and Handler-Safe Locks)

2.1 Job Control Interface

Job startup in GASNet is a two-step process. GASNet programs should start by calling gasnet_init() as the first statement in their main() function, which bootstraps the nodes and establishes command-line arguments and the job environment. All nodes then call the gasnet_attach() function to initialize the network and register shared memory segments.

GASNet initialization may register some UNIX signal handlers (e.g. to support interrupt-based implementations or aggressive segment registration policies). Client code which registers signal handlers must be careful not to preempt any GASNet-registered signal handlers (even for seemingly fatal signals such as `SIGABRT`) - the only signal which the client may always safely catch is `SIGQUIT`.

Any GASNet library implementation can be built in one of the following three configurations, which affects the behavior of remote-access memory segment registration during gasnet_attach(). The `gasnet.h` header file will define the appropriate preprocessor symbol to indicate which configuration is active.

`GASNET_SEGMENT_FAST`
: The remote-access memory segment is limited to an implementation-defined "reasonable" size, and optimized in an implementation-specific way to provide the fastest possible remote accesses. The maximum segment size may be queried using `gasnet_getMaxLocalSegmentSize()`.

`GASNET_SEGMENT_LARGE`
: This configuration allows clients with larger shared data requirements to register a larger remote-access memory segment, possibly at some cost in the efficiency of remote accesses. The maximum segment size may be queried using `gasnet_getMaxLocalSegmentSize()`, and should be comparable to the maximum total data size allowed for processes on the given system.

`GASNET_SEGMENT_EVERYTHING`
: The entire virtual memory space of each process is made available for remote access, in a way such that any memory access that would succeed when executed locally by this node would also succeed if executed by other nodes remotely. This can be used by clients which need to make the entire memory heap, stack and static data areas available for remote access.

Implementor's Note:

- The maximum segment size for `GASNET_SEGMENT_FAST` on many implementations is likely to to be limited by factors such as the amount of pinnable physical memory currently available in the system, and the access range of the NIC hardware.
- `GASNET_SEGMENT_EVERYTHING` support can trivially be provided by implementing all the remote-access operations and long AM messages using core API medium messages, such that all data accesses are actually executed by the local host processor. However, implementors are encouraged to investigate higher-performance alternatives whenever possible.
- On systems requiring pinned segments, `GASNET_SEGMENT_LARGE` can be implemented using dynamic pinning schemes (possibly with caching to amortize rendezvous and pinning costs) or combinations of direct remote accesses and AM-based accesses.

2.1.1 gasnet_init

`int gasnet_init` (*int* ****argc***, *char* ******argv***)

Bootstraps a GASNet job and performs any system-specific setup required.

Called by all GASNet-based applications upon startup to bootstrap the nodes, before any other processing takes place. Must be called before any calls to any other functions in this specification, and before any investigation of the command-line parameters passed to the program in *argc/argv*, which may be modified or augmented by this call. The semantics of any code executing before the call to `gasnet_init()` is implementation-specific (for example, it is undefined whether `stdin/stdout/stderr` are functional, or even how many nodes will run that code).

Upon return from `gasnet_init()`, all the nodes of the job will be running, stdout/stderr will be functional, and the basic job environment will be established, however the primary network resources may not yet have been initialized. The following GASNet functions are the only ones that may be called between `gasnet_init()` and `gasnet_attach()`:

```
gasnet_mynode()
gasnet_nodes()
gasnet_getMaxLocalSegmentSize()
gasnet_getMaxGlobalSegmentSize()
gasnet_getenv()
gasnet_exit()
```

All other GASNet calls are prohibited until after a successful `gasnet_attach()`.

`gasnet_init()` may fail with a fatal error and implementation-defined message if the nodes of the job cannot be successfully bootstrapped. It also may return an error code such as `GASNET_ERR_RESOURCE` to indicate there was a problem acquiring network or system resources. Otherwise, it returns `GASNET_OK` to indicate success. May only be called once during a process lifetime, subsequent calls will return an error.

2.1.2 gasnet_attach

```
typedef struct {
  gasnet_handler_t index; // == 0 for don't care
  void (*fnptr)();
} gasnet_handlerentry_t;
```

`int gasnet_attach` (*gasnet_handlerentry_t* ****table***, *int* ***numentries***,
uintptr_t ***segsize***, *uintptr_t* ***minheapoffset***)

Initializes the GASNet network system and performs any system-specific setup required.

table is an array of *numentries* gasnet_handlerentry_t elements used for registering active-message handlers provided by the client code. Clients that never explicitly call the active-message request functions in the core API need not register any handlers, and may pass a `NULL` pointer for *table*. Clients wishing to register some handlers should fill in *table* with function pointers and the desired handler index (or index 0 for "don't-care") - note that handlers 0..127 are reserved for GASNet internal use, and handlers 128..255 are available for client-provided handlers. Once `gasnet_attach()` returns, any "don't care" handler indexes in the table will be modified in place to reflect the handler index assigned for each handler - the assignment algorithm is deterministic: passing the same handler table on each node will guarantee an identical resulting assignment on each node. Handler function prototypes should match the prototypes described in the Active Message Interface section.

segsize and *minheapoffset* are used to communicate the desired size and location of the remote-access memory data segment for the local node that will be used for all remote accesses (i.e. using the data transfer functions of the extended API) or as the target of any Long active-messages in the core API. The client passes the desired size of this area in bytes as *segsize*, which must be a multiple of `GASNET_PAGESIZE`, and should be less than or equal to the value returned by `gasnet_getMaxLocalSegmentSize()`. *minheapoffset* specifies the minimum amount of virtual memory space (in bytes) to leave between the end of the current memory heap and the beginning of the remote-access memory segment (on some systems the size of this offset may limit the total future growth of the local memory heap, on other systems it may be irrelevant). All nodes are required to pass the same value for *minheapoffset*. Note that specifying a large *minheapoffset* may limit the possible size of the remote-access segment on some systems. Passing a *segsize* of zero disables the remote-access segment for this node, meaning other nodes cannot access it with remote-memory operations and this node cannot be the target of any Long AM messages.

GASNet will attempt to place the data segment in an area of the virtual memory space whose pages are currently unused (e.g. by calling `mmap`). The actual remote-access segment size achieved may be less than *segsize* if insufficient system resources are available - the exact size and location of the segment for all nodes should be queried after attach using `gasnet_getSegmentInfo()`. The segment assignment is guaranteed to have a `GASNET_PAGESIZE`-aligned base address and size, but may differ in size across nodes, according to the requested segment sizes and system resource availability. GASNet will not initialize data within the memory segment in any way, nor will it attempt to access the memory locations within the segment until directed to do so by a data transfer function or Long active message.

If the GASNet implementation defines the macro `GASNET_ALIGNED_SEGMENTS` to 1, then `gasnet_attach()` guarantees that the base of the remote-access memory segment will be aligned at the same virtual address across all nodes (and will fail if it cannot provide this). Otherwise, this guarantee is not provided. Note the segment sizes may still differ across nodes, based on *segsize* and system resource availability.

In the `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` configurations, GASNet guarantees that data transfer functions, Long active messages and local accesses referencing memory locations in the remote-access memory segment will succeed, even before any local activity takes place on those pages (i.e. in an implementation performing lazy registration, first touch = allocate).

segsize and *minheapoffset* are ignored in the `GASNET_SEGMENT_EVERYTHING` configuration, as the entire virtual memory space is implicitly shared for remote access. Under this configuration, it is the client's responsibility to ensure that any remote-memory references fall within the legal areas of the current heap and data segment for the target node - remote accesses or Long active messages to locations outside these areas will have undefined effects (for example, they *may* cause a segmentation fault on the target node).

`gasnet_attach()` may fail with a fatal error and implementation-defined message if the network cannot be successfully initialized. It also may return an error code such as `GASNET_ERR_RESOURCE` to indicate there was a problem acquiring network or system resources. Otherwise, it returns `GASNET_OK` to indicate success.

A successful call acts as a global barrier and blocks until all other nodes which are part of this parallel job have successfully called `gasnet_attach()`. May only be called once during a process lifetime, subsequent calls will return an error.

Implementor's Note:

- In the `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` configurations, GASNet must take steps to ensure the pages in the segment have been properly registered for remote access in a system-specific and implementation-specific way (e.g. mmapping them so they get added to the process page table, pinning the pages, registering the physical address with the NIC, etc.). Implementations are encouraged to defer consuming physical memory or swap space resources for pages in the segment until the first actual reference to them.
- Every implementation that pins pages needs a strategy for handling remote accesses under the `GASNET_SEGMENT_LARGE` and `GASNET_SEGMENT_EVERYTHING` configurations when the segment size exceeds the amount of pinnable pages - e.g. some implementations may dynamically pin pages, others may pin only a portion of the segment and use an extra copy to handle access to data outside the pinned region.
- Some GASNet implementations may need to allocate and pin additional memory for their own internal use in messaging (e.g. send buffers), but such memory should not fall within the client's data segment under `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` (although it may be adjacent to it).
- Some GASNet implementations may also choose to pin other pages to optimize access and remove extra copies - for example, pinning the program stack may be advisable on some systems since a large number of the data transfer functions in the extended API are likely to use stack locations as the local source/destination.

2.1.3 gasnet_getMaxLocalSegmentSize

`uintptr_t gasnet_getMaxLocalSegmentSize ()`

Retrieve an approximate, optimistic maximum size in bytes for the remote-access memory segment that may be provided to `gasnet_attach()` under the current configuration.

The return value of this function may depend on current system resource usage, and may return different values on different nodes of a job, according to current system utilization. The value returned will always be a multiple of `GASNET_PAGESIZE`.

The value returned is an optimistic approximation of the segment size which can be acquired by `gasnet_attach()` - the actual size achieved can be queried after attach using `gasnet_getSegmentInfo()`.

On many implementations, this function will return different values in the `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` configurations. Under the `GASNET_SEGMENT_EVERYTHING` configuration, this function returns -1.

This function has undefined behavior after `gasnet_attach()`.

2.1.4 gasnet_getMaxGlobalSegmentSize

`uintptr_t gasnet_getMaxGlobalSegmentSize ()`

Returns a global minimum value that would be returned by a call to `gasnet_getMaxLocalSegmentSize` on any node of the current job (i.e. the smallest max segment size estimated for any node in the job).

This function has undefined behavior after `gasnet_attach()`.

2.1.5 gasnet_exit

`void gasnet_exit (`*int* `exitcode)`

Terminate the current GASNet job and return the given *exitcode* to the console which invoked the job (in a system-specific way). This call is *not* a collective operation, meaning any node may call it at any time after initialization. It causes the system to flush all I/O, release all resources and terminate the job for all active nodes. If several nodes and/or threads call it simultaneously with different exit codes within a given synchronization phase, the result provided to the console will be one of the provided exit codes (chosen arbitrarily). This function should be called at the end of `main()` after a barrier to ensure proper system exit, and should also be called in the event of any fatal errors. GASNet clients are encouraged to call `gasnet_exit()` before explicitly exiting (by calling `exit()`, `abort()`) to reduce the possibility and lifetime of orphaned nodes, but this is not required.

GASNet will send a `SIGQUIT` signal to the node if it detects that a remote node has called `gasnet_exit` or crashed (in which case the node should catch the signal, perform any system-specific shutdown, then call `gasnet_exit()` to end the local node process). GASNet will also send a `SIGQUIT` signal if it detects that the job has received a different catchable terminate-the-program signal (e.g. `SIGTERM`, `SIGINT`) since some of these other signals may be meaningful (and non-fatal) to certain GASNet implementations.

2.2 Job Environment Queries

2.2.1 gasnet_mynode

`gasnet_node_t gasnet_mynode ()`

returns the unique, 0-based node index representing this node in the current GASNet job

2.2.2 gasnet_nodes

`gasnet_node_t gasnet_nodes ()`

returns the number of nodes in the current GASNet job

2.2.3 gasnet_getSegmentInfo

```
typedef struct {
  void *addr;
  uintptr_t size;
} gasnet_seginfo_t;
```

`int gasnet_getSegmentInfo` (*gasnet_seginfo_t* ***seginfo_table***, *int* ***numentries***)

Query the segment base addresses and sizes for all the nodes in the job. *seginfo_table* is an array of `gasnet_seginfo_t` (and *numentries* is the number of entries in the table). GASNet fills in the table with the remote-access segment base address and size in bytes for each node whose index is less than *numentries*. The value of *numentries* is usually equal to `gasnet_nodes()`, but is permitted to be greater (in which case higher array entries are left untouched) or less (in which case the higher-numbered nodes are not reported). This is a non-collective operation. Returns `GASNET_OK` on success.

Note that when GASNET_ALIGNED_SEGMENTS=1, the base addresses are guaranteed to be equal (i.e. all remote-access segments start at the same virtual addresses). However, in any case the segment sizes may differ across nodes, and specifically they may differ from the size requested by the client in the `gasnet_attach()` size hint.

2.2.4 gasnet_getenv

`char * gasnet_getenv` (*const char* ***name***)

Has the same semantics as the POSIX `getenv()` call, except it queries the system-specific environment which was used to spawn the job (e.g. the environment of the spawning console). Calling POSIX `getenv()` directly on some implementations may not correctly return values reflecting the environment that initiated the job spawn, consequently GASNet clients wishing to query a consistent snapshot of the spawning environment across nodes should never call `getenv()` directly. The semantics of POSIX `setenv()` are undefined in GASNet jobs (specifically, it will probably fail to propagate changes across nodes).

2.3 Active Messaging Interface

Active message communication is formulated as logically matching request and reply operations. Upon receipt of a request message, a request handler is invoked; likewise, when a reply message is received, the reply handler is invoked. Request handlers can reply at most once to the requesting node. If no explicit reply is made, the layer may generate one (to an implicit do-nothing reply handler). Thus a request handler can call reply at most once, and may only reply to the requesting node. Reply handlers cannot request or reply.

Here is a high-level description of a typical active message exchange between two nodes, A and B:

1. A calls `gasnet_AMRequest*()` to send a request to B. The call includes arguments, data payload, the node index of B and the index of the request handler to run on B when the request arrives
2. At some later time, B receives the request, and runs the appropriate request handler with the arguments and data (if any) provided in the `gasnet_AMRequest*()` call. The request handler does some work on the arguments, and usually finishes by calling `gasnet_AMReply*()` to issue a reply message before it exits (replying is optional in GASNet, but required in AM2 - if the request handler does not reply then no further actions are taken). `gasnet_AMReply*()` takes the token passed to the request handler, arguments and data payload, and the index of the reply handler to run when the reply message arrives. It does not take a node index because a request handler is only permitted to send a reply to the requesting node
3. At some later time, A receives the reply message from B and runs the appropriate reply handler, with the arguments and data (if any) provided in the `gasnet_AMReply*()` call. The reply handler does some work on the arguments and then exits. It is not permitted to send further messages.

The message layer will deliver requests and replies to destination nodes barring any catastrophic errors (e.g. node crashes). From a sender's point of view, the request and reply functions block until the message is sent. A message is defined to be sent once it is safe for the caller to reuse the storage (registers or memory) containing the message (one notable exception to this policy is `gasnet_RequestLongAsyncM()`). In implementations which copy or buffer messages for transmission, the definition still holds: message sent means the layer has copied the message and promises to deliver the copy with its "best effort", and the original message storage may be reused.

By best effort, the message layer promises it will take care of all the details necessary to transmit the message. These details include any retransmission attempts and buffering issues on unreliable networks.

However, in either case, sent does not imply received. Once control returns from a request or reply function, clients cannot assume that the message has been received and handled at the destination. The message layer only guarantees that if a request or reply is sent, and, if the receiver occasionally polls for arriving messages, then the message will eventually be received and handled. From a receiver's point of view, a message is defined to be received only once its handler function is invoked. The contents of partially received messages and messages whose handlers have not executed are undefined.

If the client sends an AM request or AM reply to a handler index which has not been registered on the destination node, GASNet will print an implementation-defined error message and terminate the job. It is implementation-defined whether this checking happens on the sending or receiving node.

2.3.1 Active Message Categories

There are three categories of active messages:

'`Short Active Message`'

These messages carry only a few integer arguments (up to `gasnet_AMMaxArgs()`)
handler prototype:

```
void handler(gasnet_token_t token,
             gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

'`Medium Active Message`'

In addition to integer arguments, these messages can can carry an opaque data payload (up to `gasnet_AMMaxMedium()` bytes in length), that will be made available to the handler when it is run on the remote node.
handler prototype:

```
void handler(gasnet_token_t token,
             void *buf, size_t nbytes,
             gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

'`Long Active Message`'

In addition to integer arguments, these messages can carry an opaque data payload (up to `gasnet_AMMaxLong{Request,Reply}()` bytes in length) which is destined for a particular predetermined address in the segment of the remote node (often implemented using RDMA hardware assistance)
handler prototype:

```
void handler(gasnet_token_t token,
             void *buf, size_t nbytes,
             gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

For more discussion on these three categories, see the Appendix.

The number of handler arguments (M) is specified upon issuing a request or reply by choosing the request/reply function of the appropriate name. The category of message and value of M used in the request/reply message sends determines the appropriate handler prototype, as detailed above. If a request or reply is sent to a handler whose prototype does not match the requirements as detailed above, the result is undefined.

Implementor's Note:

- Some implementations may choose to optimize medium and long messages for payloads whose base address and length are aligned with certain convenient sizes (word-aligned, doubleword-aligned, page-aligned etc.) but this does not affect correctness.

2.3.2 Active Message Size Limits

These functions are used to query the maximum size messages of each category supported by a given implementation. These are likely to be implemented as macros for efficiency of client code which uses them (within packing loops, etc.)

2.3.2.1 gasnet_AMMaxArgs

`size_t gasnet_AMMaxArgs ()`

Returns the maximum number of handler arguments (i.e. M) that may be passed with any AM request or reply function. This value is guaranteed to be at least `(2 * MAX(sizeof(int),sizeof(void*)))` (i.e. 8 for 32-bit systems, 16 for 64-bit systems), which ensures that 8 ints and/or pointers can be sent with any active message. All implementations must support *all* values of M from 0...`gasnet_AMMaxArgs()`.

2.3.2.2 gasnet_AMMaxMedium

`size_t gasnet_AMMaxMedium ()`

Returns the maximum number of bytes that can be sent in the payload of a single medium AM request or reply. This value is guaranteed to be at least 512 bytes on any implementation.

2.3.2.3 gasnet_AMMaxLongRequest

`size_t gasnet_AMMaxLongRequest ()`

Returns the maximum number of bytes that can be sent in the payload of a single long AM request. This value is guaranteed to be at least 512 bytes on any implementation. Implementations which use RDMA to implement long messages are likely to support a much larger value.

2.3.2.4 gasnet_AMMaxLongReply

`size_t gasnet_AMMaxLongReply ()`

Returns the maximum number of bytes that can be sent in the payload of a single long AM reply. This value is guaranteed to be at least 512 bytes on any implementation. Implementations which use RDMA to implement long messages are likely to support a much larger value.

2.3.3 Active Message Request Functions

In the function descriptions below, M is to be replaced with a number in [0 ... `gasnet_AMMaxArgs()`]

2.3.3.1 gasnet_AMRequestShortM

`int gasnet_AMRequestShortM (` *gasnet_node_t* `dest`, *gasnet_handler_t* `handler`,
gasnet_handlerarg_t `arg0`, ..., *gasnet_handlerarg_t* `argM-1` `);`

Send a short AM request to node *dest*, to run the handler registered on the destination node at handler table index *handler*, with the given M arguments. `gasnet_AMRequestShortM` returns control to the calling thread of computation after sending the request message. Upon receipt, the receiver invokes the appropriate active message request handler function with the M integer arguments. Returns `GASNET_OK` on success.

2.3.3.2 gasnet_AMRequestMediumM

`int gasnet_AMRequestMediumM (` *gasnet_node_t* `dest`, *gasnet_handler_t* `handler`,
void *`source_addr`, *size_t* `nbytes`,
gasnet_handlerarg_t `arg0`, ..., *gasnet_handlerarg_t* `argM-1` `)`

Send a medium AM request to node *dest*, to run the handler registered on the destination node at handler table index *handler*, with the given M arguments.

The message also carries a data payload copied from the local node's memory space as indicated by *source_addr* and *nbytes* (which need not fall within the registered data segment on the local node). The value of *nbytes* must be no larger than the value returned by `gasnet_AMMaxMedium()`, and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined).

gasnet_AMRequestMediumM returns control to the calling thread of computation after sending the associated request, and the source memory may be freely modified once the function returns. The active message is logically delivered after the data transfer finishes.

Upon receipt, the receiver invokes the appropriate request handler function with a pointer to temporary storage containing the data payload (in a buffer which is suitably aligned to hold any datatype), the number of data bytes transferred, and the M integer arguments. The dynamic scope of the storage is the same as the dynamic scope of the handler. The data should be copied if it is needed beyond this scope. Returns GASNET_OK on success.

2.3.3.3 gasnet_AMRequestLongM

int gasnet_AMRequestLongM (*gasnet_node_t* **dest**, *gasnet_handler_t* **handler**,
void ***source_addr**, *size_t* **nbytes**, *void* ***dest_addr**,
gasnet_handlerarg_t **arg0**, ..., *gasnet_handlerarg_t* **argM-1**);

Send a long AM request to node *dest*, to run the handler registered on the destination node at handler table index *handler*, with the given M arguments.

The message also carries a data payload copied from the local node's memory space as indicated by *source_addr* and *nbytes* (which need not fall within the registered data segment on the local node). The value of nbytes must be no larger than the value returned by gasnet_AMMaxLongRequest(), and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined). The memory specified by [*dest_addr*...(*dest_addr*+*nbytes*-1)] must fall entirely within the memory segment registered for remote access by the destination node. This area will receive the data transfer before the handler runs.

If the source and destination memory overlap (e.g. in a loopback message), the result is undefined. gasnet_AMRequestLongM returns control to the calling thread of computation after sending the associated request, and the source memory may be freely modified once the function returns. The active message is logically delivered after the bulk transfer finishes. Upon receipt, the receiver invokes the appropriate request handler function with a pointer into the memory segment where the data was placed, the number of data bytes transferred, and the M integer arguments. Returns GASNET_OK on success.

2.3.3.4 gasnet_AMRequestLongAsyncM

int gasnet_AMRequestLongAsyncM (*gasnet_node_t* **dest**, *gasnet_handler_t* **handler**,
void ***source_addr**, *size_t* **nbytes**, *void* ***dest_addr**,
gasnet_handlerarg_t **arg0**, ..., *gasnet_handlerarg_t* **argM-1**);

gasnet_AMRequestLongAsyncM() has identical semantics to gasnet_AMRequestLongM(), except that the handler is required to send an AM reply and the data payload source memory must NOT be modified until this matching reply handler has begun execution. Some implementations may leverage this additional constraint to provide higher performance (e.g. by reducing extra data copying).

Implementor's Note:

- Note that unlike the AM2.0 function of similar name, this function is permitted to block temporarily if the network is unable to immediately accept the new request.

2.3.4 Active Message Reply Functions

The following active message reply functions may only be called from the context of a running active message request handler, and a reply function may be called at most once from any given request handler (it is an error to do otherwise). The request and reply categories need not match (e.g. a short AM request handler may send a long AM reply).

2.3.4.1 gasnet_AMReplyShortM

`int gasnet_AMReplyShortM (` *gasnet_token_t* **`token`**, *gasnet_handler_t* **`handler`**,
gasnet_handlerarg_t **`arg0`**, ..., *gasnet_handlerarg_t* **`argM-1`** *);*

Send a short AM reply to the indicated *handler* on the requesting node (i.e. the node responsible for this particular invocation of the request handler), and include the given M arguments. `gasnet_AMReplyShortM` returns control to the calling thread of computation after sending the reply message.

Upon receipt, the receiver invokes the appropriate active message reply handler function with the M integer arguments. Returns `GASNET_OK` on success.

2.3.4.2 gasnet_AMReplyMediumM

`int gasnet_AMReplyMediumM (` *gasnet_token_t* **`token`**, *gasnet_handler_t* **`handler`**,
void **`*source_addr`**, *size_t* **`nbytes`**,
gasnet_handlerarg_t **`arg0`**, ..., *gasnet_handlerarg_t* **`argM-1`** *);*

Send a medium AM reply to the indicated *handler* on the requesting node (i.e. the node responsible for this particular invocation of the request handler), with the given M arguments and given data payload copied from the local node's memory space (*source_addr* need not fall within the registered data segment on the local node). The value of *nbytes* must be no larger than the value returned by `gasnet_AMMaxMedium()`, and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined). `gasnet_AMReplyMediumM` returns control to the calling thread of computation after sending the associated reply, and the source memory may be freely modified once the function returns. The active message is logically delivered after the data transfer finishes.

Upon receipt, the receiver invokes the appropriate reply handler function with a pointer to temporary storage containing the data payload, the number of data bytes transferred, and the M integer arguments. The dynamic scope of the storage is the same as the dynamic scope of the handler. The data should be copied if it is needed beyond this scope. Returns `GASNET_OK` on success.

2.3.4.3 gasnet_AMReplyLongM

`int gasnet_AMReplyLongM (` *gasnet_token_t* **`token`**, *gasnet_handler_t* **`handler`**,
void **`*source_addr`**, *size_t* **`nbytes`**, *void* **`*dest_addr`**,
gasnet_handlerarg_t **`arg0`**, ..., *gasnet_handlerarg_t* **`argM-1`** *);*

Send a long AM reply to the indicated *handler* on the requesting node (i.e. the node responsible for this particular invocation of the request handler), with the given M arguments and given data payload copied from the local node's memory space (*source_addr* need not fall within the registered data segment on the local node). The value of *nbytes* must be no larger than the value returned by `gasnet_AMMaxLongReply()`, and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined). The memory specified by [*dest_addr*...(*dest_addr*+*nbytes*-1)] must fall entirely within the memory segment registered for remote access by the destination node. If the source and destination memory overlap (e.g. in a loopback message), the result is undefined. `gasnet_AMReplyLongM` returns control to the calling thread of computation after sending the associated reply, and the source memory may be freely modified once the function returns. The active message is logically delivered after the bulk transfer finishes.

Upon receipt, the receiver invokes the appropriate reply handler function with a pointer into the memory segment where the data was placed, the number of data bytes transferred, and the M integer arguments. Returns `GASNET_OK` on success.

2.3.5 Misc. Active Message Functions

2.3.5.1 gasnet_AMPoll

`int gasnet_AMPoll ()`

An explicit call to service the network, process pending messages and run handlers as appropriate. Most of the message-sending primitives in GASNet poll the network implicitly. Purely polling-based implementations of GASNet may require occasional calls to this function to ensure progress of remote nodes during compute-only loops. Any client code which spin-waits for the arrival of a message should call this function within the spin loop to optimize response time. This call may be a no-op on some implementations (e.g. purely interrupt-based implementations). Returns `GASNET_OK` unless an error condition was detected.

2.3.5.2 GASNET_BLOCKUNTIL

`#define GASNET_BLOCKUNTIL(cond) ???`

This is a macro which implements a busy-wait/blocking polling loop in the way most efficient for the current GASNet core implementation. The macro blocks execution of the current thread and services the network until the provided condition becomes true. *cond* is an arbitrary C expression which will be evaluated by the macro one or more times as active messages arrive until the condition evaluates to a non-zero value. *cond* is an expression whose value is altered by the execution of an AM handler which the client thread is waiting for - GASNet may safely assume that the value of *cond* will only change while an AM handler is executing.

Example usage:

```
int doneflag = 0;
gasnet_AMRequestShort1(..., &doneflag); // reply handler sets doneflag to 1
GASNET_BLOCKUNTIL(doneflag == 1);
```

Note that code like this would be illegal and could cause node 0 to sleep forever:

```
static int doneflag = 0;
node 0:                                 node 1:
GASNET_BLOCKUNTIL(doneflag == 1);       gasnet_put_val(0, &doneflag, 1, sizeof(int));
```

because `gasnet_put_val` (and other extended API functions) might not be implemented using AM handlers. Also note that *cond* may be evaluated concurrently with handler execution, so the client is responsible for negotiating any atomicity concerns between the cond expression and handlers (for example, protecting both with a handler-safe lock if the *cond* expression reads two or more values which are all updated by handlers). Finally, note that unsynchronized handler code which modifies one or more locations and then performs a flag write to signal a different thread may need to execute a local memory barrier before the flag write to ensure correct ordering on non-sequentially-consistent SMP hardware.

Implementor's Note:

- one trivial implementation: `#define GASNET_BLOCKUNTIL(cond) while (!(cond)) gasnet_AMPoll()`
- smarter implementations may choose to spin for awhile and then block
- Any implementation that includes blocking must ensure progress if all client threads call GASNET_BLOCKUNTIL(), and must ensure the blocked thread is awakened even if the handler is run synchronously during a `gasnet_AMPoll()` call from a different client thread. Other client threads performing sends or polls must not be prevented from making progress by the blocking thread (possibly a motivation *against* the "trivial implementation" above).

2.3.5.3 gasnet_AMGetMsgSource

`int gasnet_AMGetMsgSource (`*gasnet_token_t* `token`, *gasnet_node_t* `*srcindex`)

Can be called by handlers to query the source of the message being handled. The *token* argument must be the token passed into the handler on entry. Returns `GASNET_OK` on success.

2.4 Atomicity Control

2.4.1 Atomicity semantics of handlers

Handlers may run asynchronously with respect to the main computation (in an implementation which uses interrupts to run some or all handlers), and they may run concurrently with each other on separate threads (e.g. in an implementation where several threads may be polling the network at once). An implementation using interrupts may result in handler code running within a signal handler context. Some implementations may even choose to run handlers on a separate private thread created by GASNet (making handlers asynchronous with respect to all client threads). Note that polling-based GASNet implementations are likely to poll (and possibly run handlers) from within *any* GASNet call (i.e. not just `gasnet_AMPoll()`). Because of all this, handler code should run quickly and to completion without making blocking calls, and should not make assumptions about the context in which it is being run (special care must be taken to ensure safety in a signal handler context, see below).

Regardless, handlers themselves are not interruptible - any given thread will only be running a single AM handler at a time and will never be interrupted to run another AM handler (there is one exception to this rule - the `gasnet_AMReply*()` call in a request handler may cause reply handlers to run synchronously, which may be necessary to avoid deadlock in some implementations. This should not be a problem since `gasnet_AMReply*()` is often the last action taken by a request handler). Handlers are specifically prohibited from initiating random network communication to prevent deadlock - request handlers must generate at most one reply (to the requestor) and make no other communication calls (including polling), and reply handlers may not communicate or poll at all.

The asynchronous nature of handlers requires two mechanisms to make them safe: a mechanism to ensure signal safety for GASNet implementations using interrupt-based mechanisms, and a locking mechanism to allow atomic updates from handlers to data structures shared with the client threads and other handlers.

2.4.2 No-Interrupt Sections - Ensuring signal-safety for handlers

Traditionally, code running in signal handler context is extremely circumscribed in what it can do: e.g. none of the standard pthreads/System V synchronization calls are on the list of signal-safe functions (for such a list see *POSIX System Interfaces 2.4, IEEE Std 1003.1-2001*). Note that even most "thread-safe" libraries will break or deadlock if called from a signal handler by the same thread currently executing a different call to that library in an earlier stack frame. One specific case where this is likely to arise in practice is calls to `malloc()/free()`. To overcome these limitations, and allow our handlers to be more useful, the normal limitations on signal handlers will be avoided by allowing the client thread to temporarily disable the network interrupts that run handlers. All function calls that are not signal-safe and could possibly access state shared by functions also called from handlers MUST be called within a GASNet "No-Interrupt Section":

2.4.2.1 gasnet_hold_interrupts, gasnet_resume_interrupts

```
void gasnet_hold_interrupts ()
void gasnet_resume_interrupts ()
```

`gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` are used to define a GASNet No-Interrupt Section (any code which dynamically executes between the hold and resume calls is said to be "inside" the No-Interrupt Section). These are likely to be implemented as macros and highly tuned for efficiency. The hold and resume calls must be paired, and may *not* be nested recursively or the results are undefined (this means that clients should be especially careful when calling other functions in the client from within a No-Interrupt Section). Both calls will return immediately in the common case, although one or both may cause messages to be serviced on some implementations. GASNet guarantees that no handlers will run asynchronously **on the current thread** within the No-Interrupt Section. The no-interrupt state is a per-thread setting, and GASNet may continue running handlers synchronously or asynchronously on other client threads or GASNet-private threads (even in a `GASNET_SEQ` configuration) - specifically, a No-Interrupt Section does **not** guarantee atomicity with respect to handler code, it merely provides a way to ensure that handlers won't run on a given thread while it's inside a call to a non-signal-safe library.

2.4.3 Restrictions on No-Interrupt Sections

There is a strict set of conventions governing the use of No-Interrupt Sections which must be followed in order to ensure correct operation on all GASNet implementations. Clients which violate any of these rules may be subject to intermittent crashes, fatal errors or network deadlocks.

- `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` need not be called from within a handler context - handlers are run within an implicit No-Interrupt Section, and `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` calls are ignored within a handler context.
- Code in a No-Interrupt Section must not call any GASNet functions that may send requests or synchronously run handlers - specifically, the only GASNet functions which may legally be called within the No-Interrupt Section are:

  `gasnet_mynode(), gasnet_nodes(), gasnet_hsl_*(), gasnet_exit(), gasnet_AMReply*()`

  Note that due to the previous rule, these are also the only GASNet functions that may legally be called within a handler context (and `gasnet_AMReply*()` is only legal in a request handler).
- Code in a No-Interrupt Section must never block or spin-wait for an unbounded amount of time, especially when awaiting a result produced by a handler. The *only* exception to this rule is that a thread may call `gasnet_hsl_lock` within a No-Interrupt Section (subject to the rules in section see Section 2.4.5 [Restrictions on Handler-Safe Locks], page 17).
- No-Interrupt Sections should only be held "briefly" to avoid starving the network (could cause performance degradation, but should not affect correctness). Very long No-Interrupt Sections (i.e. on the order of 10 sec or more) could cause some GASNet implementations employing timeout-based mechanisms to fail (e.g. remote nodes may decide this node is dead and abort the job).

Implementor's Note:

- One possible implementation: Keep a bit for each thread indicating whether or not a No-Interrupt Section is in effect, which is checked by all asynchronous signal handlers. If a signal arrives while a No-Interrupt Section is in effect, a different per-thread bit in memory will be marked indicating a "missed GASNet signal": the `gasnet_resume_interrupts()` call will check this bit, and if it is set, the action for the signal will be taken (the action for a GASNet signal is always to check the queue of incoming network messages, so there's no ambiguity on what the signal meant. Since messages are queued, the single 'signal missed' bit is sufficient for an arbitrary number of missed signals during a single No-Interrupt Section - GASNet messages will be removed and processed until the queue is empty).
- Implementation needs to hold a No-Interrupt Section over a thread while running handlers or holding HSL's
- Strictly polling-based implementations which never interrupt a thread can implement these as a no-op.

2.4.4 Handler-Safe Locks

In order to support handlers atomically updating data structures accessed by the main-line client code and other handlers, GASNet provides the Handler-Safe Lock (HSL) mechanism. As the name implies, these are a special kind of lock which are distinguished as being the **only** type of lock which may be safely acquired from a handler context. There is also a set of restrictions on their usage which allows this to be safe (see below). All lock-protected data structures in the client that need to be accessed by handlers should be protected using a Handler-Safe Lock (i.e. instead of a standard POSIX mutex).

2.4.4.1 gasnet_hsl_t

`gasnet_hsl_t` is an opaque type representing a Handler-Safe Lock. HSL's operate analogously to POSIX mutexes, in that they are always manipulated using a pointer.

2.4.4.2 gasnet_hsl_init, gasnet_hsl_destroy

`gasnet_hsl_t hsl = GASNET_HSL_INITIALIZER;`

`void gasnet_hsl_init` (*gasnet_hsl_t* ***hsl***)
`void gasnet_hsl_destroy` (*gasnet_hsl_t* ***hsl***)

Similarly to POSIX mutexes, HSL's can be created in two ways. They can be statically declared and initialized using the `GASNET_HSL_INITIALIZER` constant. Alternately, HSL's allocated using other means (such as dynamic allocation) may be initialized by calling `gasnet_hsl_init()`. `gasnet_hsl_destroy()` may be called on either type of HSL once it's no longer needed to release any system resources associated with it. It is erroneous to call `gasnet_hsl_init()` on a given HSL more than once. It is erroneous to destroy an HSL which is currently locked. Any errors detected in HSL initialization/destruction are fatal.

2.4.4.3 gasnet_hsl_lock, gasnet_hsl_unlock

`void gasnet_hsl_lock` (*gasnet_hsl_t* ***hsl***)
`int gasnet_hsl_trylock` (*gasnet_hsl_t* ***hsl***)
`void gasnet_hsl_unlock` (*gasnet_hsl_t* ***hsl***)

Lock and unlock HSL's.

`gasnet_hsl_lock(hsl)` will block until the *hsl* lock can be acquired by the current thread. `gasnet_hsl_lock()` may be called from within main-line client code or from within handlers - this is the **only** blocking call which is permitted to execute within a GASNet handler context (e.g. it is erroneous to call POSIX mutex locking functions).

`gasnet_hsl_trylock(hsl)` attempts to acquire `hsl` for the current thread, returning immediately (without blocking). If the lock was successfully acquired, this function returns `GASNET_OK`. If the lock could not be acquired (e.g it was found to be held by another thread) then this function returns GASNET_ERR_NOT_READY and the lock is not acquired. It is *not* legal for an AM handler to spin-poll a lock without bound using `gasnet_hsl_trylock()` waiting for success - AM handlers must always use `gasnet_hsl_lock()` when they wish to block to acquire an HSL.

`gasnet_hsl_unlock(hsl)` releases the *hsl* lock previously acquired using `gasnet_hsl_lock(hsl)` or a successful `gasnet_hsl_trylock(hsl)`, and not yet released. It is erroneous to call any of these functions on HSL's which have not been properly initialized.

Note that under the `GASNET_SEQ` configuration, HSL locking functions may only be called from handlers and the designated GASNet client thread (*not* from other client threads that may happen to exist - those threads are not permitted to make *any* GASNet calls, which includes HSL locking calls).

All HSL locking/unlocking calls must follow the usage rules documented in the next section.

2.4.5 Restrictions on Handler-Safe Locks

There is a strict set of conventions governing the use of HSL's which must be followed in order to ensure correct operation on all GASNet implementations. Amongst other things, the restrictions are designed to ensure that HSL's are always held for a strictly bounded amount of time, to ensure that acquiring them from within a handler can't lead to deadlock. Clients which violate any of these rules may be subject to intermittent crashes, fatal errors or network deadlocks.

- Code executing on a thread holding an HSL is implicitly within a No-Interrupt Section, and must follow all the restrictions on code within a No-Interrupt Section (see Section 2.4.3 [Restrictions on No-Interrupt Sections], page 16). Calls to `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` are ignored while holding an HSL.
- Any handler which locks one or more HSL's **must** unlock them all before returning or calling `gasnet_AMReply*()`
- HSL's may **not** be locked recursively (i.e. calling `gasnet_hsl_lock()` or `gasnet_hsl_trylock(hsl)` on a lock already held by the current thread) and attempting to do so will lead to undefined behavior. It **is** permitted for a thread to acquire more than one HSL, although the traditional cautions about the possibility of deadlock in the presence of multiple locks apply (e.g. the common solution is to define a partial order on locks and always acquire them in a monotonically ascending sequence).

- HSL's must be unlocked in the reverse order they were locked (e.g. lock A; lock B; ... unlock B; unlock A; is legal - reversing the order of unlocks is erroneous)
- HSL's may not be shared across GASNet processes executing on a machine - for example, it is specifically disallowed to place an HSL in a system V or mmapped shared memory segment and attempt to access it from two different GASNet processes.

Implementor's Note:

- HSL's are likely to just be a thin wrapper around a POSIX mutex - need to add just enough state/code to ensure the safety properties (must be a real lock, even under `GASNET_PARSYNC` because client may still have multiple threads). The only specific action required is that a No-Interrupt Section is enforced while the main-line code is holding an HSL (must be careful this works properly when multiple HSL's are held or when running in a handler).
- Robust implementations may add extra error checking to help discover violations of the restrictions, at least when compiled in a debugging mode - for example, it should be easy to detect: attempts at recursive locking on HSL's, incorrectly ordered unlocks, handlers that fail to release HSL's, explicit calls to `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` in a handler or while an HSL is held or in a No-Interrupt Section, and illegal calls to GASNet messaging functions while holding an HSL or inside a No-Interrupt Section.

3 Extended API

Errors in calls to the extended API are considered fatal and abort the job (by sending a `SIGABRT` signal) after printing an appropriate error message.

3.1 Memory-to-memory Data Transfer Functions

These comments apply to all put/get functions:

- The *nbytes* parameter should be a compile-time constant whenever possible (for efficiency)
- The source memory address for all gets and the target memory address for all puts must fall within the memory area registered for remote access by the remote node (see `gasnet_attach()`), or the results are undefined
- Pointers to remote memory are passed as an ordered pair of arguments: an integer node rank (a `gasnet_node_t`) and a `void *` virtual memory address, which logically represent a global pointer to the given address on the given node. These global pointers need not be remote - the node rank passed to these functions may in fact be the rank of the current node - implementations must support this form of loopback, and should probably attempt to optimize it by avoiding network traffic for such purely local operations.
- If the source memory and destination memory regions overlap the resulting value is undefined

3.2 Blocking memory-to-memory Transfers

3.2.1 gasnet_get, gasnet_put

`void gasnet_get` (*void* ***`dest`**, *gasnet_node_t* **`node`**, *void* ***`src`**, *size_t* **`nbytes`**)
`void gasnet_put` (*gasnet_node_t* **`node`**, *void* ***`dest`**, *void* ***`src`**, *size_t* **`nbytes`**)

Blocking get/put operations for aligned data. The get operation fetches *nbytes* bytes from the address *src* on node *node* and places them at *dest* in the local memory space. The put operation sends *nbytes* bytes from the address *src* in the local address space, and places them at the address *dest* in the memory space of node *node*. A call to these functions blocks until the transfer is complete, and the contents of the destination memory are undefined until it completes. If the contents of the source memory change while the operation is in progress the result will be implementation-specific. The *src* and *dest* addresses (whether local or remote) must be properly aligned for accessing objects of size *nbytes*. *nbytes* must be >= 0 and has no maximum size, but implementations will likely optimize for small powers of 2.

3.2.2 gasnet_get_bulk, gasnet_put_bulk

`void gasnet_get_bulk` (*void* ***`dest`**, *gasnet_node_t* **`node`**, *void* ***`src`**, *size_t* **`nbytes`**)
`void gasnet_put_bulk` (*gasnet_node_t* **`node`**, *void* ***`dest`**, *void* ***`src`**, *size_t* **`nbytes`**)

Blocking get/put operations for bulk (unaligned) data. These function similarly to the aligned get/put operations above, except the data is permitted to be unaligned, and implementations are likely to optimize for larger sizes of nbytes.

3.2.3 gasnet_memset

`void gasnet_memset` (*gasnet_node_t* **`node`**, *void* ***`dest`**, *int* **`val`**, *size_t* **`nbytes`**)

Blocking operation that has the same effect as if the *dest* node had executed the POSIX call `memset(`*`dest`*`,` *`val`*`,` *`nbytes`*`)`. As with puts, the destination memory must fall entirely within the memory area registered for remote access by the *dest* node (see `gasnet_attach`).

3.3 Non-blocking memory-to-memory transfers

The following functions provide non-blocking, split-phase memory access to shared data.

All such non-blocking operations require an initiation (generally a put or get) and a subsequent synchronization on the completion of that operation before the result is guaranteed.

There are two basic categories of non-blocking operations, defined by the synchronization mechanism used:

"*explicit handle*" *(nb) operations*
: These operations return a specific handle from the initiation that is used for synchronization. The handle can be used to synchronize a specific subset of the nb operations in-flight

"*implicit handle*" *(nbi) operations*
: These operations don't return a handle from the initiation - synchronization is accomplished by calling a synchronization routine that synchronizes all outstanding nbi operations.

3.3.1 Synchronization semantics of non-blocking data transfers

Successful synchronization of a non-blocking get operation means the local result is ready to be examined, and will contain a value held by the source location at some time in the interval between the call to the initiation function and the successful completion of the synchronization (note this specifically allows implementations to delay the underlying read until the synchronization operation is called, provided they preserve the blocking semantics of the synchronization function).

Successful synchronization of a put operation means the source data has been written to the destination location and get operations issued subsequently by any thread (or load instructions issued by the destination node) will receive the new value or a subsequently written value (assuming no other threads are writing the location)

Note that the order in which non-blocking operations complete is intentionally unspecified - the system is free to coalesce and/or reorder non-blocking operations with respect to other blocking or non-blocking operations, or operations initiated from a separate thread - the only ordering constraints that must be satisfied are those explicitly enforced using the synchronization functions (i.e. the non-blocking operation is only guaranteed to occur somewhere in the interval between initiation and successful synchronization on that operation).

Implementors should attempt to make the non-blocking initiation operations return as quickly as possible - however in some cases (e.g. when a large number of non-blocking operations have been issued or the network is otherwise busy) it may be necessary to block temporarily while waiting for the network to become available. In any case, all implementations must support at least $2^{16}-1$ non-blocking operations in-progress - that is, the client is free to issue up to $2^{16}-1$ non-blocking operations before issuing a sync operation, and the implementation must handle this correctly without deadlock or livelock.

3.3.2 Non-blocking memory-to-memory transfers (explicit handle)

The explicit-handle non-blocking data transfer functions return a `gasnet_handle_t` value to represent the non-blocking operation in flight. `gasnet_handle_t` is an opaque scalar type whose contents are implementation-defined, with one exception - every implementation must provide a scalar value corresponding to an "invalid" handle (`GASNET_INVALID_HANDLE`) and furthermore this value must be the result of setting all the bytes in the `gasnet_handle_t` datatype to zero. Implementators are free to define the `gasnet_handle_t` type to be any reasonable and appropriate size, although they are recommended to use a type which fits within a single standard register on the target architecture. In any case, the datatype should be wide enough to express at least $2^{16}-1$ different handle values, to prevent limiting the number of non-blocking operations in progress due to the number of handles available. `gasnet_handle_t` has value semantics, so for example it is permitted for clients to pass them across function call boundaries.

In the case of multithreaded clients (`GASNET_PAR` or `GASNET_PARSYNC`), `gasnet_handle_t` values are thread-specific. In other words, it is an error to obtain a handle value by initiating a non-blocking operation on one thread, and later pass that handle into a synchronization function from a different thread.

Any explicit-handle, non-blocking operation may return `GASNET_INVALID_HANDLE` to indicate it was possible to complete the operation immediately without blocking (e.g. operations where the "remote" node is actually the local node)

It is always an error to discard the `gasnet_handle_t` value for an explicit-handle operation in-flight - i.e. to initiate an operation and never synchronize on its completion.

3.3.2.1 gasnet_get_nb, gasnet_put_nb

`gasnet_handle_t gasnet_get_nb` (*void* ***dest**, *gasnet_node_t* **node**, *void* ***src**, *size_t* **nbytes**)
`gasnet_handle_t gasnet_put_nb` (*gasnet_node_t* **node**, *void* ***dest**, *void* ***src**, *size_t* **nbytes**)

Non-blocking get/put functions for aligned data. These functions operate similarly to their blocking counterparts, except they initiate a non-blocking operation and return immediately with a handle (`gasnet_handle_t`) which must later be used (by calling an explicit `gasnet_*_syncnb*()` function), to synchronize on completion of the non-blocking operation. The contents of the destination memory address are undefined until a synchronization completes successfully for the non-blocking operation. For the put version, the source memory may be safely overwritten once the initiation function returns.

3.3.2.2 gasnet_get_nb_bulk, gasnet_put_nb_bulk

`gasnet_handle_t gasnet_get_nb_bulk` (*void* ***dest**, *gasnet_node_t* **node**, *void* ***src**, *size_t* **nbytes**)
`gasnet_handle_t gasnet_put_nb_bulk` (*gasnet_node_t* **node**, *void* ***dest**, *void* ***src**, *size_t* **nbytes**)

Non-blocking get/put functions for bulk (unaligned) data. For the put version, the source memory may **not** be safely overwritten until a successful synchronization for the operation. If the contents of the source memory change while the operation is in progress the result will be implementation-specific. These otherwise behave identically to the non-bulk variants (but are likely to be optimized for large transfers).

3.3.2.3 gasnet_memset_nb

`gasnet_handle_t gasnet_memset_nb` (*gasnet_node_t* **node**, *void* ***dest**, *int* **val**, *size_t* **nbytes**)

Non-blocking operation that has the same effect as if the *dest* node had executed the POSIX call `memset(dest, val, nbytes)`. As with puts, the destination memory must fall entirely within the memory area registered for remote access by the *dest* node (see `gasnet_attach`).

The synchronization behavior is identical to a non-blocking, explicit-handle put operation (the `gasnet_handle_t` return value must be synchronized using an explicit-handle synchronization operation).

3.3.3 Synchronization for explicit-handle non-blocking operations

GASNet supports two basic types of synchronization for non-blocking operations - trying (polling) and waiting (blocking). All explicit-handle synchronization functions take one or more `gasnet_handle_t` values as input and either return an indication of whether the operation has completed or block until it completes.

3.3.3.1 gasnet_wait_syncnb, gasnet_try_syncnb

`void gasnet_wait_syncnb` (*gasnet_handle_t* **handle**)
`int gasnet_try_syncnb` (*gasnet_handle_t* **handle**)

Synchronize on the completion of a single specified explicit-handle non-blocking operation that was initiated by the calling thread. `gasnet_wait_syncnb()` blocks until the specified operation has completed (or returns immediately if it has already completed). In any case, the handle value is "dead" after `gasnet_wait_syncnb()` returns and may not be used in future synchronization operations. `gasnet_try_syncnb()` always returns immediately, with the value `GASNET_OK` if the operation is complete (at which point the handle value is "dead", and may not be used in future synchronization operations), or `GASNET_ERR_NOT_READY` if the operation is not yet complete and future synchronization is necessary to complete this operation.

It is legal to pass `GASNET_INVALID_HANDLE` as input to these functions - `gasnet_wait_sync(GASNET_INVALID_HANDLE)` returns immediately and `gasnet_try_sync(GASNET_INVALID_HANDLE)` returns `GASNET_OK`.

It is an error to pass a `gasnet_handle_t` value for an operation which has already been successfully synchronized using one of the explicit-handle synchronization functions.

3.3.3.2 gasnet_wait_syncnb_all, gasnet_try_syncnb_all

`void gasnet_wait_syncnb_all (gasnet_handle_t *handles, size_t numhandles)`
`int gasnet_try_syncnb_all (gasnet_handle_t *handles, size_t numhandles)`

Synchronize on the completion of an array of non-blocking explicit-handle operations (all of which were initiated by this thread). *numhandles* specifies the number of handles in the provided array of handles. `gasnet_wait_syncnb_all()` blocks until all the specified operations have completed (or returns immediately if they have all already completed). `gasnet_try_syncnb_all` always returns immediately, with the value `GASNET_OK` if all the specified operations have completed, or `GASNET_ERR_NOT_READY` if one or more of the operations is not yet complete and future synchronization is necessary to complete some of the operations.

Both functions will modify the provided array to reflect completions - handles whose operations have completed are overwritten with the value `GASNET_INVALID_HANDLE`, and the client may test against this value when `gasnet_try_syncnb_all()` returns `GASNET_ERR_NOT_READY` to determine which operations are complete and which are still pending.

It is legal to pass the value `GASNET_INVALID_HANDLE` in some of the array entries, and both functions will ignore it so that it has no effect on behavior. For example, if all entries in the array are `GASNET_INVALID_HANDLE` (or *numhandles*==0), then `gasnet_try_syncnb_all()` will return `GASNET_OK`.

3.3.3.3 gasnet_wait_syncnb_some, gasnet_try_syncnb_some

`void gasnet_wait_syncnb_some (gasnet_handle_t *handles, size_t numhandles)`
`int gasnet_try_syncnb_some (gasnet_handle_t *handles, size_t numhandles)`

These operate analogously to the `gasnet_*_syncnb_all` variants, except they only wait/test for at least one operation corresponding to a *valid* handle in the provided list to be complete (the valid handles values are all those which are not `GASNET_INVALID_HANDLE`). Specifically, `gasnet_wait_syncnb_some()` will block until at least one of the valid handles in the list has completed, and indicate the operations that have completed by setting the corresponding handles to the value `GASNET_INVALID_HANDLE`. Similarly, `gasnet_try_syncnb_some` will check if at least one valid handle in the list has completed (setting those completed handles to `GASNET_INVALID_HANDLE`) and return `GASNET_OK` if it detected at least one completion or `GASNET_ERR_NOT_READY` otherwise.

Both functions ignore `GASNET_INVALID_HANDLE` values so those values have no effect on behavior. If the input array is empty or consists only of `GASNET_INVALID_HANDLE` values, `gasnet_wait_syncnb_some` will return immediately and `gasnet_try_syncnb_some` will return `GASNET_OK`.

3.3.4 Non-blocking memory-to-memory transfers (implicit handle)

3.3.4.1 gasnet_get_nbi, gasnet_put_nbi, gasnet_get_nbi_bulk, gasnet_put_nbi_bulk, gasnet_memset_nbi

`void gasnet_get_nbi (void *dest, gasnet_node_t node, void *src, size_t nbytes)`
`void gasnet_put_nbi (gasnet_node_t node, void *dest, void *src, size_t nbytes)`
`void gasnet_get_nbi_bulk (void *dest, gasnet_node_t node, void *src, size_t nbytes)`
`void gasnet_put_nbi_bulk (gasnet_node_t node, void *dest, void *src, size_t nbytes)`
`void gasnet_memset_nbi (gasnet_node_t node, void *dest, int val, size_t nbytes)`

Non-blocking get/put functions for aligned and unaligned (bulk) data. These functions operate similarly to their explicit-handle counterparts, except they do not return a handle and must be synchronized using the implicit-handle synchronization operations. The contents of the destination memory address are undefined until a synchronization completes successfully for the non-blocking operation. As with the explicit-handle variants, the source memory for the non-bulk put operation may be safely overwritten once the initiation function returns, but the bulk put version requires the source memory to remain unchanged until the operation has been successfully completed using a synchronization.

`gasnet_memset_nbi` behaves identically to `gasnet_memset_nb`, except that it is synchronized as if it were a non-blocking, implicit-handle put operation.

3.3.5 Synchronization for implicit-handle non-blocking operations

The following functions are used to synchronize implicit-handle non-blocking operations.

In the case of multithreaded clients, implicit-handle synchronization functions only synchronize the implicit-handle non-blocking operations initiated from the calling thread. Operations initiated by other threads sharing the GASNet interface proceed independently and are not synchronized. Implicit-handle synchronization functions will synchronize operations initiated within other function frames by the calling thread (but this cannot affect the correctness of correctly synchronized code).

3.3.5.1 gasnet_wait_syncnbi_gets, gasnet_wait_syncnbi_puts, gasnet_wait_syncnbi_all, gasnet_try_syncnbi_gets, gasnet_try_syncnbi_puts, gasnet_try_syncnbi_all

```
void gasnet_wait_syncnbi_gets ()
void gasnet_wait_syncnbi_puts ()
void gasnet_wait_syncnbi_all ()
int gasnet_try_syncnbi_gets ()
int gasnet_try_syncnbi_puts ()
int gasnet_try_syncnbi_all ()
```

These functions implicitly specify a set of non-blocking operations on which to synchronize. They synchronize on a set of outstanding non-blocking implicit-handle operations initiated by this thread - either all such gets, all such puts, or all such puts and gets (where outstanding is defined as all those implicit-handle operations which have been initiated (outside an access region) but not yet completed through a successful implicit synchronization). The wait variants block until all operations in this implicit set have completed (indicating these operations have been successfully synchronized). The try variants test whether all operations in the implicit set have completed, and return `GASNET_OK` if so (which indicates these operations have been successfully synchronized) or `GASNET_ERR_NOT_READY` otherwise (in which case *none* of these operations may be considered successfully synchronized).

If there are no outstanding implicit-handle operations, these synchronization functions all return immediately (with `GASNET_OK` for the try variants).

Implementor's Note:

- Some implementations may choose to synchronize operations from other independent threads as well, but they must ensure progress for the calling thread in the presence of another thread which is continuously initiating implicit-handle non-blocking operations.

3.3.6 Implicit access region synchronization

In some cases, it may be useful or desirable to initiate a number of non-blocking shared-memory operations (possibly without knowing how many at compile-time) and synchronize them at a later time using a single, fast synchronization. Simple implicit handle synchronization may not be appropriate for this situation if there are intervening implicit accesses which are not to be synchronized. This situation could be handled using explicit-handle non-blocking operations and a list synchronization (e.g. `gasnet_wait_syncnb_all()`), but this may not be desirable because it requires managing an array of handles (which could have negative cache effects on performance, or could be expensive to allocate when the size is not known until runtime). To handle these cases, we provide "implicit access region" synchronization, described below.

3.3.6.1 gasnet_begin_nbi_accessregion, gasnet_end_nbi_accessregion

```
void gasnet_begin_nbi_accessregion ();
gasnet_handle_t gasnet_end_nbi_accessregion ();
```

`gasnet_begin_nbi_accessregion()` and `gasnet_end_nbi_accessregion()` are used to define an implicit access region (any code which dynamically executes between the begin and end calls is said to be "inside" the region) The begin and end calls must be paired, and may not be nested recursively or the results are undefined. It is erroneous to call any implicit-handle synchronization function within the access region. All implicit-handle non-blocking operations initiated inside the region become "associated" with the abstract access region handle being constructed. `gasnet_end_nbi_accessregion()` returns an explicit handle which jointly represents all the associated implicit-handle operations (those initiated within the access region). This handle can then be passed to the regular explicit-handle synchronization functions, and will be successfully synchronized when *all* of the associated non-blocking operations (both puts and gets) initiated in the access region have completed. The associated operations cease to be implicit-handle operations, and are *not* synchronized by subsequent calls to the implicit-handle synchronization functions occurring after the access region (e.g. `gasnet_wait_syncnbi_all()`). Explicit-handle operations initiated within the access region operate as usual and do *not* become associated with the access region.

Sample code:

```
gasnet_begin_nbi_accessregion(); // begin the access region

gasnet_put_nbi(...); // becomes assoc. with access region
while (...) {
  gasnet_put_nbi(...); // becomes assoc. with access region
}

// unrelated explicit-handle operation not assoc. with access region
h2 = gasnet_get_nb(...);
gasnet_wait_syncnb(h2);

// end the access region and get the handle
handle = gasnet_end_nbi_accessregion();

.... // other code, which may include unrelated implicit-handle
     // operations+syncs, or other regions, etc

// wait for all the operations assoc. with access region to complete
gasnet_wait_syncnb(handle);
```

3.4 Register-memory operations

Register-memory operations allow client code to avoid forcing communicated data to pass through the local memory system. Some interconnects may be able to take advantage of this capability and launch remote puts directly from registers or recieve remote gets directly into registers.

3.4.1 Value Put

3.4.1.1 gasnet_put_val, gasnet_put_nb_val, gasnet_put_nbi_val

`void gasnet_put_val` (*gasnet_node_t* **`node`**, *void* *__`dest`__, *gasnet_register_value_t* **`value`**, *size_t* **`nbytes`**);
`gasnet_handle_t gasnet_put_nb_val` (*gasnet_node_t* **`node`**, *void* *__`dest`__, *gasnet_register_value_t* **`value`**, *size_t* **`nbytes`**);
`void gasnet_put_nbi_val` (*gasnet_node_t* **`node`**, *void* *__`dest`__, *gasnet_register_value_t* **`value`**, *size_t* **`nbytes`**);

Register-to-remote-memory put - these functions take the value to be put as input parameter to avoid forcing outgoing values to local memory in client code. Otherwise, the behavior is identical to the memory-to-memory versions of put above. Requires: `nbytes > 0 && nbytes <= SIZEOF_GASNET_REGISTER_VALUE_T`. The value written to the target address is a direct byte copy of the 8**nbytes* low-order bits of value, written with the endianness appropriate for an *nbytes* integral value on the current architecture. The non-blocking forms of value put must be synchronized using the explicit or implicit synchronization functions defined above, as appropriate

3.4.2 Blocking Value Get

3.4.2.1 gasnet_get_val

`gasnet_register_value_t gasnet_get_val` (*gasnet_node_t* **node**, *void* ***src**, *size_t* **nbytes**);

This function returns the fetched value to avoid forcing incoming values through local memory (on architectures which pass the return value in a register). Otherwise, the behavior is identical to the memory-to-memory blocking get. Requires: `nbytes > 0 && nbytes <= SIZEOF_GASNET_REGISTER_VALUE_T`. The value returned is the one obtained by reading the *nbytes* bytes starting at the source address with the endianness appropriate for an *nbytes* integral value on the current architecture and setting the high-order bits (if any) to zero (i.e. no sign-extension)

3.4.3 Non-Blocking Value Get (explicit-handle)

This operates similarly to the blocking form of value get, but is split-phase. Non-blocking value gets are synchronized independently of all other operations in GASNet.

```
typedef ??? gasnet_valget_handle_t;
```

3.4.3.1 gasnet_get_nb_val, gasnet_wait_syncnb_valget

`gasnet_valget_handle_t gasnet_get_nb_val` (*gasnet_node_t* **node**, *void* ***src**, *size_t* **nbytes**);
`gasnet_register_value_t gasnet_wait_syncnb_valget` (*gasnet_valget_handle_t* **handle**);

`gasnet_get_nb_val` initiates a non-blocking value get and returns an explicit handle which **must** be synchronized using `gasnet_wait_syncnb_valget`. `gasnet_wait_syncnb_valget` synchronizes one such outstanding operation and returns the retrieved value as described for the blocking version. Note that `gasnet_valget_handle_t` and `gasnet_handle_t` are completely different datatypes and may not be intermixed (i.e. `gasnet_valget_handle_t` cannot be used with other synchronization functions, and `gasnet_handle_t` cannot be passed to `gasnet_wait_syncnb_valget`). The `gasnet_valget_handle_t` type is completely opaque (with no special "invalid" value), although implementors are recommended to make `sizeof(gasnet_valget_handle_t) <= sizeof(gasnet_register_value_t)` to facilitate register reuse. There is no try variant of value get synchronization, and no implicit-handle variant.

3.5 Barriers

The following functions can be used to execute a parallel split-phase barrier with the given barrier identifier across all nodes in the job. Note that the barrier wait/notify functions should only be called once (i.e. by one representative thread) on each node per barrier phase. The client must synchronize its own accesses to the barrier functions and ensure that only one thread is ever inside a GASNet barrier function at a time (esp. `gasnet_barrier_try()`).

```
#define GASNET_BARRIERFLAG_ANONYMOUS ???
#define GASNET_BARRIERFLAG_MISMATCH ???
```

3.5.1 gasnet_barrier_notify

`void gasnet_barrier_notify` (*int* **id**, *int* **flags**)

Execute the notification for a split-phase barrier, with a barrier value *id*. This is a non-blocking operation that completes immediately after noting the barrier value. No synchronization is performed on outstanding non-blocking memory operations.

Generates a fatal error if this is the second call to `gasnet_barrier_notify()` on this node since the last call to `gasnet_barrier_wait()` or the beginning of the program.

If *flags* == 0 then this is a "named" barrier notify that carries the given *id* value. If *flags* == `GASNET_BARRIERFLAG_ANONYMOUS`, then *id* is ignored and the barrier is anonymous - it has no specific value. If *flags* == `GASNET_BARRIERFLAG_MISMATCH`, then the subsequent `gasnet_barrier_wait()` call on every node will return `GASNET_ERR_BARRIER_MISMATCH` (i.e. allows the client to force a global mismatch error when a mismatch was detected locally).

3.5.2 gasnet_barrier_wait

`int gasnet_barrier_wait` (*int* `id`, *int* `flags`)

Execute the wait for a split-phase barrier, with a barrier value. This is a blocking operation that returns only after all remote nodes have called `gasnet_barrier_notify()`. No synchronization is performed on outstanding non-blocking memory operations .

Generates a fatal error if there were no preceding calls to `gasnet_barrier_notify()` on this node, or if this is the second call to `gasnet_barrier_wait()` (or successful call to `gasnet_barrier_try()`) since the last call to `gasnet_barrier_notify()` on this node. On a `GASNET_PAR` or `GASNET_PARSYNC` configuration, the thread calling `gasnet_barrier_notify()` is permitted to differ from the thread which calls the paired `gasnet_barrier_wait()`, but the ordering between the calls must still be maintained.

Returns `GASNET_ERR_BARRIER_MISMATCH` if *flags* is not equal to the *flags* value passed to the preceding `gasnet_barrier_notify()` call made by this node. Returns `GASNET_ERR_BARRIER_MISMATCH` if the *flags* value passed to `gasnet_barrier_notify()` on this or any other node was `GASNET_BARRIERFLAG_MISMATCH`. Returns `GASNET_ERR_BARRIER_MISMATCH` if *flags*==0 and the supplied *id* value doesn't match the *id* value provided in the preceding `gasnet_barrier_notify()` call made by this node. Returns `GASNET_ERR_BARRIER_MISMATCH` if any two nodes passed non-anonymous barrier values which didn't match during the `gasnet_barrier_notify()` calls which began this barrier phase. Otherwise, returns `GASNET_OK` to indicate that all nodes have called a matching `gasnet_barrier_notify()` and the barrier phase is complete.

3.5.3 gasnet_barrier_try

`int gasnet_barrier_try` (*int* `id`, *int* `flags`)

`gasnet_barrier_try()` functions similarly to `gasnet_wait()`, except that it always returns immediately. If the barrier has been notified by all nodes, the call behaves as a call to `gasnet_barrier_wait()` with the same barrier *id* and *flags*, and returns `GASNET_OK` (or `GASNET_ERR_BARRIER_MISMATCH` in the case a mismatch is detected). If the barrier has not yet been notified by some node, the call is a no-op and returns the value `GASNET_ERR_NOT_READY`.

Generates a fatal error if there were no preceding calls to `gasnet_barrier_notify()` on this node, or if this is the second call to `gasnet_barrier_wait()` (or successful call to `gasnet_barrier_try()`) since the last call to `gasnet_barrier_notify()` on this node.

3.6 Threading support

3.6.1 Thread-identification optimization

When compiled in the `GASNET_PAR` or `GASNET_PARSYNC` configurations, GASNet is capable of handling multiple client threads. It is likely that GASNet implementations will need to distinguish these threads, specifically they may need to store some metadata associated with each client thread. Unfortunately, the overhead of discovering the identity of a particular client thread making a GASNet call (hereafter termed "thread discovery") can have a non-trivial overhead on some threading systems (e.g. the cost of calling `pthread_self()` or `pthread_getspecific()`). Many of the simpler GASNet functions could have their performance dominated by this cost if they need to perform thread discovery on every call.

The following macros provide a way for the client to amortize the cost of thread discovery over many GASNet calls made by the same thread. This is an optimization which is *totally* optional - clients need not make any of the calls below to have a working system, although GASNet performance may suffer without it in a `GASNET_PAR` or `GASNET_PARSYNC` configuration on some platforms.

```
typedef void *gasnet_threadinfo_t;
```

`gasnet_threadinfo_t` is an opaque pointer representing the internal GASNet metadata associated with a particular client thread.

3.6.1.1 GASNET_GET_THREADINFO

`#define GASNET_GET_THREADINFO() ???`

Returns a value of type `gasnet_threadinfo_t` which represents the GASNet internal metadata associated with the current client thread. This `gasnet_threadinfo_t` value can be passed into or out of functions and may be posted for GASNet's use with `GASNET_POST_THREADINFO()`. May be called from anywhere in the client program, at any time after GASNet initialization. It is erroneous to hand-off this `gasnet_threadinfo_t` value to a different client thread.

3.6.1.2 GASNET_POST_THREADINFO

`#define GASNET_POST_THREADINFO(info) ???`

This macro may *optionally* be placed (followed by a semi-colon) at the top of functions which make calls to GASNet. It has no runtime semantics, but it may provide a performance boost on some implementations (especially in functions which make multiple calls to the extended API - e.g. it provides the implementation with a place for minimal per-function initialization or temporary storage that may be helpful in amortizing implementation-specific overheads). When used, it must appear only at the very beginning of a function or block (before any declarations or calls to the API in that function). It may not appear as a global declaration. The info argument must be a `gasnet_threadinfo_t` value acquired from a previous call to `GASNET_GET_THREADINFO()` on this thread.

3.6.1.3 GASNET_BEGIN_FUNCTION

`#define GASNET_BEGIN_FUNCTION() ???`

A convenience macro that may *optionally* be placed (followed by a semi-colon) at the top of functions which repeatedly make GASNet calls, to amortize the overhead of thread discovery on some implementations.

It has behavior equivalent to `GASNET_POST_THREADINFO(GASNET_GET_THREADINFO())`, however some implementations may choose to lazily postpone performing thread discovery until the first place where it is actually needed.

3.6.2 Thread management

3.6.2.1 gasnet_set_waitmode

`int gasnet_set_waitmode (`*int* `wait_mode)`

Optional call which gives the GASNet implementation a hint about how aggressively threads within blocking GASNet calls should contend for CPU resources. *wait_mode* must be one of the following recognized values:

`GASNET_WAIT_SPIN`
: contend aggressively for CPU resources while waiting (spin)

`GASNET_WAIT_BLOCK`
: yield CPU resources immediately while waiting (block)

`GASNET_WAIT_SPINBLOCK`
: spin for an implementation-dependent period, then block

Wait mode is a per-node hint which is permitted to differ across GASNet nodes.

Returns `GASNET_OK` on success.

Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet Version 2.0

Lawrence Berkeley National Lab Tech Report LBNL-56495 v2.0

Dan Bonachea
University of California, Berkeley

March 22, 2007

bonachea@cs.berkeley.edu

1 Introduction

1.1 Abstract

This document outlines a proposal for extending UPC's point-to-point memcpy library with support for explicitly non-blocking transfers, and non-contiguous (indexed and strided) transfers. Various portions of this proposal could stand alone as independent extensions to the UPC library. The designs presented here are heavily influenced by analogous functionality which exists in other parallel communication systems, such as MPI, ARMCI, Titanium, and network hardware API's such as Quadrics elan, Infiniband vapi, IBM LAPI and Cray X-1.

Each section contains proposed extensions to the libraries in the UPC Language Specification (section 7) and corresponding extensions to the GASNet communication system API.

1.2 Motivation

The UPC Language specification (version 1.1.1) provides a very minimal library for performing bulk-transfer communication. The *upc_memput*, *upc_memget* and *upc_memcpy* functions operate analogously to C99's *memcpy* function, and each provide the ability to move a single contiguous block of memory to/from locations specified by a pointer-to-local and pointer-to-shared or between locations specified by two pointers-to-shared. No further libraries are provided for directly expressing more complicated non-collective communication patterns - such as the movement of bulk data to/from non-contiguous locations (eg the column of a multi-dimensional array, or a set of locations in an irregular data structure). Non-contiguous access interfaces have historically been used to achieve speedups through communication aggregation - the transformation of fine-grained access patterns (which could naïvely be implemented using a large number of small messages), into more coarse-grained communication operations that improve network efficiency by sending larger messages and performing packing and unpacking at either end (possibly with hardware assistance). Furthermore, no mechanism is provided for the application programmer to express that a given communication operation can proceed independently with respect to other surrounding computation or communication operations

- a data independence property which has traditionally been used to obtain substantial parallel speedups by hiding communication latency with overlapped computation and other communication. The best an application programmer can currently do with the UPC 1.1.1 libraries is to express all non-collective bulk communication operations using a set of blocking contiguous transfers, and pray that very smart optimizers can transform this naïve access pattern (which has been forced by the restrictive library interface) into optimized communication operations that provide communication aggregation and overlap.

In an ideal world, UPC compilers could always automatically perform this transformation and achieve the maximal possible benefit from communication aggregation and overlap. However, the truth is that many factors force compilers to be overly conservative in such communication transformations and therefore the resulting communication pattern often falls quite short of the best one could hope to do with full application-level knowledge. Part of this failure is due to conservatism forced by features of UPC inherited from C (most notably pointer aliasing and separate compilation), however even assuming a perfect solution to these analysis problems, other fundamental sources of forced conservatism remain. In a complicated application, many important behavioral properties of the program are not directly expessed anywhere in the source program - they exist solely in the programmer's mind. Furthermore, many of the most useful properties (from an optimizer-writer's perspective) are not even possible to infer solely by inspection of the source program, even given an infinitely smart optimizer - because they depend on constraints such as the set of legal inputs, that are implicitly part of the program design but are often not expressed anywhere in the program, nor are they inferable solely from the program text. This is important because many of the most aggressive optimizations (such as some forms of static communication aggregation) need to make assumptions which are based on this sort of unexpressed knowledge in order to be safe (because for example, they might be unsafe in situations where the unexpressed assumptions are violated). Because the application programmer has this unexpressed algorithmic knowledge in his mind, he's in a unique position to direct these more aggressive optimizing transformations (given the proper tools), which even a perfect static compiler could not do without extra-linguistic help.

The UPC library extensions proposed in this document give the application programmer or library writer the tools necessary to request and express such beneficial transformations directly while tuning communication operations occuring in the application's critical path, rather than being constrained by the library interface to write communication in a naïve style and therefore being forced to rely upon a mythical perfect optimizer to automatically apply these important transformations (which we've just argued that the compiler often has insufficient information to legally perform). By allowing the programmer to directly and conveniently express communication aggregation and overlap in places where the algorithmic data dependencies allow, compiler implementors can focus their efforts on ensuring the requested communication is performed as efficiently as possible - for example leveraging available network hardware capabilities for non-blocking transfers and non-contiguous access.

From the perspective of GASNet as a compilation target, we want to provide interfaces for non-blocking and non-contiguous accesses to support the implementation of such language-level libraries, and additionally support automated communication optimizations that the compiler may apply to transform fine-grained communication patterns into bulk non-blocking and/or non-contiguous operations. Furthermore, we want the ability to implement and tune each such operation in the way most appropriate for the underlying network hardware, taking advantage of the wide variety of support for non-contiguous access available on modern HPC networks.

1.3 Implementation Notes

All of the proposed extensions described in this document have been implementated and are available as a prototype implementation in the Berkeley UPC compiler, version 2.4.0 (http://upc.lbl.gov). All functions in the prototype implementation operate exactly as described in this document *with* the notable exception that all functions, types and constants named using the prefix *bupc_* instead of *upc_*. This naming convention reflects the fact that these extensions are not currently part of the official UPC language specification.

The proposed GASNet extensions are also implemented exactly as described in this document, starting as a prototype implementation in GASNet 1.4. This documentation will soon be merged into the GASNet 2.0 specification.

2 Explicit-handle non-blocking bulk-contiguous operations

The following functions provide non-blocking, split-phase memory access to shared data. All such non-blocking operations require an initiation (e.g., a put or get) and a subsequent synchronization on the completion of that operation before the result is guaranteed. These are "explicit-handle" non-blocking operations because the initiation function returns an explicit handle value, which must be passed to the synchronization function for completing the corresponding operation.

2.1 Non-blocking explicit handle type

```
type  upc_handle_t
value UPC_COMPLETE_HANDLE
```

The explicit-handle non-blocking data transfer functions return a *upc_handle_t* value to represent the non-blocking operation in flight. *upc_handle_t* is an opaque private data type whose contents are implementation-defined, with one exception - every implementation must provide a value corresponding to an "invalid" handle ($UPC\_COMPLETE\_HANDLE$) and furthermore this value must be the result of setting all the bits in the *upc_handle_t* data type to zero. Implementors are free to define the *upc_handle_t* type to be any reasonable and appropriate size, although they are recommended to use a type which fits within a single standard register on the target architecture. In any case, the data type should be wide enough to express at least $2^{16} - 1$ different handle values, to prevent limiting the number of non-blocking operations in progress due to the number of handles available.

It is legal for threads to pass *upc_handle_t* values into function callees or back to function callers. However, *upc_handle_t* values are thread-specific. In other words, it is an error to obtain a handle value by initiating a non-blocking operation on one thread, and later pass that handle value into a synchronization function from a different thread.

Any explicit-handle, non-blocking initiation operation may return the value $UPC\_COMPLETE\_HANDLE$ to indicate that the requested operation was completed synchronously. It is always an error to discard the *upc_handle_t* value for an explicit-handle operation in-flight - i.e. to initiate an operation and never synchronize on its completion.

2.2 Explicit-handle non-blocking operations

```
upc_handle_t upc_memcpy_async(shared void *dst, shared const void *src, size_t n);
upc_handle_t upc_memget_async(       void *dst, shared const void *src, size_t n);
upc_handle_t upc_memput_async(shared void *dst,        const void *src, size_t n);
upc_handle_t upc_memset_async(shared void *dst, int c, size_t n);
```

These operations have the same semantics as the corresponding functions defined in the UPC Language Specification section 7.2.5, except they are split-phase. The specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined.

2.3 Explicit-handle non-blocking synchronization

```
void upc_waitsync(upc_handle_t handle);
int upc_trysync(upc_handle_t handle);
```

Synchronize on the completion of a single specified explicit-handle non-blocking operation that was initiated by the calling thread. *upc_waitsync*() blocks until the specified operation has completed (or returns immediately if it has already completed). In any case, the handle value is "dead" after *upc_waitsync*() returns and may not be passed to future synchronization operations. *upc_trysync*() always returns immediately, with a non-zero value if the operation is complete (at which point the handle value is "dead", and may not be used in future synchronization operations), or zero if the operation is not yet complete and future synchronization is necessary to complete the corresponding operation. It is legal to pass *UPC_COMPLETE_HANDLE* as input to these functions - *upc_waitsync* (*UPC_COMPLETE_HANDLE*) returns immediately and *upc_trysync* (*UPC_COMPLETE_HANDLE*) returns non-zero. It is an error to pass a *upc_handle_t* value (other than *UPC_COMPLETE_HANDLE*) for an operation which has already been successfully synchronized using one of the explicit-handle synchronization functions.

Note that the order in which non-blocking operations complete is intentionally unspecified - the system is free to coalesce and/or reorder non-blocking operations with respect to other blocking or non-blocking operations, or operations initiated from a separate thread - the only ordering constraints that must be satisfied are those explicitly enforced using the synchronization functions (i.e. the non-blocking operation is only guaranteed to occur somewhere in the interval between initiation and successful synchronization on that operation).

Implementors should attempt to make the non-blocking initiation operations return as quickly as possible - however in some cases (e.g. when a large number of non-blocking operations have been issued or the network is otherwise busy) it may be necessary to block temporarily while waiting for the network to become available. In any case, all implementations must support at least $2^{16}-1$ non-blocking operations in-progress per thread - that is, each thread is free to issue up to $2^{16}-1$ non-blocking operations before issuing a sync operation, and the implementation must handle this correctly without deadlock or livelock. Additionally, note that non-blocking operations proceed independently of barriers and other forms of inter-thread synchronization - these are not a substitute for *upc_waitsync*/*upc_trysync*.

Example: The following example demonstrates an explicitly asynchronous nearest neighbor exchange of data. We assume a regular domain decomposition in the data array A which is blocked in shared space. Each thread initiates a fetch of the neighbor data into local buffers, then performs independent computation while the communication proceeds overlapped in the background.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS];
double leftdata[BLKSZ];
double rightdata[BLKSZ];
upc_handle_t leftfetch_handle = UPC_COMPLETE_HANDLE;
upc_handle_t rightfetch_handle = UPC_COMPLETE_HANDLE;

if (MYTHREAD > 0) /* initiate fetch of data from left neighbor */
 leftfetch_handle = upc_memget_async(leftdata, &(A[BLKSZ*(MYTHREAD-1)]),BLKSZ*sizeof(double));
 if (MYTHREAD < THREADS-1)  /* initiate fetch of data from right neighbor */
 rightfetch_handle=upc_memget_async(rightdata,&(A[BLKSZ*(MYTHREAD+1)]), BLKSZ*sizeof(double));

/* perform some independent computations here */

upc_waitsync(leftfetch_handle); /* block for completion of communication, if necessary */
upc_waitsync(rightfetch_handle);

/* now safe to operate on leftdata and rightdata */
```

2.4 Multiple explicit-handle non-blocking synchronization

The following convenience functions assist in synchronizing arrays of explicit handles:

```
void upc_waitsync_all (upc_handle_t *ph, size_t numhandles);
int  upc_trysync_all  (upc_handle_t *ph, size_t numhandles);
void upc_waitsync_some(upc_handle_t *ph, size_t numhandles);
int  upc_trysync_some (upc_handle_t *ph, size_t numhandles);
```

These functions synchronize on the completion of an array of explicit handles (all of which were created by the calling thread). *numhandles* specifies the number of handles in the provided array of handles. *upc_waitsync_all* blocks until all the specified operations have completed (or returns immediately if they have all already completed). *upc_trysync_all* always returns immediately, with a non-zero value if all the specified operations have completed, or a zero value if one or more of the operations is not yet complete and future synchronization is necessary to complete some of the operations. *upc_waitsync_some* blocks until at least one *incomplete* handle in the list has completed (where the incomplete handles are those which are not $UPC\_COMPLETE\_HANDLE$). *upc_trysync_some* always returns immediately, with a non-zero value if at least one incomplete handle in the provided array has completed, or a zero value if none of the incomplete handles in the provided array has completed.

All of these functions will modify the provided array to reflect completions - handles whose operations have completed are overwritten with the value $UPC\_COMPLETE\_HANDLE$, and the client may test against this value upon return to determine which operations are complete and which are still pending.

It is legal to pass the value $UPC\_COMPLETE\_HANDLE$ in some of the array entries, and the functions will ignore all such entries so that they have no effect on behavior. In the case where all entries in the array are $UPC\_COMPLETE\_HANDLE$ or *numhandles* $== 0$, then the wait variants will return immediately and the try variants will return immediately with a non-zero value to indicate success.

3 Implicit-handle non-blocking bulk-contiguous operations

The following functions provide non-blocking, split-phase access to shared data. All such non-blocking operations require an initiation (e.g., a put or get) and a subsequent synchronization on the completion of that operation before the result is guaranteed. These are "implicit-handle" non-blocking operations because the initiation function does not return a handle - rather, the operation becomes associated with an implicit handle owned by the calling thread, and all implicit-handle operations are synchronized together using a call to an implicit-handle synchronization function. These operations have the same (weak) ordering guarantees which apply to the explicit-handle variants.

3.1 Implicit-handle non-blocking operations

```
void upc_memcpy_asynci(shared void *dst, shared const void *src, size_t n);
void upc_memget_asynci(       void *dst, shared const void *src, size_t n);
void upc_memput_asynci(shared void *dst,        const void *src, size_t n);
void upc_memset_asynci(shared void *dst, int c, size_t n);
```

These operations have the same semantics as the corresponding functions defined in the UPC Language Specification section 7.2.5, except they are split-phase. The specified operation is initiated with a call to the above functions. The operation is not guaranteed to be complete until after the next successful call to *upc_waitsynci* or *upc_trysynci* made by the initiating thread (unless access region synchronization is in effect, as explained in section 4). The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined.

3.2 Implicit-handle non-blocking synchronization

The following functions are used to synchronize implicit-handle non-blocking operations:

```
void upc_waitsynci();
int upc_trysynci();
```

These functions synchronize the set of non-blocking implicit-handle operations previously issued by the calling thread outside any access region, and not yet synchronized through a successful implicit-handle synchronization. *upc_waitsynci* blocks until all operations in this set have completed (indicating these operations have been successfully synchronized). *upc_trysynci* tests whether all operations in the set have completed, and returns a non-zero value if so (which indicates these operations have been successfully synchronized) or zero otherwise (in which case **none** of these operations may be considered successfully synchronized).

If there are no outstanding implicit-handle operations (i.e., the set is empty), then *upc_waitsynci* returns immediately, and *upc_trysynci* returns immediately with a non-zero value to indicate success.

These functions notably do **not** synchronize any outstanding explicit-handle operations - those operations proceed independently and must be synchronized using the explicit-handle synchronization functions. Because the set of operations is determined dynamically and not lexically, implicit-handle synchronization functions can synchronize operations initiated within other function frames by the calling thread (but this cannot affect the correctness of correctly synchronized code - at worst it oversynchronizes).

Example: Here is the same example from section 2.3, written using implicit-handle synchronization. The example is semantically equivalent, but more concise as there are no explicit handles to manage.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS];
double leftdata[BLKSZ];
double rightdata[BLKSZ];

if (MYTHREAD > 0) /* initiate fetch of data from left neighbor */
  upc_memget_asynci(leftdata, &(A[BLKSZ*(MYTHREAD-1)]), BLKSZ*sizeof(double));
if (MYTHREAD < THREADS-1)  /* initiate fetch of data from right neighbor */
  upc_memget_asynci(rightdata, &(A[BLKSZ*(MYTHREAD+1)]), BLKSZ*sizeof(double));

/* perform some independent computations here */

upc_waitsynci(); /* block for completion of communication, if necessary */

/* now safe to operate on leftdata and rightdata */
```

4 Access region synchronization

In some cases, it may be useful or desirable to initiate a number of non-blocking operations (possibly without knowing how many at compile-time) and synchronize them at a later time using a single, fast synchronization. Simple implicit handle synchronization may not be appropriate for this situation if there are intervening implicit accesses which are not to be synchronized. This situation could be handled using explicit-handle non-blocking operations and *upc_waitsync_all*, but this may not be desirable because it requires managing an array of handles (which may be inconvenient or costly when the number of operations is not known until runtime). To handle these cases, we provide *access region* synchronization, described below. It provides a useful middle ground between implicit and explicit handles in the expressiveness versus conciseness tradeoff.

4.1 Access region functions

```
void         upc_begin_accessregion();
upc_handle_t upc_end_accessregion();
```

The *upc_begin_accessregion* and *upc_end_accessregion* functions are used to define an access region - any statements which execute on the calling thread after a begin call and before the next end call are said to be *inside* the region. The begin and end calls must be paired, and may not be nested or the results are undefined. It is erroneous to call any implicit-handle synchronization function (section 3.2) inside an access region. All implicit-handle non-blocking operations initiated inside the region by the functions in section 3.1 become *associated* with the abstract access region handle being constructed. *upc_end_accessregion* returns an explicit handle which collectively represents all the associated operations (those implicit-handle operations initiated within the access region). This handle must later be passed to the regular explicit-handle synchronization functions in sections 2.3 and 2.4, and will be successfully synchronized when **all** of the associated operations initiated in the access region have completed. The associated operations are **not** synchronized by subsequent calls to the implicit-handle synchronization functions occurring after the access region (e.g. *upc_waitsynci*). Explicit-handle operations initiated within the access region operate as usual and do **not** become associated with the access region.

Example: Here is the same example from section 2.3, written using an access region. The example is semantically equivalent, but more concise as there is only one handle to manage.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS];
double leftdata[BLKSZ];
double rightdata[BLKSZ];

upc_begin_accessregion(); // begin the access region

if (MYTHREAD > 0) /* initiate fetch of data from left neighbor */
  upc_memget_asynci(leftdata, &(A[BLKSZ*(MYTHREAD-1)]), BLKSZ*sizeof(double));
if (MYTHREAD < THREADS-1)  /* initiate fetch of data from right neighbor */
  upc_memget_asynci(rightdata, &(A[BLKSZ*(MYTHREAD+1)]), BLKSZ*sizeof(double));

// end the access region and get the handle
upc_handle_t handle = upc_end_accessregion();

/* perform some independent computations here */

upc_waitsync(handle); /* block for completion of communication, if necessary */

/* now safe to operate on leftdata and rightdata */
```

Example: A more complicated example of an access region.

```
upc_begin_accessregion(); // begin the access region

upc_memput_asynci(...); // becomes associated with access region
while (...) {
  upc_memget_asynci(...); // becomes associated with access region
}

// unrelated explicit-handle operation not associated with access region
upc_handle_t h2 = upc_memget_async(...);
upc_waitsync(h2);

// end the access region and get the handle
upc_handle_t handle = upc_end_accessregion();

.... // other code, which may include unrelated implicit or explicit handle
     // operations+syncs, or other access regions, etc

// wait for all the operations associated with the access region to complete
upc_waitsync(handle);
```

5 Indexed/Vector memcpy operations

The indexed memcpy functions provide a general mechanism to express an operation which gathers data from arbitrary source regions of memory and scatters data into arbitrary destination regions of memory. Expressing such a data movement pattern as a single high-level operation (as opposed to many small, contiguous operations) allows for more aggressive optimization of the data movement within the UPC implementation - for example, tuning the transfer mechanism for maximal performance on the given memory hierarchy or taking advantage of platform-specific scatter/gather support in network hardware. All the functions are non-collective - they are called by a single thread to initiate an indexed memory copy transfer.

5.1 Common Requirements

The total amount of data specified by the source regions must equal the total amount of data specified by the destination regions (although the individual regions in each list need not be of equal size). In other words, counts and lengths in the source and destination lists need not match, so long as they both specify the same total amount of data. The effect of the operation is that data is copied from the source regions, in the order specified by *srclist*, to the destination regions, in the order specified by *dstlist*. Note the contents of the destination regions is undefined while the operation is in progress (i.e. the actual order in which the writes take place is undefined), and if the contents of the source regions change while the operation is in progress the result is undefined.

The destination regions must be completely disjoint and must not overlap with any source regions, otherwise the result is undefined. Source regions are permitted to overlap with each other.

If *dstcount* and *srccount* are zero, the operation is a no-op and the other arguments are ignored.

5.2 Possible Design A - List of variable-sized regions

```
typedef struct {
  void *addr;
  size_t len;
} upc_pmemvec_t;

typedef struct {
  shared void *addr; // treated as a (shared [] char *) - ie. no wrapping
  size_t len;
} upc_smemvec_t;
```

A *upc_pmemvec_t* specifies a contiguous region of local memory valid on the current thread starting at base address *addr* and extending for *len* bytes. A *upc_smemvec_t* specifies a contiguous region of shared memory with affinity to a single thread, starting at base address *addr* and extending for *len* bytes. In both cases *len* may be zero, in which case that entry is ignored.

```
void upc_memcpy_vlist(size_t dstcount, upc_smemvec_t const dstlist[],
                      size_t srccount, upc_smemvec_t const srclist[]);
void upc_memput_vlist(size_t dstcount, upc_smemvec_t const dstlist[],
                      size_t srccount, upc_pmemvec_t const srclist[]);
void upc_memget_vlist(size_t dstcount, upc_pmemvec_t const dstlist[],
                      size_t srccount, upc_smemvec_t const srclist[]);
```

```
upc_handle_t upc_memcpy_vlist_async(size_t dstcount, upc_smemvec_t const dstlist[],
                                    size_t srccount, upc_smemvec_t const srclist[]);
upc_handle_t upc_memput_vlist_async(size_t dstcount, upc_smemvec_t const dstlist[],
                                    size_t srccount, upc_pmemvec_t const srclist[]);
upc_handle_t upc_memget_vlist_async(size_t dstcount, upc_pmemvec_t const dstlist[],
                                    size_t srccount, upc_smemvec_t const srclist[]);
```

- *srclist* and *dstlist* specify a list of contiguous memory regions to be used as the source and destination for the memory transfer. Each *upc_smemvec_t* entry is permitted to specify data with affinity to a different thread.
- *srccount* and *dstcount* indicate the number of region entries in the *srclist* and *dstlist* array, respectively.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined. The *srclist* and *dstlist* arrays must remain valid and unchanged until the operation is complete.

Dan's Comments
PROS: good for specifying bounding boxes, efficiently allows packing in a contiguous buffer at either end, allows multiple remote affinities, mirrors the UPC-IO List IO interface
CONS: bad for vectorization, high metadata space consumption, full generality provided may not map well to more restrictive lower-level scatter/gather network layers

Example: The following example demonstrates the use of *upc_memget_vlist* (Design A) to fetch some individual elements, a group of elements, and an entire block in a single operation into a single, contiguous local buffer. For demonstration purposes the data was fetched from shared memory with affinity to different threads, although this need not always be the case.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS]; /* assume THREADS >= 3 */
upc_smemvec_t srclist[] = {
        { &(A[14]), sizeof(double) },     /* element 14 (from thread 0) */
        { &(A[20]), sizeof(double) },     /* element 20 (from thread 0) */
        { &(A[100]), 50*sizeof(double) }, /* elements 100..149 (from thread 1) */
        { &(A[2*BLKSZ]), BLKSZ*sizeof(double) } /* entire block (from thread 2) */
      };
double mybuf[52+BLKSZ];
upc_pmemvec_t dstlist[] = { { mybuf, sizeof(mybuf) } };

upc_memget_vlist(1, dstlist, 4, srclist);

/* compute on contents of mybuf */
```

5.2.1 GASNet interface for Indexed/Vector Design A

```
typedef struct {
  void *addr;
  size_t len;
} gasnet_memvec_t;

void gasnet_putv_bulk(gasnet_node_t dstnode,
                      size_t dstcount, gasnet_memvec_t const dstlist[],
                      size_t srccount, gasnet_memvec_t const srclist[]);
void gasnet_getv_bulk(size_t dstcount, gasnet_memvec_t const dstlist[],
                      gasnet_node_t srcnode,
                      size_t srccount, gasnet_memvec_t const srclist[]);

gasnet_handle_t gasnet_putv_nb_bulk(gasnet_node_t dstnode,
                                    size_t dstcount, gasnet_memvec_t const dstlist[],
                                    size_t srccount, gasnet_memvec_t const srclist[]);
gasnet_handle_t gasnet_getv_nb_bulk(size_t dstcount, gasnet_memvec_t const dstlist[],
                                    gasnet_node_t srcnode,
                                    size_t srccount, gasnet_memvec_t const srclist[]);

void gasnet_putv_nbi_bulk(gasnet_node_t dstnode,
                          size_t dstcount, gasnet_memvec_t const dstlist[],
                          size_t srccount, gasnet_memvec_t const srclist[]);
void gasnet_getv_nbi_bulk(size_t dstcount, gasnet_memvec_t const dstlist[],
                          gasnet_node_t srcnode,
                          size_t srccount, gasnet_memvec_t const srclist[]);
```

These vector put/get operations operate exactly analogously to the contiguous *gasnet_put/get_bulk* functions - ie. unaligned access is permitted and the user cannot free or modify the source data until after sync. Additionally, the *srclist/dstlist* metadata input arrays must remain valid and unchanged until after sync.

Note: this GASNet interface is strictly point-to-point - only one remote node may be specified (as opposed to the UPC level interface which allows (shared void *) addresses with arbitrary affinity). This is primarily motivated by the fact that current network hardware support for scatter/gather does not accommodate multi-remote-node operations, and therefore we wish to avoid a second pass over the address list within GASNet to separate the addresses by remote node (the UPC runtime can just as easily do that during its address translation pass). Also, other GASNet clients (such as Titanium) do not want to pay for the additional, unneeded generality.

5.3 Possible Design B - List of fixed-size regions

```
void upc_memcpy_ilist(size_t dstcount, shared       void * const dstlist[], size_t dstlen,
                      size_t srccount, shared const void * const srclist[], size_t srclen);
void upc_memput_ilist(size_t dstcount, shared       void * const dstlist[], size_t dstlen,
                      size_t srccount,        const void * const srclist[], size_t srclen);
void upc_memget_ilist(size_t dstcount,              void * const dstlist[], size_t dstlen,
                      size_t srccount, shared const void * const srclist[], size_t srclen);

upc_handle_t upc_memcpy_ilist_async(size_t dstcount, shared       void * const dstlist[],
                                    size_t dstlen,
                                    size_t srccount, shared const void * const srclist[],
                                    size_t srclen);
upc_handle_t upc_memput_ilist_async(size_t dstcount, shared       void * const dstlist[],
                                    size_t dstlen,
                                    size_t srccount,        const void * const srclist[],
                                    size_t srclen);
upc_handle_t upc_memget_ilist_async(size_t dstcount,              void * const dstlist[],
                                    size_t dstlen,
                                    size_t srccount, shared const void * const srclist[],
                                    size_t srclen);
```

These functions copy data elements as *srccount* contiguous regions of memory with fixed length *srclen* from base addresses $srclist[0]...srclist[srccount - 1]$, and place the data as into contiguous regions of memory with length *dstlen* at base addresses $dstlist[0]...dstlist[dstcount - 1]$.

- *srclist* and *dstlist* specify a list of element addresses be used as the source and destination for the memory transfer. Each entry is permitted to specify data with affinity to a different thread.
- *srccount* and *dstcount* indicate the number of elements in the *srclist* and *dstlist* array, respectively.
- *srclen* and *dstlen* specify the length in bytes for each contiguous region referenced by *srclist* and *dstlist*. The two need not be equal, but must both be greater than zero.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined. The *srclist* and *dstlist* arrays must remain valid and unchanged until the operation is complete.

Dan's Comments
PROS: minimizes metadata space overhead, allows multiple remote affinities, efficiently allows packing in a contiguous buffer at either end
CONS: can't efficiently handle different-sized regions in a single operation, some platforms may perform badly when *srclen* and *dstlen* are unequal.

Example: The following example demonstrates the use of *upc_memget_ilist* (Design B) to fetch some individual elements into a single, contiguous local buffer. For demonstration purposes the data was fetched from shared memory with affinity to different threads, although this need not always be the case. Note that each region of source memory in a single operation is constrained to be the same size (although it needn't match the underlying element size). The most concise way to fetch many regions of different sizes with this interface is to use a separate operation for each region size (and possibly use asynchronous operations to improve concurrency).

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS]; /* assume THREADS >= 2 */
shared void * srclist[] = {
        &(A[14]), &(A[15]), &(A[16]),   /* element 14..16 (from thread 0) */
        &(A[100]), &(A[110])            /* element 100 and 110 (from thread 1) */
      };
double mybuf[5];
void * dstlist[] = { &mybuf };

upc_memget_ilist(1, dstlist, 5*sizeof(double),
                5, srclist, sizeof(double));

/* compute on contents of mybuf */
```

5.3.1 GASNet interface for Indexed/Vector Design B

```
void gasnet_puti_bulk(gasnet_node_t dstnode,
                      size_t dstcount, void * const dstlist[], size_t dstlen,
                      size_t srccount, void * const srclist[], size_t srclen);
void gasnet_geti_bulk(size_t dstcount, void * const dstlist[], size_t dstlen,
                      gasnet_node_t srcnode,
                      size_t srccount, void * const srclist[], size_t srclen);

gasnet_handle_t
gasnet_puti_nb_bulk(gasnet_node_t dstnode,
                    size_t dstcount, void * const dstlist[], size_t dstlen,
                    size_t srccount, void * const srclist[], size_t srclen);
gasnet_handle_t
gasnet_geti_nb_bulk(size_t dstcount, void * const dstlist[], size_t dstlen,
                    gasnet_node_t srcnode,
                    size_t srccount, void * const srclist[], size_t srclen);

void gasnet_puti_nbi_bulk(gasnet_node_t dstnode,
                          size_t dstcount, void * const dstlist[], size_t dstlen,
                          size_t srccount, void * const srclist[], size_t srclen);
void gasnet_geti_nbi_bulk(size_t dstcount, void * const dstlist[], size_t dstlen,
                          gasnet_node_t srcnode,
                          size_t srccount, void * const srclist[], size_t srclen);
```

These indexed put/get operations operate exactly analogously to the contiguous *gasnet_put/get_bulk* functions - ie. unaligned access is permitted and the user cannot free or modify source data until after sync. Additionally, the *srclist/dstlist* metadata input arrays must remain valid and unchanged until after sync.

6 Strided memcpy

The strided memcpy functions are a special case of the indexed memcpy functions, with an interface specialized for efficiently expressing copies of arbitrary rectangular sections of dense multi-dimensional arrays. All the functions are non-collective - they are called by a single thread to initiate a strided memory copy transfer.

6.1 Possible Design A - fixed region size/stride (2-d rectangular array section)

This design option has a relatively simple but restrictive interface - operating on fixed size regions (chunks), with a single fixed stride through linear memory between each chunk.

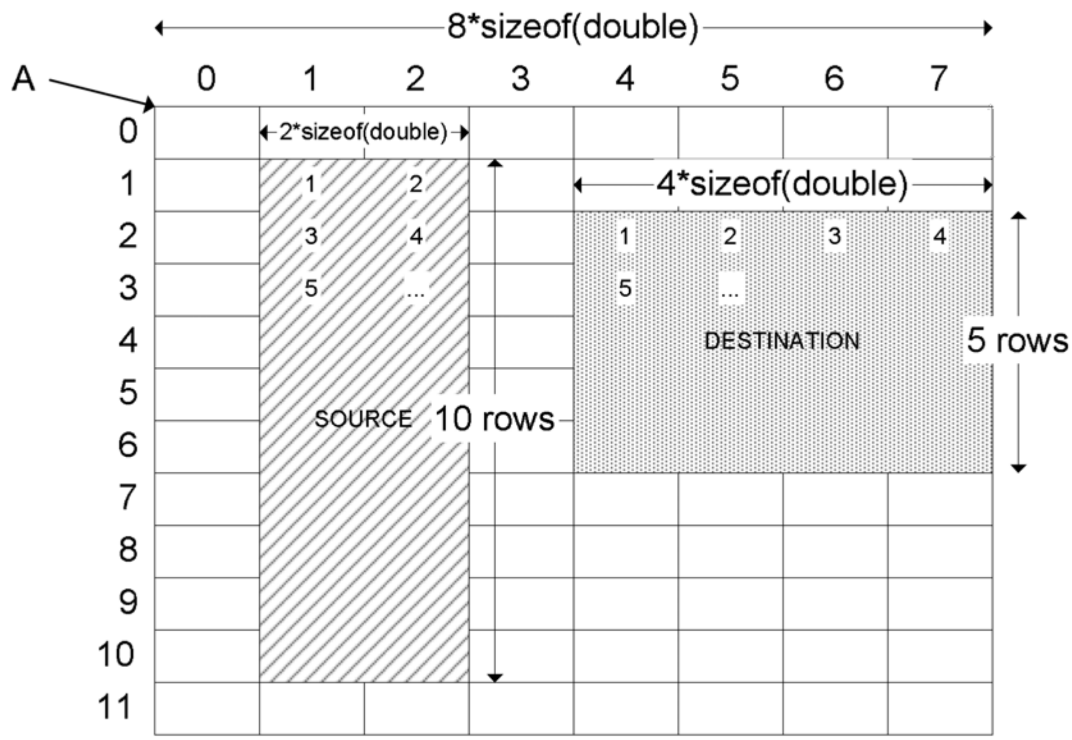
```
void upc_memcpy_fstrided(shared void *dstaddr,  size_t dstchunklen,
                         size_t dstchunkstride, size_t dstchunkcount,
                         shared void *srcaddr,  size_t srcchunklen,
                         size_t srcchunkstride, size_t srcchunkcount);
void upc_memput_fstrided(shared void *dstaddr,  size_t dstchunklen,
                         size_t dstchunkstride, size_t dstchunkcount,
                                void *srcaddr,  size_t srcchunklen,
                         size_t srcchunkstride, size_t srcchunkcount);
void upc_memget_fstrided(       void *dstaddr,  size_t dstchunklen,
                         size_t dstchunkstride, size_t dstchunkcount,
                         shared void *srcaddr,  size_t srcchunklen,
                         size_t srcchunkstride, size_t srcchunkcount);

upc_handle_t upc_memcpy_fstrided_async(shared void *dstaddr,  size_t dstchunklen,
                                       size_t dstchunkstride, size_t dstchunkcount,
                                       shared void *srcaddr,  size_t srcchunklen,
                                       size_t srcchunkstride, size_t srcchunkcount);
upc_handle_t upc_memput_fstrided_async(shared void *dstaddr,  size_t dstchunklen,
                                       size_t dstchunkstride, size_t dstchunkcount,
                                              void *srcaddr,  size_t srcchunklen,
                                       size_t srcchunkstride, size_t srcchunkcount);
upc_handle_t upc_memget_fstrided_async(       void *dstaddr,  size_t dstchunklen,
                                       size_t dstchunkstride, size_t dstchunkcount,
                                       shared void *srcaddr,  size_t srcchunklen,
                                       size_t srcchunkstride, size_t srcchunkcount);
```

- *srcaddr* and *dstaddr* base addresses for the source and destination regions, treated as a (shared [] char *) - ie. no wrapping
- *srcchunklen* and *dstchunklen* length of each chunk in bytes
- *srcchunkstride* and *dstchunkstride* number of bytes between the start of each chunk (must be >= *chunklen*)
- *srcchunkcount* and *dstchunkcount* number of chunks

The total data length in the source and destination must be equal, i.e., $srcchunklen * srcchunkcount == dstchunklen * dstchunkcount$. If the source locations overlap any destination locations, the result is undefined. If *srcchunklen* ∗ *srcchunkcount* and *dstchunklen* ∗ *dstchunkcount* are zero, the operation is a no-op and the other arguments are ignored.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined.

Dan's Comments
PROS: simplicity of interface arguments, regular sparse access, efficiently allows packing in a contiguous buffer at either end, compact metadata, allows the copied region and underlying arrays to differ in shape at the source and destination
CONS: lacks generality - unable to retrieve an arbitrary rectangular array section in more than two dimensions




```
shared [] double A[12][8];

upc_memcpy_fstrided(&(A[2][4]), 4*sizeof(double), 8*sizeof(double), 5,
                    &(A[1][1]), 2*sizeof(double), 8*sizeof(double), 10);
```

Figure 1: Example of upc_memcpy_fstrided, Design A

6.2 Possible Design B - N-d rectangular array section

```
void upc_memcpy_strided(shared       void *dstaddr, const size_t dststrides[],
                        shared const void *srcaddr, const size_t srcstrides[],
                        const size_t count[], size_t stridelevels);
void upc_memput_strided(shared       void *dstaddr, const size_t dststrides[],
                               const void *srcaddr, const size_t srcstrides[],
                        const size_t count[], size_t stridelevels);
void upc_memget_strided(             void *dstaddr, const size_t dststrides[],
                        shared const void *srcaddr, const size_t srcstrides[],
                        const size_t count[], size_t stridelevels);

upc_handle_t upc_memcpy_strided_async(shared       void *dstaddr, const size_t dststrides[],
                                      shared const void *srcaddr, const size_t srcstrides[],
                                      const size_t count[], size_t stridelevels);
upc_handle_t upc_memput_strided_async(shared       void *dstaddr, const size_t dststrides[],
                                             const void *srcaddr, const size_t srcstrides[],
                                      const size_t count[], size_t stridelevels);
upc_handle_t upc_memget_strided_async(             void *dstaddr, const size_t dststrides[],
                                      shared const void *srcaddr, const size_t srcstrides[],
                                             const size_t count[], size_t stridelevels);
```

- *srcaddr* Source starting address of the data block to copy, treated as a (shared [] char *) (i.e., no wrapping).
- *srcstrides* Source array of positive stride distances in bytes to move along each dimension. (*stridelevels* entries)
- *dstaddr* Destination starting address of the data block to receive the copy, treated as a (shared [] char *) (i.e., no wrapping).
- *dststrides* Destination array of positive stride distances in bytes to move along each dimension. (*stridelevels* entries)
- *count* Slice size in each dimension. $count[0]$ should be the number of bytes of contiguous data in the leading (rightmost) dimension. ($stridelevels + 1$ entries)
- *stridelevels* The level of strides (for an N-d array copy, one generally sets $stridelevels == (N - 1)$).

If the source locations overlap any destination locations, the result is undefined. If *stridelevels* is zero, the operation is a contiguous copy of $count[0]$ bytes, and the *srcstrides* and *dststrides* arguments are ignored. If any entry in $count[0..stridelevels]$ is zero, the operation is a no-op and the other arguments are ignored. The dimensional strides in *srcstrides* and *dststrides* must be monotonically increasing and must not specify overlapping locations - more specifically, $srcstrides[0] \geq count[0] \ \wedge \ \forall i \in [1..(stridelevels - 1)] \mid srcstrides[i] \geq (count[i] * srcstrides[i-1])$, and accordingly for *dststrides*.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined. The *srcstrides*, *dststrides*, and *count* arrays must remain valid and unchanged until the operation is complete.

Dan's Comments
PROS: fully general - can take an arbitrary rectangular section from a dense rectangular array of any dimensionality, efficiently allows packing in a contiguous buffer at either end, allows the underlying arrays to differ in shape at the source and destination
CONS: interface complexity may intimidate novice users, does not allow the copied region to differ in shape at the source and destination (i.e., the rectangular section being copied must have the same extents in N-d space at either end)

Example: To put a 3-d block of data, shaped 2x3x4, starting at location (5, 6, 7) in A to B in location (8, 9, 10), the arguments to *upc_memput_strided* can be set as follows:

```
double A[11][12][13]; /* local array */
shared [] double B[14][15][16]; /* remote array */

void * srcaddr;
shared void * dstaddr;
size_t count[3];
size_t stridelevels;

srcaddr = &(A[5][6][7]);
srcstrides[0] = 13 * sizeof(double);      /* stride in bytes for the rightmost dimension */
srcstrides[1] = 12 * 13 * sizeof(double); /* stride in bytes for the middle dimension */
dstaddr = &(B[8][9][10]);
dststrides[0] = 16 * sizeof(double);      /* stride in bytes for the rightmost dimension */
dststrides[1] = 15 * 16 * sizeof(double); /* stride in bytes for the middle dimension */
count[0] = 4 * sizeof(double); /* bytes of contiguous data (width in rightmost dimension)*/
count[1] = 3; /* width in middle dimension */
count[2] = 2; /* width in leftmost dimension */
stridelevels = 2;

upc_memput_strided(srcaddr, dststrides, dstaddr, srcstrides, count, stridelevels);
```

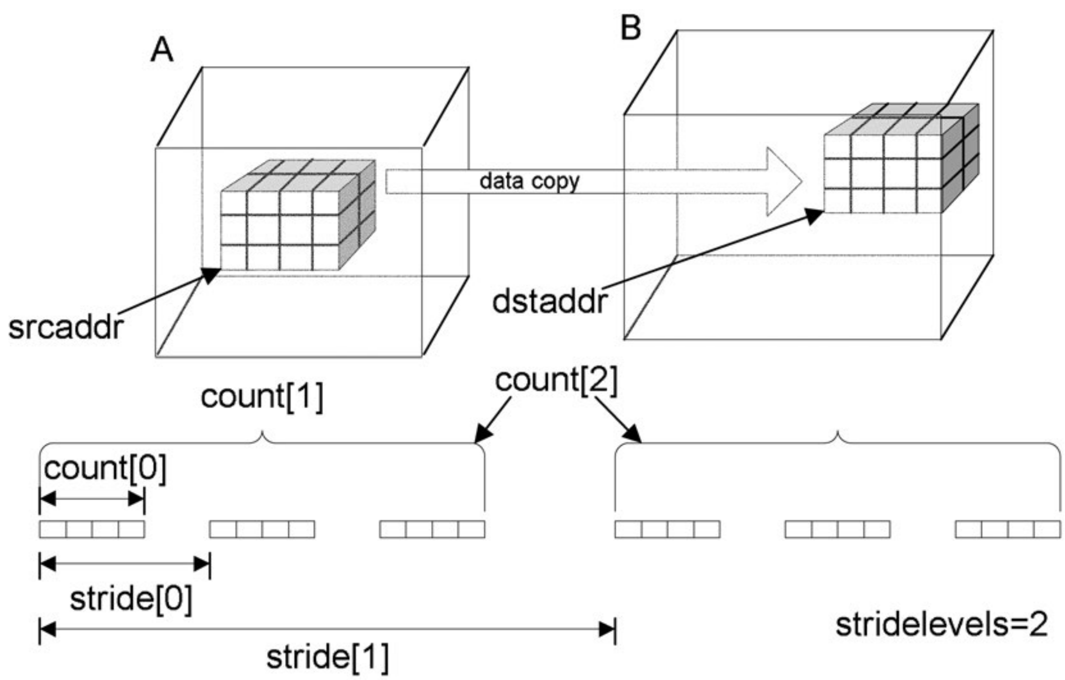


Figure 2: Illustration of a 3-d *upc_memput_strided* (Design B), and the in-memory data layout of the source or destination

6.2.1 GASNet interface for Strided Design B

```
void gasnet_puts_bulk(gasnet_node_t dstnode,
                      void *dstaddr, const size_t dststrides[],
                      void *srcaddr, const size_t srcstrides[],
                      const size_t count[], size_t stridelevels);
void gasnet_gets_bulk(void *dstaddr, const size_t dststrides[],
                      gasnet_node_t srcnode,
                      void *srcaddr, const size_t srcstrides[],
                      const size_t count[], size_t stridelevels);

gasnet_handle_t gasnet_puts_nb_bulk(gasnet_node_t dstnode,
                                    void *dstaddr, const size_t dststrides[],
                                    void *srcaddr, const size_t srcstrides[],
                                    const size_t count[], size_t stridelevels);
gasnet_handle_t gasnet_gets_nb_bulk(void *dstaddr, const size_t dststrides[],
                                    gasnet_node_t srcnode,
                                    void *srcaddr, const size_t srcstrides[],
                                    const size_t count[], size_t stridelevels);

void gasnet_puts_nbi_bulk(gasnet_node_t dstnode,
                          void *dstaddr, const size_t dststrides[],
                          void *srcaddr, const size_t srcstrides[],
                          const size_t count[], size_t stridelevels);
void gasnet_gets_nbi_bulk(void *dstaddr, const size_t dststrides[],
                          gasnet_node_t srcnode,
                          void *srcaddr, const size_t srcstrides[],
                          const size_t count[], size_t stridelevels);
```

These strided put/get operations operate exactly analogously to the contiguous *gasnet_put/get_bulk* functions - ie. unaligned access is permitted and the user cannot free or modify the source data until after sync. Additionally, the *srcstrides*, *dststrides*, and *count* metadata input arrays must remain valid and unchanged until after sync.

If the source locations overlap any destination locations, the result is undefined. If *stridelevels* is zero, the operation is a contiguous copy of *count*[0] bytes, and the *srcstrides* and *dststrides* arguments are ignored. If any entry in *count*[0..*stridelevels*] is zero, the operation is a no-op and the other arguments are ignored. The dimensional strides in *srcstrides* and *dststrides* must be monotonically increasing and must not specify overlapping locations - more specifically, $srcstrides[0] \geq count[0] \;\wedge\; \forall i \in [1..(stridelevels - 1)] \mid srcstrides[i] \geq (count[i] * srcstrides[i-1])$, and accordingly for *dststrides*.

7 Appendix: Open Issues and Possible Extensions

1. **Non-collective reblocking shared data movement**
   Consider providing non-collective memcpy mechanisms that directly support operating over a distributed array.

   Nothing in UPC currently provides a way to directly express non-collective automatic reblocking of arrays (i.e., allow a single thread to request shuffling of data to change the effective blocking factor of an array, especially for gathering to/from an indefinitely blocked array), although this seems like something we should eventually explore.

   Note that although it's not entirely elegant, one certainly can use the proposed scatter/gather functions to do the required communication in a single operation, ie:

```
/* assuming indexed memcpy design A */
shared [BLKSZ] double A[NUMELEM];
double localA[NUMELEM];
upc_pmemvec_t dst = { &localA, NUMELEM*sizeof(double) };
upc_smemvec_t myvec[THREADS];
for (int i=0; i < THREADS; i++) {
  myvec[i].addr = &A[BLKSZ*i];
  myvec[i].len = upc_affinitysize(NUMELEM*sizeof(double),
                                  BLKSZ*sizeof(double), i);
}
upc_memget_vlist(1, dst, THREADS, myvec);
```

   The code above gathers the pieces of the A array with affinity to each thread into a single, private contiguous buffer using a single operation (and orders them in the buffer by former thread affinity). If $NUMELEM > BLKSZ * THREADS$ (i.e., the blocks wrap around back to thread 0) and we want the data ordered by block number, we can use a slightly longer loop, that should still perform well for reasonably large block sizes:

```
/* assuming indexed memcpy design A */
shared [BLKSZ] double A[NUMELEM];
double localA[NUMELEM];
upc_pmemvec_t dst = { &localA, NUMELEM*sizeof(double) };
upc_smemvec_t myvec[NUMELEM/BLKSZ + 1];
shared [BLKSZ] double *p = A;
for (int i=0; i < NUMELEM/BLKSZ; i++) {
  myvec[i].addr = p;
  myvec[i].len = BLKSZ*sizeof(double);
  p += BLKSZ;
}
int leftoverelems = (&A[NUMELEM]-p);
if (leftoverelems > 0) {
  myvec[i].addr = p;
  myvec[i].len = leftoverelems*sizeof(double);
  i++;
}
upc_memget_vlist(1, dst, i, myvec);
```

   Note the same approaches also easily work under indexed memcpy design B (fixed-width regions) when $NUMELEMS\%BLKSZ == 0$ (and otherwise can be made to work with one additional separate memget of the left-over elements in the final partial block).

2. **Consider supporting strided source/destination that spans affinities**
   Currently the entire source region of a strided operation must have affinity to a single thread (and similarly for the destination region). If we ever add direct support for non-collective reblocking data movement, we might also consider extending the strided operations to work over distributed arrays (i.e., take a blocksize parameter as input). However, the strided interface is already quite high on the complexity scale, and this extension may scare off additional users. Furthermore, adding a blocksize parameter to the strided interface significantly complicates the pointer arithmetic in the implementation of the general block-distributed case, reducing performance (at least for that case) and increasing the testing/development burden of implementation.

3. **Consider allowing reshaping N-d strided transfers**
   The N-d strided interface (i.e., design B) does not allow the copied region to differ in shape at the source and destination (i.e., the rectangular section being copied must have the same extents in N-d space at either end). Note the interface *does* permit the underlying N-d arrays to differ in their dimensions, and it *does* efficiently allow transfers to/from a contiguous buffer at either end. However, it does not allow one to take the elements from an arbitrary N-d rectangular section at the source and shuffle them into an arbitrary N-d rectangular section of different shape (and equal volume) at the destination. The interface could be adapted to support this (bizarre?) usage by splitting the *count* array into *srccount* and *dstcount* (adding to the complexity of the interface and implementation) but it was perceived that there was no demand for the additional generality.

4. **Remote completion (target notification)**
   In some algorithms, one may want the ability to initiate a point-to-point non-blocking operation and allow the target thread (rather than the initiator) to synchronize on the completion of the operation. However, it's unclear how such an interface would look for UPC or even if it's consistent with UPC's general philosophy of one-sided communication through globally shared memory with logical affinity (since such a primitive is really just send/recv two-sided message passing in disguise - the only significant difference being that the initiator provides all the relevant memory addresses).

5. **Explicitly non-blocking UPC collectives and IO**
   All the collective and IO functions could be enhanced with handle-based non-blocking versions. Because these functions are collective, this should be done with a *different* collective handle type (e.g., *upc_all_handle_t*) and corresponding collective synchronizations functions (e.g., *upc_all_waitsync* / *upc_all_trysync*).

6. **Consider relaxing the required lifetime of the input metadata arrays**
   Currently the async UPC functions that take metadata input (e.g., address lists) in array form require those metadata arrays to remain unchanged until the operation has been successfully synchronized. This decision was motivated by the desire to provide the greatest freedom to implementors - this guarantee may allow an implementation to avoid copying the metadata inputs, and therefore provide better performance. The user is already required to ensure the source data remains unchanged while the operation is in progress (again, to avoid requiring synchronous copying overhead in the async initiation functions), so it doesn't seem overly burdensome to additionally require the metadata arrays to remain unchanged until the async operation has been synchronized. However, if this becomes problematic for applications in practice, then we could consider relaxing the lifetime requirement for the metadata arrays.

7. **Clean up the limit on the number of outstanding async operations**
   We basically want a limit which is guaranteed to be high enough such that application writers and code generators never have to worry about it (ie firmly disallow implementations that provide some paltry amount, like four non-blocking operations), but clearly an unbounded number of outstanding operations is not efficiently supportable (due to handle representation constraints, if nothing else).

   We may want to provide a compile-time constant defined by the implementation (e.g., *UPC_MAX_ASYNC_INFLIGHT*) that specifies a per-thread limit on how many async operations are permitted to be in-flight (unsynced) at any given time, and furthermore require all implementations to provide a value $UPC\_MAX\_ASYNC\_INFLIGHT >= 2^{16} - 1$.

8. **Provide a transpose operation**
The monotonicity restrictions on the contents of *srcstrides*/*dststrides* in the strided API (design B) effectively imply that one cannot transpose the dimensions of the copied region during a strided copy using this interface. Given that transpose is a frequently-used operation (and one that may need to be done to/from remote memory), it may be worthwhile to add a version of the strided interface which allows one to specify a transpositional strided copy. This should be a separate function for documentation reasons (it's conceptually different than copy) and because efficient implementations are likely to differ considerably from the non-transpositional case. In addition to relaxing the monotonicity property, we'd also want to add one more element to the *srcstrides*/*dststrides* array so the user can explicitly indicate the dimension with unit-stride contiguity (in the current interface, the lowest order dimension always has an implicit stride of 1, which is not true in a transpositional copy). If we choose to provide this extension, we may additionally consider allowing negative stride values, which would cause the transposition to execute a negative injection on the index space (i.e., values would be "reflected" across a dimension, effectively "flipping over" the values in the rows along the given direction).

9. **Provide wrappers for 2-d and 3-d strided copy**
Given that 2-d and 3-d arrays are so commonly used, we could provide wrapper functions around the strided copy (design B) function which take all the necessary parameters as values and construct the metadata expected by the general N-d strided copy function. There would be some overhead associated with the wrapper (especially if the metadata lifetime requirement remains unchanged), but it would provide a simpler interface for less sophisticated users.

10. **Consider adding multi-remote-node scatter-gather API to GASNet**
To provide the conduit with higher-level information about ongoing operations, possibly allowing more intelligent messaging decisions. This would need to outweigh the significant performance penalties associated with the increase in metadata size, and the additional scanning/sorting/copying of the metadata required in the implementation under this design. It also seems likely that a more general implementation approach to improving messaging decisions (i.e., an asynchronous agent or queue that facilitates cross-operation optimizations) is likely to provide better total performance and would obviate any perceived performance motivation for such an interface extension. In any case, we can easily extend the API with multi-remote-node flavors of each function in the future if empirical evidence reveals a significant net advantage.