/* The Berkeley UPC Runtime Specification
* Version: 3.8
* $Revision: 1.20 $
* Copyright 2002-4, Dan Bonachea <bonachea@cs.berkeley.edu>
*/

/* This file describes the interface between the platform-independent code
generated by a UPC-to-C translating compiler, and the hand-written UPC
runtime layer that implements the language on a given architecture

Many/most of the operations below will be implemented using macros or inline functions in an actual
implementation (a number of design decisions in the interface were based on the expected optimizations
that will occur in such an implementation).

They are specified using function declaration syntax below to make the types clear
All correct generated 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

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

Implementation-specific values in declarations are indicated using "???
Sections marked "Implementor's note" are recommendations to implementors and are
not part of the specification
*/

#include <inttypes.h>

(['/**
     * UPC Runtime Types
     * */

     /* For more information on a particular type, see the relevant section of the
     * specification below.
     *
     * upcr_thread_t               // UPC thread number
     * upcr_shared_ptr_t           // shared pointer
     * upcr_pshared_ptr_t          // phaseless shared pointer (indefinitely blocked or blocksize=1)
     * upcr_phase_t                // phase of shared pointer
     * upcr_register_value_t       // largest unsigned integer type that will fit in a CPU register
     * upcr_handle_t               // handle for nonblocking operations
     * upcr_valget_handle_t        // handle for nonblocking value get operations
     * upcr_startup_shalloc_t      // Information struct for statically allocated shared data
     * upcr_startup_pshalloc_t     // Information struct for statically allocated, phaseless shared data
     * upcr_startup_arrayinit_diminfo_t // Initialization info for each dimension of a statically
                                               // allocated shared array
     */

*/

(['/**
     * Control Interface
     * */

/* Runtime initialization functions.
* There are two sets of initialization functions for the Berkeley UPC
* runtime: one low-level set targeted at compiler developers, who want the
* largest amount of control over behavior, and one 'simpler' interface,
* targeted at application/library developers who wish to use UPC within a
* larger, non-UPC C/C++ program (though it can also be used by UPC
* compilers).  The simpler interface (bupc_init() and bupc_init_reentrant())
* uses the lower-level API, plus a set of 'magic' global variables provided
* by the UPC linker, to provide the full set of information needed, while the
* low-level API takes all needed information in the function parameters.
*/

Low-level API:
* If used, the low-level initialization functions must be called in the
* following order:
  * upcr_startup_init()
  * upcr_startup_attach()
  * upcr_startup_spawn()
  * upcr_exit()       // not always needed: see description
*/
/*
 * Bootstraps a UPC job and performs any system-specific setup required.
 *
 * Called by all applications that use the UPC runtime at startup to bootstrap
 * the job before any other processing takes place. Must be called before
 * any calls to any other functions in this specification, with the
 * command-line parameters passed to main (argc/argv), which may be modified
 * or augmented by this call (and are thus not safe to use before this call).
 * The semantics of any code executing before the call to
 * 'upcr_startup_init()' is implementation-specific (for example, it is
 * undefined whether 'stdin/stdout(stderr' are functional, or even how many
 * nodes will run that code).
 *
 * If the application using the runtime requires that it be run with a
 * fixed number of UPC threads, pass the thread count in the
 * 'static_threadcnt' parameter, and the program will abort with a error
 * message if the provided value does not match the execution environment that
 * will be provided. Pass <= 0 for applications that can run with a dynamic
 * number of UPC threads. If pthreads are used, a positive integer must be
 * supplied for 'default_pthreads_per_proc'; otherwise, pass 0.
 *
 * The 'main_name' parameter should be passed the name of the user's main()
 * UPC function: it is used to help users find that symbol name when debugging.
 * You may pass NULL if this is not needed.
 *
 * Upon return from 'upcr_startup_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 runtime functions are the only ones that
 * may be called between 'upcr_startup_init()' and 'upcr_startup_attach()':
 * 'upcr_mynode()'  
 * 'upcr_nodes()'    
 * 'gasnet_getMaxLocalSegmentSize()' 
 * 'gasnet_getMaxGlobalSegmentSize()' 
 * 'upcr getenv()'  
 * 'upcr_global_exit()'
 *
 * All other runtime calls are prohibited until after a successful
 * 'upcr_startup_attach()'.
 * 'upcr_startup_init()' may fail with a fatal error and
 * implementation-defined message if the nodes of the job cannot be
 * successfully bootstrapped.
 * This function may be called repeatedly, but only the first invocation will
 * have any effect.
 */

void upcr_startup_init(int *pargc, char ***pargv,
upcr_thread_t static_threadcnt,
upcr_thread_t default_pthreads_per_proc,
const char * main_name);

#define UPCR_ATTACH_ENV_OVERRIDE        1
#define UPCR_ATTACH_REQUIRE_SIZE        2
#define UPCR_ATTACH_SIZE_WARN           4

/*
 * Initializes the UPC runtime's network system, including shared memory
 * regions. This function must be called after upcr_startup_init(), but
 * before any of the other upcr_startup_ functions.
 *
 * The 'default_shared_size' parameter gives the default size to request for
 * each UPC thread's shared memory region.
 *
 * The 'default_shared_offset' parameter specifies the minimum distance (in
 * bytes) to provide between the current end of the regular C heap (commonly
 * provided by sbrk(0)) and the beginning of the shared memory region. On
 * some platforms this offset becomes the growth limit for the regular C heap
 * (and thus determines how much more memory malloc(), calloc(), etc. can
 * return before failing). On most systems, it is irrelevant, and 0 should be
 * passed, since using a large offset may limit the size of the shared memory
 * region.
 *
 * Values for 'default_shared_size' and 'default_shared_offset' must be
 * multiples of UPCR_PAGESIZE. Both parameters may each be overridden at run
 * time if the 'flags' parameter allows it (see below).
 *
 * The size and address of the shared region that is created for each node in
 * the application can be determined after this call with the
 * gasnet_getSegmentInfo() function. The size of the shared segment is
 */
#define UPCR_ATTACH_ENV_OVERRIDE        1
#define UPCR_ATTACH_REQUIRE_SIZE        2
#define UPCR_ATTACH_SIZE_WARN           4

/*
guaranteed to be no larger than the requested size times the number of
threads on the node (with pthreads==1 if pthreads are not being used).
The region can be smaller than the requested amount, unless
UPCR_ATTACH_REQUIRE_SIZE is passed in the 'flags' parameter or the
UPC_REQUIRE_SHARED_SIZE environment variable is set to a nonempty value.

The 'flags' parameter can contain one or more of the following values (OR
them together if multiple flags are used):

UPCR_ATTACH_ENV_OVERRIDE
- if passed, the function checks the process' environment for
  UPC_SHARED_HEAP_SIZE and/or UPC_SHARED_HEAP_OFFSET. If these are
  set to valid values (a number immediately followed by a 'MB' or
  'GB', for example '32MB' for 32 megabytes, or '4GB' for 4
  gigabytes), they override the default_shared_size and
  default_shared_offset values, respectively.

UPCR_ATTACH_REQUIRE_SIZE
- if this flag is passed, the function will die with an error message
  printed to stderr if the allocated shared region on any node is
  smaller than the amount that was asked for times the number of
  pthreads. Can be overridden at startup by setting the
  UPC_REQUIRE_SHARED_SIZE environment variable to 'yes' or 'no'.

UPCR_ATTACH_SIZE_WARN
- if this flag is passed, the runtime will issue a warning to stderr
  if a smaller shared memory segment than requested will be used.
  Can be overridden at startup by setting the UPC_SIZE_WARN
  environment variable to 'yes' or 'no'.

If any errors are encountered during upcr_startup_attach, an error message
is printed and the job is aborted.

/*

void upcr_startup_attach(uintptr_t default_shared_size,
    uintptr_t default_shared_offset,
    int flags);
*/

/*
* Struct argument to upcr_startup_spawn.
*/

struct upcr_startup_spawnfuncs {
    void (*pre_spawn_init)();
    void (*per_pthread_init)();
    void (*cache_init)(void *start, uintptr_t len);
    void (*heap_init)(void * start, uintptr_t len);
    void (*static_init)(void *start, uintptr_t len);
    int  (*main_function)(int argc, char **argv);
};
/*
* Completes runtime initialization, including launching of any additional
* pthreads (if a pthreaded runtime is used), and running of the user's main()
* function (if any).
* */

'*pargc' and '*pargv' will be passed to the 'main_function' in the
'spawnfuncs' argument (if it is non-NULL). The 'static_data_size'
parameter should have a non-zero value if and only if static shared data get
their own section of the shared memory segment, separate from the shared
heaps (in Berkeley upc static data are allocated off of the heap; GCCUPC
uses a separate segment), and this should be the size of the static data
for each UPC thread. The 'default_cache_size' indicates how much shared
memory to reserve for caching (by default): since caching is not yet
implemented, pass 0.

The 'spawnfuncs' parameter is a struct containing pointers to six
functions.
The 'pre_spawn_init' function, if not NULL, is called first, before any
 pthreads are launched. It can contain any arbitrary initializations that
 should happen only once per-process.

Each of the remaining function pointers is called once on each UPC thread.
The 'per_pthread_init' function, if not NULL, is called by each pthread,
and can contain arbitrary initializations that need to happen on a
per-pthread basis.
The 'cache_init' function is called next, but only if caching is being
used (i.e. if UPCR_USING_CACHING is defined). It may be set to NULL
otherwise. It must initialize the cache within the given region.
The 'heap_init' function is called next, and must initialize the runtime shared heaps. It is passed parameters indicating the starting address and length of the region to use for the heap.

The 'static_init' function is then called. It must set up all static data for the UPC thread, and is passed the address and length of the segment to be used. The length passed is guaranteed to be at least as large as provided in the 'static_data_size' parameter, and the locations in the region are guaranteed to have lower virtual addresses than the local addresses for any shared data with affinity to this thread allocated using the dynamic shared memory allocation functions.

Next, a barrier is performed. Finally, if the 'main_function' parameter is NULL, the function returns (and upcr_exit() should be used for any program exit path, including the end of 'main'). Otherwise 'main_function' is called with the command line arguments passed in 'argv' and 'argc' (with a new copy made for each pthread if pthreads are used). Again, upcr_exit() should be used for any exit paths, except that returns from 'main_function' are handled automatically, with the return value used as the program's exit code.

If 'main_function' != NULL, this function never returns.

If any errors occur during this function, an error message is printed to stderr and the job is terminated.

void upcr_startup_spawn(int *pargc, char ***pargv,
uintptr_t static_data_size,
uintptr_t default_cache_size,
struct upcr_startup_spawnfuncs *spawnfuncs);

/* Runtime shutdown/exit function. */

This function should be called as the last program statement for all exit paths from a UPC application, with the single exception that the 'main_function' used by upcr_startup_spawn() may simply return an integer, in which case the behavior is the same as if a call to this function had been made with that value.

The behavior of any code called after this function is undefined (i.e. it may not execute).

void upcr_exit(int exitcode);

*******************************************************************************

Framework for external bootstrapping of the UPC runtime.

The 'bupc_init()' and 'bupc_init_reentrant()' functions allow 'external' bootstrapping of the UPC runtime, i.e., initialization of the runtime by programs which are not written entirely in UPC, and whose main() does not appear in a UPC file.

To provide the full amount of needed data to the runtime, these functions require a set of 'magic' global variables to be set by the Berkeley UPC linker (upcc).

*******************************************************************************/

/* Public, user-accessible function for bootstrapping the Berkeley UPC runtime from a non-UPC C or C++ program that does not use pthreads. */

A call to this function should be the first statement in main(). The semantics of any code appearing before it is implementation-defined (for example, it is undefined how many threads of control will run that code, or whether stdin/stdout/stderr are functional). The presence of environment variables is also not guaranteed, but after this call returns bupc_getenv() can be used to retrieve them (regular getenv() is not guaranteed to provide them).

The addresses of the command-line parameters must be passed, and it is not safe to otherwise refer to them until after this function returns, as it may supplement or modify them.

Once bupc_init() has returned, the application may safely call into UPC routines. All exit paths from the program should call bupc_exit() as their last program statement.

If any errors are encountered during this function’s execution, an error message is printed to stderr and the job will be aborted.
bupc_init(int *argc, char ***argv);

bupc_init_reentrant(int *argc, char ***argv, int (*pmain_func)(int, char **));

bupc_getenv(const char *env_name);
/* Runtime shutdown/exit routine.
 *
* This function should be called as the last program statement by any program
* that uses bupc_init() to bootstrap the UPC runtime. It does not need to be
* used when bupc_init_reentrant() is used. The 'exitcode' provided will be
* returned to the console that invoked the job, assuming all of the threads
* terminate with this function, and use the same exitcode. If different
* threads of the program exit with different values, one of the values will
* be chosen arbitrarily. The behavior of any program statements after a call
* to bupc_exit() is undefined.
 *
* If used within a hybrid MPI/UPC program, bupc_exit() ensures that
* MPI_Finalize() is called, if needed. MPI_Finalize should NOT be called
* by user code if this function is used.
*/

void bupc_exit(int exitcode);

/* "Magic" variables that must appear in the linked executable to support use
* of the bupc_init() and/or bupc_init_reentrant() functions.
 *
* Definitions of all variables with the 'UPCRL_' prefix must be provided by
* client code. NULL/zero values can be used if system does not support
* creating executables that call UPC functions from within a non-UPC C or C++
* program.
*/

#define UPCR_USING_LINKADDRS

/* Set to 0 if dynamic threads used, else to the static UPC thread count */
extern upcr_thread_t    UPCRL_static_thread_count;

/* Default size of shared memory segment and offset */
extern uintptr_t UPCRL_default_shared_size;
extern uintptr_t UPCRL_default_shared_offset;

/* Support for systems which store shared variables in a separate linker
* section.
 *
* Some systems (ex: GCC UPC) convert ‘shared’ static data into a separate
* linker section. In this case, the values stored in shared pointers are
* within that linker section (since they are assigned by the linker).
 *
* To work with Berkeley UPC, the linker section must be mapped into a portion
* of the shared region provided by gasnet. Also, if pthreads are used, a
* separate copy of the linker section must exist for each pthread.
 *
* These requirements are handled by the runtime so long as
* UPCR_USING_LINKADDRS is defined, and the beginning/ending addresses of the
* linker section are provided in 'UPCRL_shared_begin' and 'UPCRL_shared_end'.
* The runtime uses these addresses to make a copy for each pthread of the
* linker section. Then, during each shared <=> local address conversion, an
* offset is used to convert between the linker-assigned address for a given
* shared pointer and its the address within a pthread’s copy of the static
* data region.
 *
* On ELF-based systems, the beginning and ending addresses are typically
* provided by arranging for the UPCRL_shared_begin/end to be the first and
* last variables in the linker section that the linker sees (most linker
* this can be achieved by putting the symbols in separate 'first.o' and
* 'last.o' object files that are then passed to the linker as the first and
* last objects on the linker command line).
*/
#endif

/* Nonzero to launch an extra progress thread. Primarily useful for parallel
* debugger support. Redundant on some networks, and likely to slow down
* performance on the others */
extern int UPCRL_progress_thread;

/* Default size of runtime cache, if used. */
extern uintptr_t UPCRL_default_cache_size;

/* default flags to pass to upcr_attach, if upcr_startup_init() is used to
* bootstrap the runtime */
extern int UPCRL_attach_flags;

/* default pthreads per process: pass 0 if not using pthreads */
extern upcr_thread_t UPCRL_default_pthreads_per_node;
/* Name used to rename user’s main() function */
extern const char * UPCRL_main_name;

/* Hook for arbitrary per-process initializations */
extern void (*UPCRL_pre_spawn_init)();

/* Hook for arbitrary per-pthread initializations */
extern void (*UPCRL_per_pthread_init)();

 /* Cache initialization function to pass to upcr_startup_attach() */
extern void (*UPCRL_cache_init)(void *start, uintptr_t len);

/* Heap initialization function to pass to upcr_startup_attach(): */
extern void (*UPCRL_heap_init)(void *start, uintptr_t len);

/* Static data initialization function to pass to upcr_startup_attach() */
extern void (*UPCRL_static_init)(void *start, uintptr_t len);

/* Function to ensure MPI has been initialized. Use only if both MPI and a *
gasnet conduit are being used, else set to NULL. This function must not */
call MPI_Init is it has already been called by gasnet (use *
MPI_Initialized() to check). No other code in the application should call *
MPI_Init(), else behavior is undefined. */
extern void (*UPCRL_mpi_init)(int *pargc, char ***pargv);

/* Function to ensure MPI is shut down at program completion. Use only if both *
MPI and a gasnet conduit are being used, else set to NULL. MPI_Finalize *
should only be called if the UPCRL_mpi_init function called *
MPI_Inititalize(). No other code in the application should call *
MPI_Finalize(), else behavior is undefined. */
extern void (*UPCRL_mpi_finalize)();

/* terminate the current job with a given exit code - non-collective operation */
 /* this function may be called by any thread at any time after initialization and will cause the */
system to flush all I/O, release all resources and terminate the job for all active threads */
 /* this function is called automatically by the runtime system in the event of any */
fatal error or catchable terminate-the-program signals (e.g. segmentation fault) */
 /* this function must be called at the end of main() after a barrier to ensure proper system exit */
 /* the console which initiated the current job will receive the provided exitcode */
 /* as a program return value in a system-specific way */
 /* if more than one thread calls upcr_global_exit() within a given synchronization phase */
 /* with different exitcode values, the value returned to the console will be one of the */
 /* provided exit codes (chosen arbitrarily) */
/* Implementation notes: */
gasnet may send a fatal signal to indicate a remote node exited or crashed *
calls gasnet_exit to terminate the job on remote nodes */
void upcr_global_exit(int exitcode);

/* UPCR_BEGIN_FUNCTION() - this declaration must appear at the very beginning of every function */
/* (before any declarations) in generated code that intends to call any of the entry points */
/* provided by this API. It provides the runtime system with a place for minimal per-function */
/* initialization that may be necessary on some platforms, particularly when pthreads are used */
#define UPCR_BEGIN_FUNCTION() ???

/* UPCR_EXIT_FUNCTION() - this declaration must appear immediately before every */
/* exit point from any function that intends to call any of the entry points */
/* provided by this API. It provides the runtime system with information for */
/* hooking events (temporary object cleanup, profiling) that need to occur at function */
/* exit. */
#define UPCR_EXIT_FUNCTION() ???

/* UPC thread number: this is an unsigned integral type used to represent the */
0-to-((N-1) thread numbers of UPC threads within an application. The size of this type may */
/* vary depending on the shared pointer representation used. */
typedef ??? upcr_thread_t;

/* Job Layout Queries - Interrogate thread information */
upcr_thread_t upcr_mythread(); /* returns a 0-based UPC thread index */
upcr_thread_t upcr_threads(); /* returns the number of UPC threads in the system */
upcr_thread_t upcr_mynode(); /* returns a 0-based GASNet node index */
upcr_thread_t upcr_nodes(); /* returns the number of GASNet nodes in the system */

/* Implementors note:
all code should be written such that UPCR_MAX_THREADS can simply be changed (up to 2^31-1) and
the system recompiled to increase the thread limit
all code should be written such that UPCR_MAX_BLOCKSIZE can simply be changed
(along with a possible change to the type used to represent phase in upcr_shared_t)
and the system recompiled to increase the block size limit
*/

/* UPCR_PLATFORM_ENVIRONMENT provides the platform-independent UPC compiler with
* some clues about the memory layout of the current platform to aid optimization
* trade-offs.
* The possible configuration values are:
* UPCR_PURE_SHARED - purely shared memory, remote memory accesses are handled entirely
* by hardware with no software interpretation overhead
* UPCR_PURE_DISTRIBUTED - purely distributed memory, remote memory accesses are handled by some
* software networking layer
* UPCR_SHARED_DISTRIBUTED - a mixture of the above - some remote memory accesses are handled
* by hardware, others by a software networking layer
* UPCR_OTHER - any configuration not captured by the above options
*/
#define UPCR_PLATFORM_ENVIRONMENT ???

/* size of memory page on operating system, in bytes */
#define UPCR_PAGESIZE ???

/* Shared Pointer Representation
* Implementation Notes:
The contents of these typedefs is NOT part of the specification and will
vary with implementation. Therefore, the fields shown should NOT be accessed
by the generated code or compiler
typedef ??? upcr_shared_ptr_t;
typedef ??? upcr_pshared_ptr_t;

/* Shared pointer phase: represents the phase of a shared pointer, i.e., the index of the current
element in the current block of shared memory. This is an unsigned integral type, whose size
may vary depending on the shared pointer implementation.
*/
typedef ??? upcr_phase_t;

typedef ??? upcr_shared_ptr_t;
typedef ??? upcr_pshared_ptr_t;

} upcr_shared_ptr_t

typedef struct {  
    uintptr_t _localaddr;  // make this the first field to speed pointer use  
    short _threadid;      // use shorts so the entire struct fits in 2 words  
} upcr_pshared_ptr_t

Shared Pointer Manipulation

/* Convert a shared ptr with affinity to the current thread  
   into a local pointer.  
   If sptr does not have affinity to the calling thread the  
   result is implementation-specific */
void *upcr_shared_to_local(upcr_shared_ptr_t sptr);
void *upcr_pshared_to_local(upcr_pshared_ptr_t sptr);

/* Convert a local ptr into the current thread's shared memory space into  
   a shared pointer appropriate for use in remote operations from other threads.  
   The phase field is set to zero. Some implementations may issue an error if lptr  
   does not point into the shared region for the current thread.  
   Note this operation is not accessible from the UPC source level, but may be useful  
   for generated code nonetheless (e.g. to support a debugger)  
   The _ref versions modify a shared pointer in place rather than returning a  
   shared pointer value, which may be more efficient in some implementations */
upcr_shared_ptr_t upcr_local_to_shared(void *lptr);
void upcr_local_to_shared_ref(void *lptr, upcr_shared_ptr_t *result);

upcr_pshared_ptr_t upcr_local_to_pshared(void *lptr);
void upcr_local_to_pshared_ref(void *lptr, upcr_pshared_ptr_t *result);

/* Same as above, but sets the phase and thread to a particular value.  
   phase is expressed in number of elements */
upcr_shared_ptr_t upcr_local_to_shared_withphase(void *lptr, upcr_phase_t phase, upcr_thread_t threadid);
void upcr_local_to_shared_ref_withphase(void *lptr, upcr_phase_t phase, upcr_thread_t threadid,  
                                         upcr_shared_ptr_t *result);

/* Convert back and forth between shared and pshared representations  
   upcr_pshared_to_shared sets phase to zero  
   The _ref versions modify a shared pointer in place rather than returning a  
   shared pointer value, which may be more efficient in some implementations */
upcr_pshared_ptr_t upcr_shared_to_pshared(upcr_shared_ptr_t sptr);
void upcr_shared_to_pshared_ref(upcr_shared_ptr_t sptr, upcr_pshared_ptr_t *result);

upcr_pshared_ptr_t upcr_pshared_to_shared(upcr_pshared_ptr_t sptr);
void upcr_pshared_to_shared_ref(upcr_pshared_ptr_t sptr, upcr_shared_ptr_t *result);

/* Same as above, but sets the phase to a particular value.  
   phase is expressed in number of elements */
upcr_pshared_ptr_t upcr_pshared_to_shared_withphase(upcr_pshared_ptr_t sptr, upcr_phase_t phase);
void upcr_pshared_to_shared_ref_withphase(upcr_pshared_ptr_t sptr, upcr_phase_t phase,  
                                           upcr_shared_ptr_t *result);

/* reset the phase field of a given shared pointer to zero  
   (used for casting between block sizes) */
upcr_shared_ptr_t upcr_shared_resetphase(upcr_shared_ptr_t sptr);
void upcr_shared_resetphase_ref(upcr_shared_ptr_t *sptr);

/* Returns the thread number that has affinity to the given shared pointer,  
   or 0 for a NULL shared pointer. If sptr is not a valid shared pointer,  
   the results are undefined. */
upcr_thread_t upcr_threadof_shared(upcr_shared_ptr_t sptr);
upcr_thread_t upcr_threadof_pshared(upcr_pshared_ptr_t sptr);

/* Returns the phase field of the given shared pointer,  
   Returns 0 for a NULL shared pointer or any pshared pointer */
phase is expressed in number of elements
*/

upcr_phase_t upcr_phaseof_shared(upcr_shared_ptr_t sptr);
upcr_phase_t upcr_phaseof_pshared(upcr_pshared_ptr_t sptr); /* always returns zero */

/* Returns an implementation-defined value reflecting the local address
of the object pointed to. This may or may not be the actual virtual address
where the object is stored - use upcr_to_local() when casting shared pointers
to local pointers.
*/
uintptr_t upcr_addrfield_shared(upcr_shared_ptr_t sptr);
uintptr_t upcr_addrfield_pshared(upcr_pshared_ptr_t sptr);

/* upcr_affinitysize calculates the exact size of the local portion of the data
in a shared object with affinity to a given thread, specified by threadid.
totalsize should be the total number of bytes in the shared object.
nbytes is the block size in BYTES.
*/
size_t upcr_affinitysize(size_t totalsize, size_t nbytes, upcr_thread_t threadid);

/* return non-zero iff the given shared pointer is a null reference */
int upcr_isnull_shared(upcr_shared_ptr_t sptr);
int upcr_isnull_pshared(upcr_pshared_ptr_t sptr);

/* Does not point to a valid shared memory address on some thread */
int upcr_isvalid_shared(upcr_shared_ptr_t *p);
int upcr_isvalid_pshared(upcr_pshared_ptr_t *p);

/* Set a shared ptr to NULL. */
int upcr_setnull_shared(upcr_shared_ptr_t *p);
int upcr_setnull_pshared(upcr_pshared_ptr_t *p);

/* Shared pointer increments/decrements -
add a positive or negative displacement to a shared pointer.
Both the inc and blockelems arguments should be expressed in number of elements
elemsz is the target element size in bytes.
The "add" versions return an updated shared pointer,
the "inc" versions modify the input shared pointer in place.
Points with a definite static blocksize > 1 should use the "shared" version,
shared pointers with indef blocksize use the "psharedI" version
shared pointers with blocksize == 1 use the "pshared1" version
*/
upcr_shared_ptr_t upcr_add_shared(upcr_shared_ptr_t sptr, size_t elemsz, ptrdiff_t inc, size_t blockelems);
void              upcr_inc_shared(upcr_shared_ptr_t *psptr, size_t elemsz, ptrdiff_t inc, size_t blockelems);

upcr_pshared_ptr_t upcr_add_psharedI(upcr_pshared_ptr_t sptr, size_t elemsz, ptrdiff_t inc);
void               upcr_inc_psharedI(upcr_pshared_ptr_t *psptr, size_t elemsz, ptrdiff_t inc);

upcr_pshared_ptr_t upcr_add_pshared1(upcr_pshared_ptr_t sptr, size_t elemsz, ptrdiff_t inc);
void               upcr_inc_pshared1(upcr_pshared_ptr_t *psptr, size_t elemsz, ptrdiff_t inc);

/* return non-zero iff ptr1 and ptr2 are both null,
 or if they currently reference the same memory location */
int upcr_isequal_shared_shared(upcr_shared_ptr_t ptr1, upcr_shared_ptr_t ptr2);
int upcr_isequal_shared_pshared(upcr_shared_ptr_t ptr1, upcr_pshared_ptr_t ptr2);
int upcr_isequal_pshared_pshared(upcr_pshared_ptr_t ptr1, upcr_pshared_ptr_t ptr2);
int upcr_isequal_pshared_local(upcr_pshared_ptr_t ptr1, void *ptr2);
int upcr_isequal_pshared_local(upcr_pshared_ptr_t ptr1, void *ptr2);

/* Shared pointer / Shared pointer comparison and subtraction -
Compare shared pointers sptr1 and sptr2 and calculate sptr1 - sptr2.
blockelems is the block size for both ptrs, expressed in num elements
(UPC type compatibility semantics require both pointers have the same blocksize)
elemsz is the target element size in bytes.
Points with a definite static blocksize > 1 should use the "shared" version,
shared pointers with indef blocksize use the "psharedI" version
shared pointers with blocksize == 1 use the "pshared1" version
There are three possible cases:
returns 0 if sptr1 and sptr2 currently reference the same memory cell (i.e. upcr_isequal() would return true)
returns a positive or negative value N (an element count) to indicate that
upcr_add_shared(sptr2, elemsz, N, blockelems2) would yield a shared pointer that is upcr_isequal() to sptr1
(if N > 0, we say that sptr1 is "greater than" sptr2, and if N < 0 we say that sptr1 is "less than" sptr2)
otherwise, fatal error if there is no value which can be added to sptr1 to make it equal sptr2
(e.g. sptr1 and sptr2 are indef blocksize pointers with different affinities)
ptrdiff_t upcr_sub_shared (upcr_shared_ptr_t sptr1, upcr_shared_ptr_t sptr2, size_t elemsz, size_t blockelems);
pptrdiff_t upcr_sub_psharedI(upcr_pshared_ptr_t sptr1, upcr_pshared_ptr_t sptr2, size_t elemsz);
pptrdiff_t upcr_sub_psharedI1(upcr_pshared_ptr_t sptr1, upcr_pshared_ptr_t sptr2, size_t elemsz);

/* Affinity checks - return non-zero iff the given shared pointer currently
has affinity to the calling thread (or indicated thread, respectively)*/
int upcr_hasMyAffinity_shared (upcr_shared_ptr_t sptr);
int upcr_hasMyAffinity_pshared(upcr_pshared_ptr_t sptr);
int upcr_hasAffinity_shared (upcr_shared_ptr_t sptr, upcr_thread_t threadid);
int upcr_hasAffinity_pshared(upcr_pshared_ptr_t sptr, upcr_thread_t threadid);

/* ------------------------------------------------------------------------------------ */

/ * Shared Memory Accesses
 */

Transfer scalar values to/from shared memory which may or may not be remote

These comments apply to all put/get functions:

- Only functions suffixed with '_strict' can be used to implement a strict operation; all other data movement functions in this specification are implicitly relaxed.
- nbytes should be a compile-time constant whenever possible
- nbytes must be >= 0 and has no maximum size, but implementations will likely optimize for small powers of 2
- source and target addresses (both local and shared) are assumed to be properly aligned for accessing objects of size nbytes
- if nbytes extends beyond the current block the results are undefined
- destoffset(srcoffset) is an optional positive or negative BYTE offset, which is added to the address indicated by dest(src) to determine the target(source) address for the put(get) operation (Useful for puts(gets) with shared structures)
- if adding the number of bytes indicated by destoffset(srcoffset) to dest(src) would cause dest(src) to pass the end of the current block, the result is undefined
- If the source and target memory areas overlap (for memory-to-memory transfers) but do not exactly coincide, the resulting target memory contents are undefined
- Implementations are likely to optimize for the important special case of zero destoffset(srcoffset)

/*
/ * UPCR_ATOMIC_MEMSIZE() is a macro describing the datatype sizes at which memory accesses
will be done atomically. Given a datatype width sz (in bytes) it will return non-zero at compile time
iff a local or shared memory access of exactly sz bytes, to an address aligned by sz bytes, will happen
atomically with respect to accesses from other threads to the same location.
A non-zero return value for a given size does not guarantee atomicity for smaller sizes
or unaligned accesses of the given size
Some architectures may provide no atomic sizes
UPCR_ATOMIC_MEMSIZE(0) will return the largest atomic size available, or zero if none exists
*/
#define UPCR_ATOMIC_MEMSIZE(sz) ???

/* --- Blocking memory-to-memory puts and gets ---
A call to these functions will block 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 '_strict' versions implement strict UPC puts(gets). It is an error for any
nonblocking (relaxed or strict) operation to overlap a strict put/get.
*/
void upcr_put_shared (upcr_shared_ptr_t dest, ptrdiff_t destoffset, const void *src, size_t nbytes);
void upcr_put_pshared(upcr_pshared_ptr_t dest, ptrdiff_t destoffset, const void *src, size_t nbytes);
void upcr_put_shared_strict (upcr_shared_ptr_t dest, ptrdiff_t destoffset, const void *src, size_t nbytes);
void upcr_put_pshared_strict(upcr_pshared_ptr_t dest, ptrdiff_t destoffset, const void *src, size_t nbytes);

void upcr_get_shared (void *dest, upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
void upcr_get_pshared(void *dest, upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
void upcr_get_shared_strict (void *dest, upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
void upcr_get_pshared_strict(void *dest, upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);

/* --- Non-blocking operations ---
The following functions provide non-blocking, split-phase memory access to shared data.
All such non-blocking operations require an initiation (put or get) and a subsequent
synchronization on the completion of that operation before the result is guaranteed.
Synchronization of a get operation means the local result is ready to be examined, and
will contain a value held by the shared 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)
Synchronization of a put operation means the source data has been written to the shared location and get operations issued subsequently by any thread will receive the new value or a subsequently written value (assuming no other threads are writing the location).

There are two categories of non-blocking operations:

- "explicit handle" (nb) - return a specific handle to caller which is used for synchronization this handle can be used to synchronize a specific subset of the nb operations in-flight
- "implicit handle" (nbi) - don't return a handle - synchronization is accomplished by calling a synchronization routine that synchronizes all outstanding nbi operations

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).

The compiler bears full responsibility for maintaining the memory consistency semantics presented to the UPC user when using non-blocking operations - the compiler must generate synchronizations at the appropriate points (e.g. before calling upcr_unlock() or upcr_notify())

Implementors should attempt to make the non-blocking 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 an unlimited number of non-blocking operations in-progress - that is, the client is free to issue an unlimited number of non-blocking operations before issuing a sync operation, and implementations must handle this correctly without deadlock.

The '_strict' versions of these functions implement strict nonblocking UPC puts/gets.

It is an error for any nonblocking (relaxed or strict) operation to overlap a strict put/get. Only one strict nonblocking operation may be pending at any time, and no other operation (relaxed or strict) may be initiated or completed in between that strict operation's initiation and its completion.

/*

typedef ??? upcr_handle_t;
#define UPCR_INVALID_HANDLE ???

/* --- Non-blocking memory-to-memory, explicit handle (nb) ---

These calls initiate a non-blocking operation and return "immediately" with a non-blocking handle that can be used to later synchronize the operation, using one of the explicit sync operations

Once the put version returns, the source memory may safely be overwritten for the get version, if the contents of the source memory change while the operation is in progress the result will be implementation-specific.

The contents of the destination memory address are undefined until a synchronization completes successfully for the non-blocking operation.

The operations may return UPCR_INVALID_HANDLE to indicate it was possible to complete the operation immediately without blocking (e.g. operations on shared memory with affinity to this thread).

It is an error to discard the upcr_handle_t value for an operation in-flight - i.e. to initiate an operation and never synchronize on its completion

upcr_handle_t upcr_put_nb_shared (upcr_shared_ptr_t dest, ptrdiff_t destoffset, const void *src, size_t nbytes);
upcr_handle_t upcr_get_nb_shared (void *dest, upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_handle_t upcr_put_nb_pshared(upcr_pshared_ptr_t dest, ptrdiff_t destoffset, const void *src, size_t nbytes);
upcr_handle_t upcr_get_nb_pshared(void *dest, upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);

*/
upcr_handle_t upcr_put_nb_shared_strict (upcr_shared_ptr_t dest, ptrdiff_t destoffset, 
    const void  *src, size_t nbytes);
upcr_handle_t upcr_get_nb_shared_strict (void *dest, upcr_shared_ptr_t src, ptrdiff_t srcoffset, 
    size_t nbytes);
upcr_handle_t upcr_put_nb_pshared_strict (upcr_pshared_ptr_t dest, ptrdiff_t destoffset, 
    const void  *src, size_t nbytes);
upcr_handle_t upcr_get_nb_pshared_strict (void  *dest, upcr_pshared_ptr_t src, 
    ptrdiff_t srcoffset, size_t nbytes);

/* --- Explicit handle synchronization (for get_nb and put_nb) ---
upcr supports two basic variants of synchronization for non-blocking operations -
   trying (polling) and waiting (blocking).
All explicit synchronization functions take one or more upcr_handle_t values
   as input and either return an indication of whether the operation has completed
   or block until it completes.
*/

/* Single operation explicit synchronization
Synchronize on the completion of a single, particular non-blocking operation
   that was initiated by this thread.
upcr_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 upcr_wait_syncnb() returns and
   may not be passed to future synchronization operations
upcr_try_syncnb() always returns immediately, with the value 1 if the
   operation is complete (at which point the handle value is "dead", and may
not be used in future synchronization operations), or 0 if the operation is
   not yet complete and future synchronization is necessary to complete this operation.
upcr_(try,wait)_syncnb_strict() operate just as upcr_(try,wait)_syncnb() do,
   but must be used for strict operations (and only for strict operations).
It is legal to pass UPCR_INVALID_HANDLE as input to these functions:
   upcr_wait_syncnb(_, strict)(UPCR_INVALID_HANDLE) return immediately and
upcr_try_syncnb(_, strict)(UPCR_INVALID_HANDLE) return 1.
It is an error to pass a upcr_handle_t value for an operation which has already
   been successfully synchronized using one of the explicit synchronization functions
   and doing so has undefined results*/
void upcr_wait_syncnb(upcr_handle_t handle)
int  upcr_try_syncnb(upcr_handle_t handle)
void upcr_wait_syncnb_strict(upcr_handle_t handle)
int  upcr_try_syncnb_strict(upcr_handle_t handle)

/* Multiple operation explicit synchronization
Synchronize on the completion of an array of non-blocking operation handles
   (all of which were initiated by this thread).
numhandles specifies the number of handles in the provided array of handles.
   requires numhandles >= 0
upcr_wait_syncnb_all() blocks until all the specified operations have completed
   (or returns immediately if they have all already completed).
upcr_try_syncnb_all() always returns immediately, with the value 1 if all
   the specified operations have completed, or 0 if one or more of the operations
is not yet complete and future synchronization is necessary to complete some
   of the operations.
upcr_try_syncnb_all() will modify the provided array to reflect completions -
   handles whose operations have completed are overwritten with the value UPCR_INVALID_HANDLE,
and the client may test against this value when upcr_try_syncnb_all() returns 0
   to determine which operations are complete and which are still pending.
implementations of upcr_wait_syncnb_all() _may_ modify the provided array to reflect completions,
   but this is not required (and not necessarily for the client since it always blocks until all
operations in the list are complete)
It is legal to pass the value UPCR_INVALID_HANDLE in some or all of the array entries,
   and both functions will ignore them so those values have no effect on behavior.
If all entries in the array are UPCR_INVALID_HANDLE (or numhandles==0), then
   upcr_try_syncnb_all() will return 1.
Note that there are no strict variants, since the UPC memory consistency
   model prohibits multiple outstanding strict operations.
*/
void upcr_wait_syncnb_all(upcr_handle_t *, size_t numhandles)
int  upcr_try_syncnb_all(upcr_handle_t *, size_t numhandles)

/* These operate analogously to the 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
   UPCR_INVALID_HANDLE).*/
Specifically, upcr_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 UPCR_INVALID_HANDLE.
Similarly, upcr_try_syncnb_some will check if at least one valid handle in the
list has completed (setting all completed handles to UPCR_INVALID_HANDLE) and
return 1 if it detected at least one completion or 0 otherwise (except as below)

Both functions ignore UPCR_INVALID_HANDLE values. If the input list is empty or
consists only of UPCR_INVALID_HANDLE values, upcr_wait_syncnb_some will
return immediately and upcr_try_sync_some will return 1.

void upcr_wait_syncnb_some(upcr_handle_t *handles, size_t numhandles)
int upcr_try_syncnb_some(upcr_handle_t *handles, size_t numhandles)

/* --- Non-blocking memory-to-memory, implicit handle (nbi) ---
These calls initiate a non-blocking operation and return "immediately"
the operation must later be completed using a call to one of the implicit sync functions
Once the put version returns, the source memory may safely be overwritten
For a get operation, if the contents of the source memory change while the operation is in progress
the result will be implementation-specific.
The contents of the destination memory address are undefined until a
synchronization completes successfully for the non-blocking operation.
There are no strict nbi operations, as the UPC memory consistency model prohibits multiple
outstanding strict operations.
*/

void upcr_put_nbi_shared (upcr_shared_ptr_t dest, ptrdiff_t destoffset,
    const void *src, size_t nbytes);
void upcr_get_nbi_shared (void *dest, upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);

void upcr_put_nbi_pshared(upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    const void *src, size_t nbytes);
void upcr_get_nbi_pshared(void *dest, upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);

/* --- Implicit handle synchronization (for get_nbi and put_nbi) ---*/
/* Synchronize on an implicit list of outstanding non-blocking operations.
These functions implicitly specify a set of non-blocking operations on which to synchronize -
either all outstanding implicit-handle gets initiated by this thread,
all outstanding implicit-handle puts initiated by this thread,
or all outstanding implicit-handle operations (both puts and gets) initiated by this thread
(where outstanding is defined as all those operations which have been initiated
but not yet completed through a successful implicit-handle synchronization).
The wait variants block until all operations in this implicit set have completed
The try variants test whether all operations in the implicit set have completed,
and return 1 if so (or if there are no outstanding implicit-handle operations)
or 0 otherwise
Implicit synchronization functions will synchronize operations initiated within
other function frames by this thread
As with the initiation functions, there are no strict variants here.
*/

void upcr_wait_syncnbi_gets();
void upcr_wait_syncnbi_puts();
void upcr_wait_syncnbi_all();
int upcr_try_syncnbi_gets();
int upcr_try_syncnbi_puts();
int upcr_try_syncnbi_all();

/* --- Implicit 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. upcr_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.
*/

/* upcr_begin_nbi_accessregion() and upcr_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 region.
All implicit-handle non-blocking operations initiated inside the region become "associated"
with the abstract region handle being constructed. upcr_end_nbi_accessregion() returns an
explicit handle which collectively represents all the associated implicit-handle operations
(those initiated within the 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 initiated in the 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 (e.g. upcr_wait_syncnbi_all())
Explicit-handle operations initiated within the region operate as usual and do _not_ become
associated with the region.
*/
void upcr_begin_nbi_accessregion();
upcr_handle_t upcr_end_nbi_accessregion();
/* sample code:
upcr_begin_nbi_accessregion(); // begin the region
upcr_put_nbi_shared(...); // becomes associated with this region
while (...) {
    upcr_put_nbi_shared(...); // becomes associated with this region
}
h2 = upcr_get_nb_shared(...); // unrelated explicit-handle operation not associated with region
upcr_wait_syncnb(h2);
handle = upcr_end_nbi_accessregion(); // end the region and get the handle
.... // other code, which may include unrelated implicit-handle operations+syncs, or other regions, etc
upcr_wait_syncnb(handle); // wait for all the operations associated with the region to complete
*/
/* --- Register-memory operations --- */
/* upcr_register_value_t represents the largest unsigned integer type that can fit entirely
in a single CPU register for the current architecture and ABI.
SIZEOF_UPCR_REGISTER_T is a preprocess-time literal integer constant (i.e. not "sizeof()")
indicating the size of this type in bytes
*/
typedef unsigned ??? upcr_register_value_t;
#define SIZEOF_UPCR_REGISTER_VALUE_T ???
/* the value forms of put - these take the value to be put as input parameter to avoid
forcing outgoing values to local memory in generated code.
Otherwise, the behavior is identical to the memory-to-memory versions of put above
requires: nbytes > 0 && nbytes <= SIZEOF_UPCR_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 nbyte 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
The semantics of the _strict versions are the same as for the regular, non-value put/get functions
*/
void upcr_put_shared_val (upcr_shared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
void upcr_put_shared_val_strict (upcr_shared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
upcr_handle_t upcr_put_nb_shared_val (upcr_shared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
upcr_handle_t upcr_put_nb_shared_val_strict (upcr_shared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
void upcr_put_nbi_shared_val(upcr_shared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
void upcr_put_pshared_val (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
void upcr_put_pshared_val_strict (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
upcr_handle_t upcr_put_nb_pshared_val (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
upcr_handle_t upcr_put_nb_pshared_val_strict (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
void upcr_put_nbi_pshared_val(upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
/* blocking value get - these return the fetched value to avoid
forcing incoming values to local memory in generated code.
Otherwise, the behavior is identical to the memory-to-memory blocking get
requires: nbytes > 0 && nbytes <= SIZEOF_UPCR_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 nbyte integral value on the current architecture
and setting the high-order bits (if any) to zero (i.e. no sign-extension)
The semantics of the _strict versions are the same as for the regular,
non-value put/get functions
*/
void upcr_put_pshared_val (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
void upcr_put_pshared_val_strict (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
upcr_handle_t upcr_put_nb_pshared_val (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
upcr_handle_t upcr_put_nb_pshared_val_strict (upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
void upcr_put_nbi_pshared_val(upcr_pshared_ptr_t dest, ptrdiff_t destoffset,
    upcr_register_value_t value, size_t nbytes);
/*
 upcr_register_value_t upcr_get_shared_val (upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_register_value_t upcr_get_shared_val_strict (upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_register_value_t upcr_get_pshared_val(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_register_value_t upcr_get_pshared_val_strict(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_register_value_t upcr_wait_syncnb_valget(upcr_valget_handle_t handle);
 upcr_valget_handle_t upcr_get_nb_pshared_val_strict(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_valget_handle_t upcr_get_nb_pshared_val(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_valget_handle_t upcr_get_nb_shared_val_strict(upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_valget_handle_t upcr_get_nb_shared_val(upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
 upcr_register_value_t upcr_wait_syncnb_valget(upcr_valget_handle_t handle);
*/

typedef ??? upcr_valget_handle_t;

upcrvalget_handle_t upcr_get_nb_shared_val(upcrshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcrvalget_handle_t upcr_get_nb_pshared_val(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcrvalget_handle_t upcr_get_nb_pshared_val_strict(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcrvalget_handle_t upcr_get_nb_shared_val_strict(upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcrvalget_handle_t upcr_get_nb_shared_val(upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);

upcr_valget_handle_t upcr_get_pshared_val_strict(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_valget_handle_t upcr_get_pshared_val(upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_valget_handle_t upcr_get_shared_val_strict (upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_valget_handle_t upcr_get_shared_val (upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_get_pshared_val (upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_get_shared_val (upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_pshared_val_strict (upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_pshared_val (upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);

/* Blocking value puts/gets for floating-point quantities (float, double)
 these operate similarly to the blocking value puts/get for integral types, except are specialized
 for the float and double types on the current platform
 the source/target address is assumed to be correctly aligned for accessing the given FP type
 the primary motivation is to permit puts/gets directly between local shared memory locations
 and the floating point registers, without forcing the use of an integer register
 or stack temporary as an intermediary (which would be otherwise necessary without these functions)
 there are no non-blocking variants for these functions because they are meant primarily for
 optimizing low-latency local memory accesses
*/
void upcr_put_shared_floatval (upcr_shared_ptr_t dest, ptrdiff_t destoffset, float value);
void upcr_put_shared_doubleval (upcr_shared_ptr_t dest, ptrdiff_t destoffset, double value);
float upcr_get_shared_floatval (upcr_shared_ptr_t src, ptrdiff_t srcoffset);
double upcr_get_shared_doubleval (upcr_shared_ptr_t src, ptrdiff_t srcoffset);

void upcr_put_pshared_floatval (upcr_pshared_ptr_t dest, ptrdiff_t destoffset, float value);
void upcr_put_pshared_doubleval (upcr_pshared_ptr_t dest, ptrdiff_t destoffset, double value);
float upcr_get_pshared_floatval (upcr_pshared_ptr_t src, ptrdiff_t srcoffset);
double upcr_get_pshared_doubleval (upcr_pshared_ptr_t src, ptrdiff_t srcoffset);

upcr_register_value_t upcr_get_pshared_val_strict (upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_get_pshared_val (upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_get_shared_val_strict (upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_get_shared_val (upcr_shared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_pshared_val_strict (upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);
upcr_register_value_t upcr_pshared_val (upcr_pshared_ptr_t src, ptrdiff_t srcoffset, size_t nbytes);

/*
 * Shared Memory Bulk Memory Operations
 *---------------------------------------------
 Transfer bulk data to/from shared memory which may be remote
 Note these operations all take upcr_shared_ptr_t's (not phaseless ptrs)
 All sizes are specified in BYTES, nbytes >= 0
 Semantics are the same as those specified in the UPC spec
 Implementations will likely optimize for larger values of nbytes
 If the source and target memory areas overlap (but do not exactly coincide),
 the resulting target memory contents are undefined

The motivation for having memget and memput, separately from the memory ops above:
 - well defined semantics for crossing block boundaries
 - no alignment constraints on the pointers
 - non-blocking memput constrains source memory from changing while operation is
   in progress to avoid a potential buffering copy
 - optimize for large sizes

Implementor's notes:
 upcr_memset() can be implemented on GASNet using a single small active message, which makes
 it very efficient in terms of network communication
*/

void upcr_memget(void *dst, upcr_shared_ptr_t src, size_t nbytes);

*/
void upcr_memput(upcr_shared_ptr_t dst, const void *src, size_t nbytes);
void upcr_memcpy(upcr_shared_ptr_t dst, upcr_shared_ptr_t src, size_t nbytes);
void upcr_memset(upcr_shared_ptr_t dst, int c, size_t nbytes);

/* non-blocking versions of the bulk memory operations
   must be synchronized using explicit or implicit synchronization as with
   non-blocking scalar memory access operations
   The contents of the memory referenced by src must NOT change between
   initiation and successful synchronization, or the result is undefined
   upcr_nbi_memset is synchronized as if it were an implicit-handle put operation
*/

upcr_handle_t upcr_nb_memget(void *dst, upcr_shared_ptr_t src, size_t nbytes);
upcr_handle_t upcr_nb_memput(upcr_shared_ptr_t dst, const void *src, size_t nbytes);
upcr_handle_t upcr_nb_memcpy(upcr_shared_ptr_t dst, upcr_shared_ptr_t src, size_t nbytes);
upcr_handle_t upcr_nb_memset(upcr_shared_ptr_t dst, int c, size_t nbytes);

/* Dynamic Memory Allocation:
   -------------------------------------------
   UPC runtime interface to generated code for memory allocation
   -------------------------------------------
*/

/* Non-collective operation that allocates nblocks * blocksz bytes in the shared memory area
   with affinity to this thread, and returns a pointer to the new data,
   which is suitably aligned for any kind of variable.
   Requires nblocks >= 0 and blocksz >= 0
   The memory is not cleared or initialized in any way, although it has been properly
   registered with the network system in a way appropriate for the current platform such
   that remote threads can read and write to the memory using upcr shared data transfer operations.
   If insufficient memory is available, the function will print an implementation-defined
   error message and terminate the job.
*/

upcr_shared_ptr_t upcr_local_alloc(size_t nblocks, size_t blocksz);

/* Non-collective operation that allocates nblocks * blocksz bytes spread across the shared memory area
   of 1 or more threads, and returns a pointer to the new data,
   which is suitably aligned for any kind of variable.
   Requires nblocks >= 0 and blocksz >= 0
   The memory is blocked across all the threads as if it had been created by the UPC declaration:
   shared [blocksz] char[nblocks * blocksz] (i.e. both sizes are expressed in bytes).
   Specifically, thread i allocates (at least):
   \[ \text{Max}(0 \cup \{0 < n <= \text{nblocks} \times \text{blocksz} \mid \text{floor}(\text{n}-1/\text{blocksz}) \% \text{THREADS} == i\}) \text{ bytes.} \]
   More specifically, thread i allocates (at least) this many bytes:
   \[
   \begin{align*}
   \text{blocksz} \times \text{ceil}(\text{nblocks}/\text{THREADS}) & \text{ if } i <= (\text{nblocks} \% \text{THREADS}) \\
   \text{blocksz} \times \text{floor}(\text{nblocks}/\text{THREADS}) & \text{ if } i > (\text{nblocks} \% \text{THREADS})
   \end{align*}
   \]
   Implementor’s note: Some implementations may allocate the full \(\text{blocksz} \times \text{ceil}(\text{nblocks}/\text{THREADS})\)
   memory on each thread for simplicity, even though less may be required on some threads.
   Note if nblocks == 1, then all the memory will be allocated in the shared memory space
   of thread 0 (and implementations should attempt not to waste space on other threads in this
   common special case).
   In all cases the returned pointer will point to a memory location in the shared memory space
   of thread 0, and any subsequent chunks in the shared space of other threads will be logically
   aligned with this pointer (such that incrementing a shared pointer of the appropriate blocksz
   past the end of a block on one thread will bring it to the start of the next block on the next thread).
   The phase of the returned pointer is set to zero
   The memory is not cleared or initialized in any way, although it has been properly registered
   with the network system in a way appropriate for the current platform such that remote threads
   can read and write to the memory using the upcr shared data transfer operations.
   If insufficient memory is available, the function will print an implementation-defined error
   message and terminate the job.
*/
upcr_shared_ptr_t upcr_global_alloc(size_t nblocks, size_t blocksz);

/* Collective version of upcr_global_alloc() - the semantics are identical to upcr_global_alloc()
 with the following exceptions:
 * the function must be called by all threads during the same synchronization phase,
 * and all threads must provide the same arguments
 * may act as a barrier for all threads, but might not in some implementations
 * all threads receive a copy of the result, and the shared pointer values
 will compare equal (according to upcr_isequal_shared_shared()) on all threads
 */

upcr_shared_ptr_t upcr_all_alloc(size_t nblocks, size_t blocksz);

/* Non-collective operation used to deallocate a shared memory region previously allocated
 (but not deallocated) using one of: upcr_local_alloc(), upcr_global_alloc() or upcr_all_alloc().
 If sptr is a null pointer the operation is ignored.
The shared pointer value passed to upcr_free() must be the same value returned by the
 allocation function that created the region (i.e. it must point to the beginning of the object,
 and for upcr_global_alloc() and upcr_all_alloc() the thread field must indicate thread 0).
If sptr has been freed by a previous call to upcr_free() or does not point to the beginning of
 a live object in shared memory, the behavior is undefined.

Note that any thread may call upcr_free() to free a given dynamically-allocated shared object,
even if that object was created by a call to upcr_local_alloc() from a different thread.
Also note that memory allocated using upcr_all_alloc() should only be freed by a call to upcr_free()
from a _single_ thread.
*/

void upcr_free(upcr_shared_ptr_t sptr)

/* ------------------------------------------------------------------------------------ */

/* Execute the notification for a split-phase barrier, with a barrier value */
This is a non-blocking operation that completes immediately after noting the barrier value.
No synchronization is performed on outstanding memory accesses (i.e. the
compiler is responsible for inserting the appropriate syncs to implement
the null strict reference implied by upcr_notify before calling upcr_notify())
Generates a fatal error if this is the second call to upcr_notify() on this thread
since the last call to upcr_wait() or the beginning of the program
flags should be 0 to indicate a normal barrier (which carries the value barrierval)
or UPCR_BARRIERFLAG_ANONYMOUS to indicate an "anonymous" barrier, where the
barrierval argument is ignored and the notify automatically "matches" with any
anonymous or non-anonymous value provided by the notify called on other threads
Implementation notes:
check value of thread's notify/wait toggle which records current state of synchronization
save this thread's barrier value and flags
increment a counter of local threads that called notify this epoch & return
last thread on this node to call upcr_notify()
checks the barrier values calls gasnet_notify() with appropriate flags then resets the counter
*/

void upcr_notify(int barrierval, int flags);

/* Execute the wait for a split-phase barrier, with a barrier value */
This is a blocking operation that returns only after all threads have called upcr_notify()?
No synchronization is performed on outstanding memory accesses (i.e. the
compiler is responsible for inserting the appropriate syncs to implement
the null strict reference implied by upc_wait after calling upc_wait())
Generates a fatal error if there were no preceding calls to upcr_notify() from this thread,
or if this is the second call to upc_wait() since the last call to upcr_notify() on this thread
Generates a fatal error if flags is not equal to the flags value passed in the preceding
upc_notify() call made by this thread
Generates a fatal error if flags==0 and the supplied barrierval doesn’t match the value provided
in the preceding upc_notify() call made by this thread
Generates a fatal error if any two threads passed non-anonymous barrier values which didn’t match
during the notify calls which began this barrier phase
Implementation notes:
check and toggle value of thread’s notify/wait status which records current state of synchronization
check if i matches previous value provided by thread in this barrier epoch
first thread to enter grabs a lock, spin waits until all threads have called notify (counter reset)
calls gasnet_wait with appropriate flags, (aborts if there is a mismatch reported)
and signals that wait is complete by writing a barrier_done flag
all other threads either block on the lock (if they arrive during gasnet_wait) or merely
see that wait is complete and return the mismatch value
increment a counter of local threads that called wait this epoch & block (sleep or spin-wait)
last thread on this node to call upcr_wait calls gasnet_wait(i), then releases the other threads
when it returns
*/
void upcr_wait(int barrierval, int flags);

/* upcr_try_wait() functions similarly to upcr_wait(), except that it always returns immediately.
If the barrier has been notified by all threads, the call behaves as a call to upcr_wait()
with the same barrierval and flags, and returns the value 1
If the barrier has not yet been notified by some thread,
the call is a no-op and returns the value 0
Note this call is not mandated by the UPC spec, but may be useful for performing purely local
computation in optimized code or performing system housekeeping duties
*/
int upcr_try_wait(int barrierval, int flags);

/* Network polling
* ===============
* The upcr_poll() function explicitly causes the runtime to attempt to make
* progress on any network requests that may be pending. While many other
* runtime functions implicitly do this as well (i.e. most of those which call
* the network layer) this function may be useful in cases where a large amount
* of time has elapsed since the last runtime call (e.g. if a great deal of
* application-level calculation is taking place). This function may also be
* indirectly when a upc_fence is used.
*/
void upcr_poll();

/* UPC locks
* =========
The following assumes the updates in the UPC spec 1.1 regarding upc locks,
namely:
- upc_lock_t is an opaque shared datatype with incomplete type (prohibits
  statically-allocated upc_lock_t objects)
- upc_lock_init() is no longer necessary or useful and is removed
- upc_lock_free() is added to allow users to free dynamically-allocated locks
- UPC locks are _not_ recursive (a thread must not attempt to re-acquire a lock it already owns)
similar to upc_lock_t, the runtime lock datatype is totally opaque and
always manipulated through upcr_shared_ptr_t pointers, which must NEVER be
dereferenced by generated code
this spec intentionally doesn’t even provide a name or size for the lock datatype
the shared pointer returned by the lock allocation routines has reference semantics,
(i.e. copying the pointer yields a reference to the same lock)
but otherwise need not even be a real pointer. In other words, the thread affinity
and addrfield components of these shared pointers is completely undefined,
so casting them to a local pointer on _any_ thread may yield a pointer value which
doesn’t point to a valid memory address (or points to a random object)
this allows implementations which (for example) store an integer lock identifier
in the address field rather than a true pointer
*/

/* non-collective operation (intended to be called by a single thread)
which dynamically allocates and initializes a lock,
and returns a upcr_shared_ptr_t which references that lock.
If insufficient resources are available, the function will print an implementation-defined
error message and terminate the job.
*/
upcr_shared_ptr_t upcr_global_lock_alloc();

/* collective operation which dynamically allocates and initializes a lock,
and returns a upcr_shared_ptr_t which references that lock.
* the function must be called by all threads during the same synchronization phase,
* may act as a barrier for all threads, but might not in some implementations
* all threads receive a copy of the result, and the shared pointer values will
* compare equal (according to upcr_isequal_shared_shared()) on all threads
If insufficient resources are available, the function will print an implementation-defined
error message and terminate the job.
*/
upcr_shared_ptr_t upcr_all_lock_alloc();

/* block until the referenced lock can be acquired by this thread
if no other thread is currently holding or contending for the referenced lock,
this operation must return within a bounded amount of time
implementations should attempt to provide fairness in the presence of
contention for this lock, but this property is not required
if lockptr does not reference a valid lock object (i.e. one previously allocated by 
upcr_global_lock_alloc() or upcr_all_lock_alloc() and not deallocated using 
upcr_lock_free()) then the results are undefined
if the current thread is already holding the referenced lock, the result is undefined
(although implementations are recommended to print a useful error message and abort) 
*/

void upcr_lock(upcr_shared_ptr_t lockptr);

/* attempt to acquire the referenced lock without blocking
the operation always returns immediately, with the value 1 if the lock was 
successfully acquired, or with the value 0 if the lock could not be acquired at this time 
if no other thread is currently holding or contending for the referenced lock, 
repeated calls to this operation will eventually succeed within a bounded amount of time 
if lockptr does not reference a valid lock object then the results are undefined 
if the current thread is already holding the referenced lock, the result is undefined 
(although implementations are encouraged to print a useful error message and abort) 
*/

int upcr_lock_attempt(upcr_shared_ptr_t lockptr);

/* unlock the referenced lock 
this operation releases the referenced lock, which must have been previously locked 
by this thread using upcr_lock(), or a successful call to upcr_lock_attempt() 
(otherwise the results are undefined)
if lockptr does not reference a valid lock object then the results are undefined 
this operation always completes within a bounded amount of time 
implementations are encouraged to detect violations to the locking semantics 
(e.g. unlock with no matching lock) but this is not required 
*/

void upcr_unlock(upcr_shared_ptr_t lockptr);

/* free a lock - non-collective operation 
this call (always made from a single thread) releases any system resources 
associated with the referenced lock and makes the lock object "invalid" for all threads 
the lock need not have been explicitly created by the current thread (i.e. it may have 
been created by a call to upcr_global_lock_alloc() on a separate thread and passed to this one) 
any subsequent calls from any thread using this invalidated lock object have undefined effects 
if lockptr does not reference a valid lock object then the results are undefined 
this operation always completes within a bounded amount of time 
repeated calls to upcr_lock_free(upcr_global_lock_alloc()) must succeed indefinitely 
(i.e. it must actually reclaim any associated resources)
the call will succeed immediately regardless of whether the referenced lock is currently 
unlocked or currently locked (by any thread) 
*/

void upcr_lock_free(upcr_shared_ptr_t lockptr);

/* ------------------------------------------------------------------------------------ */
/* Statically-allocated user variables 
* =================================== 
* The following interfaces provide portable support for statically-allocated user variables 
* (shared and unshared, scalar and array) 
*/

/* Thread-Local Data (TLD) 
* ----------------------- 
* Thread-local data (TLD) is defined to be any NON-shared, statically-allocated 
* (i.e. not automatic lifetime) objects declared in UPC source files, namely non-shared 
* file-scope (global) objects or static local variables (block-scope TLD). 
* TLD must be declared and accessed specially by generated code to ensure correct operation 
* across the variety of platforms implementing the UPC runtime. 
* 
* The macros below must be used to declare all TLD - global or static user unshared 
* variables (unless they are declared with 'extern', or are located in a 
* regular C file (such as a header file with a name ending in '.h'), since if 
* pthreads are used, these variables will need to be made thread-specific. 
* Static variables need to be transform into global variables before this 
* macro can be used (and their names should be mangled to avoid name 
* collisions). 
* 
* Since uses of these macros are intended to be filterable by tools like 
* grep, they must be used at the start of a new line, and their contents 
* cannot contain line breaks. 
*/

/* UPCR_TLD_DEFINE(name, initval, size) must be used when declaring 
unshared global/static variables that the user has initialized. The 
macro takes the variable name of the value, and the size (in bytes, as a 
single literal number--"sizeof", expressions like '3 + 4', etc., are not 
allowed). So the UPC compiler should transform 

  int foo = 5; 
*/
* on a platform with 4 byte integers into
* int
* UPCR_TLD_DEFINE(foo, 4) = 5;

* Unshared pointers to shared types (i.e. thread local variables
with type upcr_shared_ptr_t or upcr_pshared_ptr_t) should be initialized
with UPCR_INITIALIZED(P)SHARED rather than the value the user specified.
* The full type of the variable must preceding the macro, and so arrays and
function pointers must use a typedef. For instance,

  int natural_nums[3] = { 1, 2, 3};
  void (*int_taker)(int) = &print_int;

* Would become

  typedef int _type_natural_nums[3];
  _type_natural_nums
  UPCR_TLD_DEFINE(natural_nums, 12) = { 1, 2, 3 };
  typedef void (*_type_int_taker)(int);
  _type_int_taker
  UPCR_TLD_DEFINE(int_taker, 4) = &print_int;

* For variables that are not explicitly initialized by the user,
* UPCR_TLD_DEFINE_TENTATIVE(name, size) must be used. The macro
* works the same way as UPCR_TLD_DEFINE, except that it should not be
* followed by "= initializer_expr."
* For more information on the uses of these macros, and the treatment of
thread-local data generally, see the web page on "static user data" in the
Runtime documentation on the Berkeley UPC web site.
*/

#define UPCR_TLD_DEFINE(name, size)
#define UPCR_TLD_DEFINE_TENTATIVE(name, size)

/* UPCR_TLD_ADDR: retrieve the address of the current thread’s representative
of the TLD variable with the given name (name must be a simple identifier)
address is returned as a (void *) and should be cast to the proper type before use
*/
#define UPCR_TLD_ADDR(name) ???

/* Example usage:
int x = *(int*)UPCR_TLD_ADDR(foo);
*(int*)UPCR_TLD_ADDR(foo) = 100;
{{int*UPCR_TLD_ADDR(natural)[2] = 27;

Implementors note:
UPCR_TLD_ADDR() returns an address rather than an l-value because some planned
implementations of TLD may not have the TLD type information available
(TLD will just be opaque bytes in a special data segment)
*/

/* Statically-allocated Shared Data (SSD)

Statically-allocated Shared Data (SSD) is defined to be any shared,
statically-allocated (i.e. not automatic lifetime) objects declared in UPC
source files, namely any shared file-scope (global) objects or static local
variables (block-scope SSD).

All SSD is allocated and initialized dynamically at runtime, instead of being
truly statically allocated (since on most platforms network-addressable memory
can not be assigned at compile time, and must be dynamically allocated). The
basic idea is the compiler replaces each SSD declaration with a
upcr_shared_ptr or upcr_pshared_ptr that will point to the relevant data item
at runtime (all SSD access operations must be modified appropriately to
traverse the extra level of indirection). The compiler also adds an
allocation and an initialization function for each UPC file it compiles, in
which all SSD declared in the file is allocated and initialized (some
thread-local data initializations may also be performed there).

The functions listed below should only be used in these per-file startup
allocation/initialization routines. For more information on the naming
conventions for these functions, the content that should go in them, and the
framework that calls them, refer to the "Handling Static Data in the UPC
Runtime" document (available in the documentation section of the Berkeley UPC
These values are guaranteed to be defined by every shared pointer representation. `UPCR_INITIALIZED_{P}SHARED` should be used by the compiler to initialize all `upcr_shared_ptr_t` and `upcr_pshared_ptr_t`'s that represent shared variables the user defines with an initial value (if the user does not provide a value, do not provide any value for the `upcr_{p}shared_ptr`, either). `UPCR_NULL_{P}SHARED` should be used to initialize `upcr_{p}shared_ptr`'s that represent unshared pointers to shared data that the user explicitly initialized to NULL.

Note these values are only guaranteed to work as variable initializer expressions, and may not safely be used as the rhs for a general assignment statement. `(upcr_setnull_{p}shared` must be used for such applications

```c
#define UPCR_INITIALIZED_SHARED { ??? }
#define UPCR_NULL_SHARED { ??? }
#define UPCR_INITIALIZED_PSHARED { ??? }
#define UPCR_NULL_PSHARED { ??? }
```

Shared pointer variables that contain NULL values.

Note that these can resolve to either a basic type or a struct (depending on the shared pointer representation), so code that uses them must work in either case (e.g., it would be illegal to use them in a context requiring a scalar value, such as passing it to `==` operator.)

```c
const upcr_shared_ptr_t upcr_null_shared;
const upcr_pshared_ptr_t upcr_null_pshared;
```

This function will be provided by each shared pointer representation, and returns nonzero if the passed pointer is initialized to `UPCR_INITIALIZED_{P}SHARED`.

```c
int upcr_is_init_shared(upcr_shared_ptr_t p);
int upcr_is_init_pshared(upcr_pshared_ptr_t p);
```

Allocation information struct for shared arrays that will be striped across the UPC threads (with blocking size != 1 element):

```c
typedef struct {
  upcr_shared_ptr_t *sptr_addr;
  size_t blockbytes;
  size_t numblocks;
  int    mult_by_threads;
  size_t elemsz;
  const char *namestr;
  const char *typestr;
} upcr_startup_shalloc_t;
```

Allocation information struct for indefinitely blocked (or blocksize == 1 element) shared arrays.

```c
typedef struct {
  upcr_pshared_ptr_t *psptr_addr;
  size_t blockbytes;
  size_t numblocks;
  int    mult_by_threads;
  size_t elemsz;
  const char *namestr;
  const char *typestr;
} upcr_startup_shalloc_t;
```
int mult_by_threads;
size_t elemsz;
const char *namestr;
const char *typestr;
}
upcr_startup_pshalloc_t;

/*
 * Allocates the specified amount of memory for each shared pointer in the
 * array of info structs.
 *
 * Only performs a given allocation if the memory has not already been allocated
 * for the pointer. If the pointer was not initialized (i.e., is equal to 0
 * instead of UPCR_INITIALIZED_SHARED), any memory allocated is also memset
 * to 0.
 *
 * This function must be called by all threads collectively (like
 * upc_all_alloc, etc.). The function does not guarantee that all threads
 * will have received the data when any particular thread
 * returns from the call (i.e. it does not guarantee a barrier is performed
 * after initialization). The function does guarantee that it may be called
 * repeatedly without the need for client barrier calls to be placed in
 * between the calls.
 *
 * See the upcr_startup_shalloc_t struct definition for options affecting how
 * memory is allocated.
 *
 */
void upcr_startup_shalloc(upcr_startup_shalloc_t *infos, size_t count);

/*
 * Allocates the specified amount of memory for each phaseless shared pointer
 * in the array of info structs.
 *
 * Only performs a given allocation if the memory has not already been allocated
 * for the pointer. If the pointer was not initialized (i.e., is equal to 0
 * instead of UPCR_INITIALIZED_PSHARED), any memory allocated is also memset
 * to 0.
 *
 * This function must be called by all threads collectively (like
 * upc_all_alloc, etc.). When the function returns, the shared pointers
 * pointed to by ‘infos’ will be initialized to the correct shared memory
 * location on all UPC threads.
 *
 * See the upcr_startup_shalloc_t struct definition for options affecting how
 * memory is allocated.
 *
 */
void upcr_startup_pshalloc(upcr_startup_pshalloc_t *infos, size_t count);

/*
 * Information for a single dimension of a shared array initialization.
 *
 *      local_elems         // Number of elements in local init array’s dimension
 *      shared elems        // Number of elements in shared array’s dimension
 *      mult_by_threads     // Nonzero if shared array’s dimension should be
 *                                multiplied by THREADS
 *
 * Note that the UPC language specification mandates that for a dynamic
 * translation environment (i.e. one in which THREADS is not a compile-time
 * constant) only one dimension of a shared array can contain THREADS, and it
 * can only be used once in that dimension, to multiply a constant size.
 *
 */
typedef struct upcr_startup_arrayinit_diminfo {
    size_t local_elems;
    size_t shared_elems;
    int    mult_by_threads;
    }
upcr_startup_arrayinit_diminfo_t;

/*
 * Initializes a shared array from a local array, or to 0s if NULL is passed
 * for the local array.
 *
 * This function is used to copy initial values from a local array (generated
 * by the UPC compiler) that contains any initial values provided by the user.
 * The local array does not need to have the same size as the shared array
 * (indeed, if the shared array contains THREADS in one of its dimensions, its
 * size is not even knowable at compile time). It does, however, need to have
 * the same number of dimensions as the shared array, and the same element
 * size. All values in the shared array that do not have corresponding values
 * in the local array are memset to 0.
 */
The function takes the addresses of the shared and local arrays, a pointer to an array of structures (each of which describes a single dimension of the array), a count of the number of dimensions in the array, the size (in bytes) of the array’s element type, and the blocking factor of the array (as a number of elements).

If NULL is passed for the local array address, all local array parameters will be ignored, and the function will simply set all elements of the shared array to 0.

Here is an example:

```c
// in UPC program

  {
    { 1, 2 },
    { 3, 4 },
    { 5, 6 },
    { 3, 2, 3, 4, 5 } // the user may specify extra elems if THREADS
    // is part of the dimension
  }
};
```

Here the user has only provided a small subset of the initial values in the array (even disregarding the THREADS in the final dimension). The UPC compiler should place the initial values into a [1][4][5] array, and then set up and call the initialization function:

```c
// output .c file, at file scope

upcr_shared_ptr_t j = UPCR_INITIALIZED_SHARED;

int j_initarray[1][4][5] = {
  {
    { 1, 2 },
    { 3, 4 },
    { 5, 6 },
    { 3, 2, 3, 4, 5 }
  }
};

upcr_startup_arrayinit_diminfo_t j_diminfos[] = {
  { 1, 3, 0 },
  { 4, 4, 0 },
  { 5, 2, 1 }
};
```

// In initialization function

```c
upcr_startup_initarray(&j, j_initarray, j_diminfos, 3, sizeof(int), 5);
```

This function must be called collectively by each UPC thread for each array, in the same order and with the same arguments. The function does not guarantee that all threads will have completed their initializations when any particular thread returns from the call (i.e. it does not guarantee a barrier is performed after initialization).

Implementation notes:
----------------------

For efficiency, each thread should only copy elements that belong to its portion of the shared array, so the function should not cause any network traffic.

To save space, the local array’s dimensions should only be as large as needed to contain all the initial values specified by the user.

```c
void upcr_startup_initarray(upcr_shared_ptr_t dst, void * src,
upcr_startup_arrayinit_diminfo_t *diminfos,
size_t dimcnt, size_t elembytes, size_t blockelems);
```

This function initializes a phaseless array from a local array, or to 0s if NULL passed.

This function is identical to upcr_startup_initarray, except that it takes a phaseless shared ptr.

For phaseless shared arrays with indefinite blocksize, pass ‘0’ for the ‘blockelems’ parameter.
Implementor’s note: It should be possible to simply write this as an
inline function that calls upcr_startup_initarray(),
with upcr_pshared_to_shared() used to convert dst to
the correct type.

```c
void upcr_startup_initparray(upcr_pshared_ptr_t dst, void * src,
                               upcr_startup_arrayinit_diminfo_t *diminfos,
                               size_t dimcnt, size_t elembytes, size_t blockelems);
```

A string representing all the relevant upcr 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.

```c
#define UPCR_CONFIG_STRING "???"
```

Integer values which correspond to the major and minor version numbers
for this specification, indicating the highest version level of
runtime spec conformance which is implemented in a given runtime
(available in specification version 3.8 and higher)

```c
#define UPCR_RUNTIME_SPEC_MAJOR x
#define UPCR_RUNTIME_SPEC_MINOR x
```

 *-------------------------------------------------------------------------- *