UPC Optional Library Specifications
Version 1.3 (change-annotated)

A publication of the UPC Consortium

November 16, 2013

Change-Annotation Note:
Change annotations appearing in this document are relative to the baseline Version 1.3 Draft 1, which is believed to be semantically identical in every detail to UPC language specification version 1.2 (ratified May 2005). Change annotations in the spec body are for reviewer convenience only and are not normative. The officially ratified and normative version of this document is available at `http://upc-lang.org`.

To learn more details about each change performed during the UPC specification revision process, please visit: `http://code.google.com/p/upc-specification/`

Contents

7 Library

1 This section provides UPC parallel extensions of [ISO/IEC00 Sec 7.1.2]. Also see the UPC Required Library Specifications.

2 The libraries specified in this document are optional – conforming implementations of the UPC language may provide or omit each library at the subsection level (e.g. Sec 7.4). Any subsection which is provided must be implemented in its entirety and predefine the specified feature macro.

7.6 UPC Atomic Memory Operations `<upc_atomic.h>`

1 This subsection provides the UPC parallel extensions of [ISO/IEC00 Sec 7.19]. All the characteristics of library functions described in [ISO/IEC00 Sec 7.1.4] apply to these as well. Implementations that support this interface shall predefine the feature macro `__UPC_ATOMIC__` to the value 1.

7.6.1 Standard headers

1 The standard header is

`<upc_atomic.h>`

2 Unless otherwise noted, all of the functions, types and macros specified in Section 7.6 are declared by the header `<upc_atomic.h>`.

3 Every inclusion of `<upc_atomic.h>` has the effect of including `<upc_types.h>`.

7.6.2 Common Requirements

1 The following requirements apply to all of the functions defined in Section 7.6.

2 The UPC Atomic Memory Operations library introduces an *atomicity domain*, an object that specifies a single operand type and a set of operations over which access to a memory location in a given synchronization phase is guaranteed to be atomic if and only if no other mechanisms or atomicity domains are used to access the same memory location in the same synchronization phase. [1]

3 The following table presents the required support for operations and operand types

Operand Type	Accessors	Bit-wise Ops	Numeric Ops
Integer	X	X	X
Floating Point	X		X
`UPC_PTS`	X		

[1] In particular, this implies that atomicity is only guaranteed if atomic operations accessing a given memory location are separated from any other accesses to that location (via direct read/writes or a different domain) by a `upc_barrier` or `upc_notify`/`upc_wait`.

where

- Supported integer types are UPC_INT, UPC_UINT, UPC_LONG, UPC_ULONG, UPC_INT32, UPC_UINT32, UPC_INT64, and UPC_UINT64.
- Supported floating-point types are UPC_FLOAT and UPC_DOUBLE.
- Supported accessors are UPC_GET, UPC_SET, and UPC_CSWAP.
- Supported bit-wise operations are UPC_AND, UPC_OR, and UPC_XOR.
- Supported numeric operations are UPC_ADD, UPC_SUB, UPC_MULT, UPC_INC, UPC_DEC, UPC_MAX, and UPC_MIN.

4 The value macros listed below are defined in `<upc_atomic.h>`. All other `UPC_*` value macros used in this subsection are defined by `<upc_types.h>` (see UPC Language Specification, Section 7.3.1 and UPC Language Specification, Section 7.3.2).

Macro name	Specified operation
`UPC_GET`	`Read`
`UPC_SET`	`Write or swap`
`UPC_CSWAP`	`Conditional swap`
`UPC_SUB`	`Subtraction`
`UPC_INC`	`Increment by 1`
`UPC_DEC`	`Decrement by 1`

7.6.3 Atomic Library Types

7.6.3.1 The `upc_atomicdomain_t` type

1 The header `<upc_atomic.h>` declares the type

```
upc_atomicdomain_t
```

2 The type `upc_atomicdomain_t` is an opaque UPC type. `upc_atomicdomain_t` is a shared datatype with incomplete type (as defined in [ISO/IEC00 Sec 6.2.5]). Objects of type `upc_atomicdomain_t` may therefore only be manipulated through pointers.

3 Two pointers that reference the same atomicity domain object will compare as equal. The results of applying `upc_phaseof()`, `upc_threadof()`, and `upc_addrfield()` to such pointers are undefined.

7.6.3.2 The `upc_atomichint_t` type

1 The header `<upc_atomic.h>` declares the integral type

```
upc_atomichint_t
```

2 The following macros expand to positive integer constant expressions with type `upc_atomichint_t` and distinct values. They allow the specification of a “hint” to the library implementation to indicate a *preferred* mode of optimization for atomic operations performed on a domain.

`UPC_ATOMIC_HINT_DEFAULT == 0` An implementation-defined default mode

`UPC_ATOMIC_HINT_LATENCY` Favor low-latency atomic memory operations

`UPC_ATMOIC_HINT_THROUGHPUT` Favor high-throughput atomic memory operations

`UPC_ATOMIC_HINT_*` Implementation-defined additional hint values

7.6.4 Atomic Library Functions

7.6.4.1 The `upc_all_atomicdomain_alloc` function

Synopsis

1
```
#include <upc_atomic.h>
upc_atomicdomain_t *upc_all_atomicdomain_alloc(upc_type_t type,
     upc_op_t ops, upc_atomichint_t hints);
```

Description

2 The `upc_all_atomicdomain_alloc` function dynamically allocates an atomicity domain and returns a pointer to it.

3 The `upc_all_atomicdomain_alloc` function is a *collective* function, with *single-valued* arguments. The return value on every thread points to the same atomicity domain object.

4 The atomicity domain created supports atomic library calls to operate on objects of a unique type, specified by the `type` parameter. The `upc_type_t` values and the corresponding type they specify are listed in UPC Language Specification, Section 7.3.2. The `type` parameter shall specify a type permitted by Section 7.6.2, otherwise behavior is undefined.

5 The `ops` parameter specifies the atomic operations to be supported by the atomicity domain. The `ops` parameter shall only specify operations within the set permitted for `type` (as defined in 7.6.2), otherwise behavior is undefined. Multiple atomic operation value macros from 7.6.2 can be combined by using the bitwise OR operator (|), and each value has a unique bitwise representation that can be unambiguously tested using the bitwise AND operator (`&`).

6 The implementation is free to ignore the `hints` parameter.

7 EXAMPLE: Collectively allocate an atomicity domain that supports the addition, maximum, and minimum operations (i.e., `UPC_ADD`, `UPC_MAX`, `UPC_MIN`) on signed 64-bit integers (i.e., `int64_t`).

```
#include <upc_atomic.h>
upc_atomicdomain_t* domain = upc_all_atomicdomain_alloc(
     UPC_INT64, UPC_ADD | UPC_MAX | UPC_MIN, 0);
```

7.6.4.2 The `upc_all_atomicdomain_free` function

Synopsis

1
```
#include <upc_atomic.h>
void upc_all_atomicdomain_free(upc_atomicdomain_t *ptr);
```

Description

2 The `upc_all_atomicdomain_free` function is a *collective* function, with the *single-valued* argument `ptr`.

3 The `upc_all_atomicdomain_free` function frees the resources associated with the atomicity domain pointed to by `ptr`. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `upc_all_atomicdomain_alloc` function, or if the atomicity domain has been deallocated by a previous call to `upc_all_atomicdomain_free` the behavior is undefined.

4 The atomicity domain referenced by `ptr` is guaranteed to remain valid until all threads have entered the call to `upc_all_atomicdomain_free`, but the function does not otherwise guarantee any synchronization or strict reference.

5 Any subsequent calls to atomic library functions from any thread using `ptr` have undefined behavior.

7.6.4.3 The `upc_atomic_strict` and `upc_atomic_relaxed` functions

Synopsis

1
```
#include <upc_atomic.h>
void upc_atomic_strict(upc_atomicdomain_t *domain,
     void * restrict fetch_ptr, upc_op_t op,
     shared void * restrict target,
     const void * restrict operand1,
     const void * restrict operand2);
void upc_atomic_relaxed(upc_atomicdomain_t *domain,
     void * restrict fetch_ptr, upc_op_t op,
     shared void * restrict target,
     const void * restrict operand1,
     const void * restrict operand2);
```

Description

2 The `op` argument shall specify an operation included in the `ops` argument to the `upc_all_atomicdomain_alloc` call used to construct `domain`, otherwise behavior is undefined.

3 The `target` argument shall point to an object having the type specified in the `type` argument to the `upc_all_atomicdomain_alloc` call used to construct `domain`, otherwise behavior is undefined. The function treats the arguments `fetch_ptr`, `target`, `operand1` and `operand2` as pointers to this type.

4 The `upc_atomic_strict` and `upc_atomic_relaxed` functions perform an atomic update of the object pointed to by `target` such that:

> `*target = *target` $\oplus$ `*operand1`, where "$\oplus$" is the operator specified by the variable `op` and `op` $\in$ {`UPC_AND`, `UPC_OR`, `UPC_XOR`, `UPC_ADD`, `UPC_SUB`, `UPC_MULT`, `UPC_MIN`, `UPC_MAX`}
>
> `*target = *target + 1`, where `op` is `UPC_INC`
>
> `*target = *target - 1`, where `op` is `UPC_DEC`
>
> `*target = (*target == *operand1) ? *operand2 : *target`, where `op` is `UPC_CSWAP`[2]
>
> `*target = *operand1`, where `op` is `UPC_SET`
>
> `*target` is unchanged, where `op` is `UPC_GET`

5 The arguments `operand1` and `operand2` shall each be a null pointer for those operations that do not require them.[3]

6 The value of `*target` prior to performing the specified update is stored in `*fetch_ptr` if and only if `fetch_ptr` is not a null pointer.[4] If `op` is `UPC_GET`, `fetch_ptr` shall not be a null pointer.

7 The following requirements apply when `domain` was allocated with `type` $\in$ {`UPC_FLOAT`, `UPC_DOUBLE`}: If `*target`, `*operand1` or `*operand2` is a *signalling NaN* value (as defined in [ISO/IEC00 Sec 5.2.4.2.2]), behavior is undefined. If `op` is `UPC_CSWAP` and `*target` or `*operand1` is a *quiet NaN* value (as defined in [ISO/IEC00 Sec 5.2.4.2.2]), behavior is undefined.

8 If `domain` was allocated with `type == UPC_PTS` and `op` is `UPC_CSWAP`, the

[2]UPC_CSWAP does not fail spuriously, for example due to cache events.

[3]That is, for all permitted operations other than `UPC_CSWAP`, `operand2` shall be a null pointer, and for `UPC_GET`, `UPC_INC` and `UPC_DEC` both `operand1` and `operand2` shall be a null pointer.

[4]If `op` is `UPC_SET` and `fetch_ptr` is not a null pointer, the effect is an unconditional atomic swap.

comparison shall be performed as specified in [UPC Language Specification Sec. 6.4.2]; specifically, it ignores the phase component of the pointers-to-shared.

9 In all other cases, the value computed by `op` and stored in `*target` shall be equal to the value that would have been computed by passing the operands to the corresponding built-in language operator. In particular, this requires that overflows, underflows and quiet NaN values are handled as specified in [ISO/IEC00].

10 The `upc_atomic_relaxed` function *atomically* performs a relaxed shared read of `*target` followed by a relaxed shared write of `*target`. The `upc_atomic_strict` function *atomically* performs a strict shared read of `*target` followed by a strict shared write of `*target`. The write is omitted for `UPC_GET` or a `UPC_CSWAP` that fails, and the read is omitted for `UPC_SET` when `fetch_ptr` is a null pointer. *Atomically* requires the read and write accesses comprising one atomic operation shall not appear (to any thread) to have been interleaved (or word-torn) with the read/write pair of a conflicting atomic operation to the same location using the same atomicity domain.

11 EXAMPLE: Perform a relaxed atomic fetch-and-increment of a value of type `uint64_t` after allocating an atomicity domain `domain` to support `UPC_INC` for `UPC_UINT64`.

```
#include <upc_atomic.h>
shared uint64_t val = 42;
uint64_t oldval;
upc_atomicdomain_t* domain = upc_all_atomicdomain_alloc(
     UPC_UINT64, UPC_INC, 0);
upc_atomic_relaxed(domain, &oldval, UPC_INC, &val, 0, 0);
```

7.6.4.4 The `upc_atomic_isfast` function

Synopsis

1
```
#include <upc_atomic.h>
int upc_atomic_isfast(upc_type_t type, upc_op_t ops,
     shared void *addr);
```

Description

2 The `upc_atomic_isfast` function queries the implementation to determine the expected performance of a `upc_atomic_relaxed` call on `addr`, using a domain allocated with the arguments `type` and `ops`. The call returns non-zero if the performance is expected to be comparable to the fastest expected performance of `upc_atomic_relaxed` for any combination of `addr`, `type`, and `ops`. Otherwise the function returns zero.[5]

[5]This function allows the implementation to report which combinations of type, ops, and alignment are best supported; e.g., using hardware atomic instructions. Some implementations may also return zero when `upc_threadof(addr)` is not equal to the calling thread, to indicate the additional cost of remote access.

7.7 Castability Functions <upc_castable.h>

1 A UPC implementation may map some or all of the shared address space of another thread into the local address space of the current thread. The functions described in this section allow the programmer to determine whether this is the case, and to make use of this information by providing the ability to obtain valid local addresses for shared cells with affinity to other threads. This capability, sometimes called "privatizability", is referred to as "castability" in this section.

2 Implementations that support this interface shall predefine the feature macro __UPC_CASTABLE__ to the value 1.

7.7.1 Standard headers

1 The standard header is

`<upc_castable.h>`

2 Unless otherwise noted, all of the functions, types and macros specified in Section 7.7 are declared by the header `<upc_castable.h>`.

7.7.2 Castability Functions

7.7.2.1 The `upc_cast` function

Synopsis

1
```
    #include <upc_castable.h>
    void *upc_cast(const shared void *ptr);
```

Description

2 The `upc_cast` function converts the specified pointer-to-shared to a valid pointer-to-local. If such a conversion is not possible, a null pointer is returned.

3 The pointer `ptr` points into one or more shared objects. Consider the portions of all of these shared objects with affinity to `upc_threadof(ptr)`. Choose the shared object containing `ptr` where the portion with this affinity is largest. The conversion performed by the `upc_cast` function will be considered possi-

ble only if this entire portion may be read and written by the current thread based on the returned pointer-to-local value.

4 If the conversion is possible, the pointer-to-shared value is referred to as *castable*.

5 If `upc_threadof(ptr)` is equal to `MYTHREAD`, `upc_cast(ptr)` is equivalent to `(void *)ptr`.

6 If the `ptr` pointer is null, `upc_cast` returns a null pointer.

7 The pointer returned by `upc_cast` is valid only in the calling thread. It cannot be assumed that the return value may be passed to a different thread and used by that thread. It also cannot be assumed that two threads calling `upc_cast` with the same argument will get the same return value.

8 The pointer returned by `upc_cast` remains valid for the lifetime of the referenced shared object. In particular, if the referenced shared object was dynamically allocated, the pointer is no longer valid after the associated shared memory has been freed.

9 If a call to `upc_cast` succeeds, subsequent calls by the same thread with the same pointer, or with a pointer into the same object and with the same affinity, are also guaranteed to succeed for the lifetime of the object.

7.7.2.2 The `upc_thread_info` function

Synopsis

1
```
#include <upc_castable.h>
upc_thread_info_t upc_thread_info(size_t threadId);
```

Description

2 The `upc_thread_info` function returns information about potential uses of the `upc_cast` function in the calling thread in reference to objects with affinity to thread `threadId`. The information is returned in a `upc_thread_info_t` structure, with the following fields:

`int guaranteedCastable`
Indicates which memory regions are guaranteed to be castable.

`int probablyCastable`
Indicates which memory regions are likely (but not guaranteed) to be castable.

3 An implementation may provide additional fields in this structure, allowing `upc_thread_info` to return other information about thread `threadId` with respect to the calling thread.

4 The `guaranteedCastable` and `probablyCastable` fields contain coded integer values indicating memory regions. If the flag for a particular region is set in the `guaranteedCastable` field, it indicates that any pointer into that region with affinity to `threadId` is castable. If the flag is set for a particular region is set in the `probablyCastable` field, it indicates that it is likely, but not guaranteed, that a pointer into that region with affinity to `threadId` is castable.

5 The `<upc_castable.h>` header defines the following macros, which expand to integer constant expressions with type `int`, which are suitable for use in `#if` preprocessing directives. Each macro value designates the specified memory region. The expressions are defined such that each value can be unambiguously tested using the bitwise AND operator (`&`).

`UPC_CASTABLE_ALL_ALLOC`
  Refers to memory allocated via `upc_all_alloc`.

`UPC_CASTABLE_GLOBAL_ALLOC`
  Refers to memory allocated via `upc_global_alloc`.

`UPC_CASTABLE_ALLOC`
  Refers to memory allocated via `upc_alloc`.

`UPC_CASTABLE_STATIC`
  Refers to shared variables defined via static and file scope declarations.

6 Implementations may define additional memory region flags.

7 The macro `UPC_CASTABLE_ALL` shall be defined to be all the region-specific values (including any implementation-specific values) combined via bitwise OR (|) operations. It is defined for convenient testing of whether all shared memory regions are covered in the returned flag.

8 If no memory regions are indicated by the returned flag, the flag value shall be zero.

7.8 UPC Parallel I/O `<upc_io.h>`

1 This subsection provides the UPC parallel extensions of [ISO/IEC00 Sec 7.19]. All the characteristics of library functions described in [ISO/IEC00 Sec 7.1.4] apply to these as well. Implementations that support this interface shall predefine the feature macro `__UPC_IO__` to the value 1.

2 Unless otherwise noted, all of the functions, types and macros specified in Section 7.8 are declared by the header `<upc_io.h>`. [i]

3 Every inclusion of `<upc_io.h>` has the effect of including `<upc_types.h>`. [ii]

Common Constraints

4 All UPC-IO functions are collective and must be called by all threads collectively.[6]

5 If a program calls `exit`, `upc_global_exit`, or returns from `main` with a UPC file still open, the file will automatically be closed at program termination, and the effect will be equivalent to `upc_all_fclose` being implicitly called on the file.

6 If a program attempts to read past the end of a file, the read function will read data up to the end of file and return the number of bytes actually read, which may be less than the amount requested.

7 Writing past the end of a file increases the file size.

8 If a program seeks to a location past the end of a file and writes starting from that location, the data in the intermediate (unwritten) portion of the file is undefined. For example, if a program opens a new file (of size 0 bytes), seeks to offset 1024 and writes some data beginning from that offset, the data at offsets 0–1023 is undefined. Seeking past the end of file and performing a write causes the current file size to immediately be extended up to the end of the write. However, just seeking past the end of file or attempting to read past the end of file, without a write, does not extend the file size.

[i] Issue #91: Library section boilerplate spec text

[ii] Issue #10: Add upc_types.h to define common library types

[6] Note that collective does not necessarily imply barrier synchronization. The synchronization behavior of the UPC-IO data movement library functions is explicitly controlled by using the `upc_flag_t flags` argument. See UPC Language Specification, Section 7.3.3 for details.

9 All generic pointers-to-shared passed to the I/O functions (as function arguments or indirectly through the list I/O arguments) are treated as if they had a phase field of zero (that is, the input phase is ignored).

10 All UPC-IO read/write functions take an argument `flags` of type `upc_flag_t`. The semantics of this argument is defined in UPC Language Specification, Section 7.3.3. These semantics apply only to memory locations in user-provided buffers, not to the read/write operations on the storage medium or any buffer memory internal to the library implementation.

11 The `flags` flag is included even on the fread/fwrite_local functions (which take a pointer-to-local as the buffer argument) in order to provide well-defined semantics for the case where one or more of the pointer-to-local arguments references a shared object (with local affinity). In the case where all of the pointer-to-local arguments in a given call reference only private objects, the `flags` flag provides no useful additional guarantees and is recommended to be passed as `UPC_IN_NOSYNC|UPC_OUT_NOSYNC` to maximize performance.

12 The arguments to all UPC-IO functions are single-valued except where explicitly noted in the function description.

13 UPC-IO, by default, supports weak consistency and atomicity semantics. The default (weak) semantics are as follows. The data written to a file by a thread is only guaranteed to be visible to another thread after all threads have collectively closed or synchronized the file.

14 Writes to a file from a given thread are always guaranteed to be visible to subsequent file reads by the *same* thread, even without an intervening call to collectively close or synchronize the file.

15 Byte-level data consistency is supported.

16 If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined with the weak consistency and atomicity semantics

17 When reading data being concurrently written by another thread, the data that gets read is undefined with the weak consistency and atomicity semantics.

18 File reads into overlapping locations in a shared buffer in memory using individual file pointers or list I/O functions leads to undefined data in the

target buffer under the weak consistency and atomicity semantics.

19 A given file must not be opened at same time by the POSIX I/O and UPC-IO libraries.

20 Except where otherwise noted, all UPC-IO functions return NON-single-valued errors; that is, the occurrence of an error need only be reported to at least one thread, and the `errno` value reported to each such thread may differ. When an error is reported to ANY thread, the position of ALL file pointers for the relevant file handle becomes undefined.

21 The error values that UPC-IO functions may set in `errno` are implementation-defined, however the `perror()` and `strerror()` functions are still guaranteed to work properly with `errno` values generated by UPC-IO.

22 UPC-IO functions can not be called between `upc_notify` and corresponding `upc_wait` statements.

7.8.1 Background

7.8.1.1 File Accessing and File Pointers

1 Collective UPC-IO accesses can be done in and out of shared and private buffers, thus local and shared reads and writes are generally supported. In each of these cases, file pointers could be either common or individual. Note that in UPC-IO, common file pointers cannot be used in conjunction with pointer-to-local buffers. File pointer modes are specified by passing a flag to the collective `upc_all_fopen` function and can be changed using `upc_all_fcntl`. When a file is opened with the common file pointer flag, all threads share a common file pointer. When a file is opened with the individual file pointer flag, each thread gets its own file pointer.

2 UPC-IO also provides file-pointer-independent list file accesses by specifying explicit offsets and sizes of data that is to be accessed. List IO can also be used with either pointer-to-local buffers or pointer-to-shared buffers.

3 Examples 1-3 and their associated figures, Figures 2-4, give typical instances of UPC-IO usage. Error checking is omitted for brevity.

4 EXAMPLE 1: collective read operation using individual file pointers

```
#include <upc.h>
```

```
#include <upc_io.h>
double buffer[10];  // and assuming a total of 4 THREADS
upc_file_t *fd;

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL );
upc_all_fseek( fd, 5*MYTHREAD*sizeof(double), UPC_SEEK_SET );
upc_all_fread_local( fd, buffer, sizeof(double), 10,
                          UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
upc_all_fclose(fd);
```

Each thread reads a block of data into a private buffer from a particular thread-specific offset.

5 EXAMPLE 2: a collective read operation using a common file pointer. The data read is stored into a shared buffer, having a block size of 5 elements. The user selects the type of file pointer at file-open time. The user can select either individual file pointers by passing the flag `UPC_INDIVIDUAL_FP` to the function `upc_all_fopen`, or the common file pointer by passing the flag `UPC_COMMON_FP` to `upc_all_fopen`.

```
#include <upc.h>
#include <upc_io.h>
shared [5] float buffer[20];  // and assuming a total of 4 static THREADS
upc_file_t *fd;

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_COMMON_FP, 0, NULL );
upc_all_fread_shared( fd, buffer, upc_blocksizeof(buffer),
     upc_elemsizeof(buffer), 20, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
/* or equivalently:
 *  upc_all_fread_shared( fd, buffer, 5, sizeof(float), 20,
                                 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
 */
```

7.8.1.2 Synchronous and Asynchronous I/O

1 I/O operations can be synchronous (blocking) or asynchronous (non-blocking). While synchronous calls are quite simple and easy to use from a programming point of view, asynchronous operations allow the overlapping of computation and I/O to achieve improved performance. Synchronous calls block and wait until the corresponding I/O operation is completed. On the other hand, an

asynchronous call starts an I/O operation and returns immediately. Thus, the executing process can turn its attention to other processing needs while the I/O is progressing.

2 UPC-IO supports both synchronous and asynchronous I/O functionality. The asynchronous I/O functions have the same syntax and basic semantics as their synchronous counterparts, with the addition of the `async` suffix in their names. The asynchronous I/O functions have the restriction that only one (collective) asynchronous operation can be active at a time on a given file handle. That is, an asynchronous I/O function must be completed by calling `upc_all_ftest_async` or `upc_all_fwait_async` before another asynchronous I/O function can be called on the same file handle. This restriction is similar to the restriction MPI-IO [MPI2] has on split-collective I/O functions: only one split collective operation can be outstanding on an MPI-IO file handle at any time.

7.8.1.3 Consistency and Atomicity Semantics

1 The consistency semantics define when the data written to a file by a thread is visible to other threads. The atomicity semantics define the outcome of operations in which multiple threads write concurrently to a file or shared buffer and some of the writes overlap each other. For performance reasons, UPC-IO uses weak consistency and atomicity semantics by default. The user can select stronger semantics either by opening the file with the flag `UPC_STRONG_CA` or by calling `upc_all_fcntl` with the command `UPC_SET_STRONG_CA_SEMANTICS`.

2 The default (weak) semantics are as follows. The data written by a thread is only guaranteed to be visible to another thread after all threads have called `upc_all_fclose` or `upc_all_fsync`. (Note that the data *may* be visible to other threads before the call to `upc_all_fclose` or `upc_all_fsync` and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_fclose` or `upc_all_fsync`. Byte-level data consistency is supported. So for example, if thread 0 writes one byte at offset 0 in the file and thread 1 writes one byte at offset 1 in the file, the data from both threads will get written to the file. If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined. Similarly, if one thread tries to read the data being concurrently written by another thread, the data that

gets read is undefined. Concurrent in this context means any two read/write operations to the same file handle with no intervening calls to `upc_all_fsync` or `upc_all_fclose`.

3 For the functions that read into or write from a shared buffer using a common file pointer, the weak consistency semantics are defined as follows. Each call to `upc_all_{fread,fwrite}_shared[_async]` with a common file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different for each operation). In other words, NO file reads are guaranteed to see the results of file writes using the common file pointer until after a close or sync under the default weak consistency semantics.

4 By passing the `UPC_STRONG_CA` flag to `upc_all_fopen` or by calling `upc_all_fcntl` with the command `UPC_SET_STRONG_CA_SEMANTICS`, the user selects strong consistency and atomicity semantics. In this case, the data written by a thread is visible to other threads as soon as the file write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. Overlapping writes to a file in a single (list I/O) write function on a single thread are not permitted (see Section 7.8.6). While strong consistency and atomicity semantics are selected on a given file handle, the `flags` argument to all fread/fwrite functions on that handle is ignored and always treated as `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`.

5 The consistency semantics also define the outcome in the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions. By default, the data in the overlapping space is undefined. If the user selects strong consistency and atomicity, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order. Overlapping reads into memory buffers in a single (list I/O) read function on a single thread are not permitted (see Section 7.8.6).

6 Note that in strong consistency and atomicity mode, atomicity is guaranteed at the UPC-IO function level. The entire operation specified by a single function is performed atomically, regardless of whether it represents a single, contiguous read/write or multiple noncontiguous reads or writes as in a list I/O function.

7 EXAMPLE 1: three threads write data to a file concurrently, each with a single list I/O function. The numbers indicate file offsets and brackets indicate the boundaries of a listed vector. Each thread writes its own thread id as the data values:

```
thread 0:    {1  2  3}   {5  6  7  8}
thread 1: {0  1  2}{3  4  5}
thread 2:             {4  5  6}   {8  9 10 11}
```

8 With the default weak semantics, the results in the overlapping locations are undefined. Therefore, the result in the file would be the following, where x represents undefined data.

```
File:     1  x   x  x  x  x  x  0  x  2  2  2
```

9 That is, the data from thread 1 is written at location 0, the data from thread 0 is written at location 7, and the data from thread 2 is written at locations 9, 10, and 11, because none of these locations had overlapping writes. All other locations had overlapping writes, and consequently, the result at those locations is undefined.

10 If the file were opened with the `UPC_STRONG_CA` flag, strong consistency and atomicity semantics would be in effect. The result, then, would depend on the order in which the writes from the three threads actually occurred. Since six different orderings are possible, there can be six outcomes. Let us assume, for example, that the ordering was the write from thread 0, followed by the write from thread 2, and then the write from thread 1. The (list I/O) write from each thread happens atomically. Therefore, for this ordering, the result would be:

```
File:     1  1  1  1  1  1  2  0  2  2  2  2
```

11 We note that if instead of using a single list I/O function each thread used a separate function to write each contiguous portion, there would be six write functions, two from each thread, and the atomicity would be at the granularity of the write operation specified by each of those functions.

7.8.1.4 File Interoperability

1 UPC-IO does not specify how an implementation may store the data in a file on the storage device. Accordingly, it is implementation-defined whether or not a file created by UPC-IO can be directly accessed by using C/POSIX I/O

functions. However, the UPC-IO implementation must specify how the user can retrieve the file from the storage system as a linear sequence of bytes and vice versa. Similarly, the implementation must specify how familiar operations, such as the equivalent of POSIX `ls`, `cp`, `rm`, and `mv` can be performed on the file.

7.8.2 Predefined Types

1 The following types are defined in `<upc_io.h>`

2 `upc_off_t` is a signed integral type that is capable of representing the size of the largest file supported by the implementation.

3 `upc_file_t` is an opaque shared data type of incomplete type (as defined in [ISO/IEC00 Sec 6.2.5]) that represents an open file handle.

4 `upc_file_t` objects are always manipulated via a pointer (that is, `upc_file_t *`).

5 `upc_file_t` is a shared data type. It is allowed to pass a (`upc_file_t *`) across threads, and two pointers to `upc_file_t` that reference the same logical file handle will always compare equal.

Advice to implementors

6 The definition of `upc_file_t` does not restrict the implementation to store all its metadata with affinity to one thread. Each thread can still have local access to its metadata. For example, below is a simple approach an implementation could use:

```
#include <upc.h>
#include <upc_io.h>
/* for a POSIX-based implementation */
typedef int my_internal_filehandle_t;

#ifdef _UPC_INTERNAL
  typedef struct _local_upc_file_t  {
    my_internal_filehandle_t fd;
    ... other metadata ...
  } local_upc_file_t;
#else
```

```
  struct _local_upc_file_t;
#endif

typedef shared struct _local_upc_file_t upc_file_t;

upc_file_t *upc_all_fopen(...) {

   upc_file_t *handles =
            upc_all_alloc(THREADS, sizeof(upc_file_t));

   /* get my handle */
   upc_file_t *myhandle = &(handles[MYTHREAD]);

   /* cast to a pointer-to-local */
   local_upc_file_t* mylocalhandle = (local_upc_file_t*)myhandle;

   /* setup my metadata using pointer-to-local */
   mylocalhandle->fd = open(...);

   ...

   return handles;
}
```

7 The basic idea is that the "handle" exposed to the user actually points to a cyclic, distributed array. As a result, each thread has easy, local access to its own internal handle metadata with no communication, while maintaining the property that the handle that UPC-IO exposes to the client is a single-valued pointer-to-shared. An additional advantage is that a thread can directly access the metadata for other threads, which may occasionally be desirable in the implementation.

7.8.3 UPC File Operations

Common Constraints

1 When a file is opened with an individual file pointer, each thread will get its own file pointer and advance through the file at its own pace.

2 When a common file pointer is used, all threads positioned in the file will be aligned with that common file pointer.

3 Common file pointers cannot be used in conjunction with pointers-to-local (and hence cannot operate on private objects).

4 No function in this section may be called while an asynchronous operation is pending on the file handle, except where otherwise noted.

7.8.3.1 The `upc_all_fopen` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_file_t *upc_all_fopen(const char *fname, int flags,
         size_t numhints, struct upc_hint const *hints);
```

Description

2 `upc_all_fopen` opens the file identified by the filename `fname` for input/output operations.

3 The flags parameter specifies the access mode. The valid flags and their meanings are listed below. Of these flags, exactly one of `UPC_RDONLY`, `UPC_WRONLY`, or `UPC_RDWR`, and one of `UPC_COMMON_FP` or `UPC_INDIVIDUAL_FP`, must be used. Other flags are optional. Multiple flags can be combined by using the bitwise OR operator (|), and each flag has a unique bitwise representation that can be unambiguously tested using the bitwise AND operator(&).

`UPC_RDONLY` Open the file in read-only mode

`UPC_WRONLY` Open the file in write-only mode

`UPC_RDWR` Open the file in read/write mode

`UPC_INDIVIDUAL_FP` Use an individual file pointer for all file accesses (other than list I/O)

`UPC_COMMON_FP` Use the common file pointer for all file accesses (other than list I/O)

`UPC_APPEND` Set the *initial* position of the file pointer to end of file. (The file pointer is not moved to the end of file after each read/write)

`UPC_CREATE` Create the file if it does not already exist. If the named file does not exist and this flag is not passed, the function fails with an error.

`UPC_EXCL` Must be used in conjunction with `UPC_CREATE`. The open will fail if the file already exists.

`UPC_STRONG_CA` Set strong consistency and atomicity semantics

`UPC_TRUNC` Open the file and truncate it to zero length. The file must be opened before writing.

`UPC_DELETE_ON_CLOSE` Delete the file automatically on close

4 The `UPC_COMMON_FP` flag specifies that all accesses (except for the list I/O operations) will use the common file pointer. The `UPC_INDIVIDUAL_FP` flag specifies that all accesses will use individual file pointers (except for the list I/O operations). Either `UPC_COMMON_FP` or `UPC_INDIVIDUAL_FP` must be specified or `upc_all_fopen` will return an error.

5 The `UPC_STRONG_CA` flag specifies strong consistency and atomicity semantics. The data written by a thread is visible to other threads as soon as the write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. In the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order.

6 If the flag `UPC_STRONG_CA` is not specified, weak semantics are provided. The data written by a thread is only guaranteed to be visible to another thread after all threads have called `upc_all_fclose` or `upc_all_fsync`. (Note that the data *may* be visible to other threads before the call to `upc_all_fclose` or `upc_all_fsync` and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_fclose` or `upc_all_fsync`. Byte-level data consistency is supported. For the purposes of atomicity and consistency semantics, each call to `upc_all_{fread,fwrite}_shared[_async]` with a common file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different

for each operation).”[7]

7 Hints can be passed to the UPC-IO library as an array of key-value pairs[8] of strings. `numhints` specifies the number of hints in the `hints` array; if `numhints` is zero, the `hints` pointer is ignored. The user can free the `hints` array and associated character strings as soon as the open call returns. The following type is defined in `<upc_io.h>`:

```
struct upc_hint
```

holds each element of the `hints` array and contain at least the following initial members, in this order.

```
const char *key;
const char *value;
```

8 `upc_all_fopen` defines a number hints. An implementation is free to support additional hints. An implementation is free to ignore any hint provided by the user. Implementations should *silently* ignore any hints they do not support or recognize. The predefined hints and their meanings are defined below. An implementation is not required to interpret these hint keys, but if it does interpret the hint key, it must provide the functionality described. All hints are single-valued character strings (the content is single-valued, not the location).

`access_style` (comma-separated list of strings): indicates the manner in which the file is expected to be accessed. The hint value is a comma-separated list of any the following: “read_once", “write_once", “read_mostly", “write_mostly", “sequential", and “random". Passing such a hint does not place any constraints on how the file may actually be accessed by the program, although accessing the file in a way that is different from the specified hint may result in lower performance.

`collective_buffering` (boolean): specifies whether the application may benefit from collective buffering optimizations. Allowed values for this key are “true” and “false”. Collective buffering parameters can be further directed via additional hints: cb_buffer_size, and cb_nodes.

`cb_buffer_size` (decimal integer): specifies the total buffer space that the

[7]In other words, NO reads are guaranteed to see the results of writes using the common file pointer until after a close or sync when `UPC_STRONG_CA` is not specified.

[8]The contents of the key/value pairs passed by all the threads must be single-valued.

implementation can use on each thread for collective buffering.

`cb_nodes` (decimal integer): specifies the number of target threads or I/O nodes to be used for collective buffering.

`file_perm` (string): specifies the file permissions to use for file creation. The set of allowed values for this key is implementation defined.

`io_node_list` (comma separated list of strings): specifies the list of I/O devices that should be used to store the file and is only relevant when the file is created.

`nb_proc` (decimal integer): specifies the number of threads that will typically be used to run programs that access this file and is only relevant when the file is created.

`striping_factor` (decimal integer): specifies the number of I/O devices that the file should be striped across and is relevant only when the file is created.

`start_io_device` (decimal integer): specifies the number of the first I/O device from which to start striping the file and is relevant only when the file is created.

`striping_unit` (decimal integer): specifies the striping unit to be used for the file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

9 A file on the storage device is in the *open* state from the beginning of a successful open call to the end of the matching successful close call on the file handle. It is erroneous to have the same file *open* simultaneously with two `upc_all_fopen` calls, or with a `upc_all_fopen` call and a POSIX/C `open` or `fopen` call.

10 The user is responsible for ensuring that the file referenced by the `fname` argument refers to a single UPC-IO file. The actual argument passed on each thread may be different because the file name spaces may be different on different threads, but they must all refer to the same logical UPC-IO file.

11 On success, the function returns a pointer to a file handle that can be used to perform other operations on the file.

12 `upc_all_fopen` provides single-valued errors - if an error occurs, the function returns `NULL` on ALL threads, and sets `errno` appropriately to the same value on all threads.

7.8.3.2 The `upc_all_fclose` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    int upc_all_fclose (upc_file_t *fd);
```

Description

2 `upc_all_fclose` executes an implicit `upc_all_fsync` on `fd` and then closes the file associated with `fd`.

3 The function returns 0 on success. If `fd` is not valid or if an outstanding asynchronous operation on `fd` has not been completed, the function will return an error.

4 `upc_all_fclose` provides single-valued errors - if an error occurs, the function returns -1 on ALL threads, and sets `errno` appropriately to the same value on all threads.

5 After a file has been closed with `upc_all_fclose`, the file is allowed to be opened and the data in it can be accessed by using regular C/POSIX I/O calls.

7.8.3.3 The `upc_all_fsync` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    int upc_all_fsync(upc_file_t *fd);
```

Description

2 `upc_all_fsync` ensures that any data that has been written to the file associated with `fd` but not yet transferred to the storage device is transferred to the storage device. It also ensures that subsequent file reads from any thread will see any previously written values (that have not yet been overwritten).

3 There is an implied barrier immediately before `upc_all_fsync` returns.

4 The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.8.3.4 The `upc_all_fseek` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fseek(upc_file_t *fd, upc_off_t offset,
         int origin);
```

Description

2 `upc_all_fseek` sets the current position of the file pointer associated with `fd`.

3 This offset can be relative to the current position of the file pointer, to the beginning of the file, or to the end of the file. The offset can be negative, which allows seeking backwards.

4 The origin parameter can be specified as `UPC_SEEK_SET`, `UPC_SEEK_CUR`, or `UPC_SEEK_END`, respectively, to indicate that the offset must be computed from the beginning of the file, the current location of the file pointer, or the end of the file.

5 In the case of a common file pointer, all threads must specify the same offset and origin. In the case of an individual file pointer, each thread may specify a different offset and origin.

6 It is allowed to seek past the end of file. It is erroneous to seek to a negative position in the file. See the Common Constraints number 5 at the beginning of Section 7.8.3 for more details.

7 The current position of the file pointer can be determined by calling `upc_all_fseek(fd, 0, UPC_SEEK_CUR)`.

8 On success, the function returns the current location of the file pointer in bytes. If there is an error, it returns –1 and sets `errno` appropriately.

7.8.3.5 The `upc_all_fset_size` function

Synopsis

1

```
#include <upc.h>
#include <upc_io.h>
int upc_all_fset_size(upc_file_t *fd, upc_off_t size);
```

Description

2 `upc_all_fset_size` executes an implicit `upc_all_fsync` on `fd` and resizes the file associated with `fd`. The file handle must be open for writing.

3 `size` is measured in bytes from the beginning of the file.

4 If `size` is less than the current file size, the file is truncated at the position defined by `size`. The implementation is free to deallocate file blocks located beyond this position.

5 If `size` is greater than the current file size, the file size increases to `size`. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (between the old size and size) are undefined.

6 If this function truncates a file, it is possible that the individual and common file pointers may point beyond the end of file. This is allowed and is equivalent to seeking past the end of file (see the Common Constraints in Section 7.8.3 for the semantics of seeking past the end of file).

7 It is unspecified whether and under what conditions this function actually allocates file space on the storage device. Use `upc_all_fpreallocate` to force file space to be reserved on the storage device.

8 Calling this function does not affect the individual or common file pointers.

9 The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.8.3.6 The `upc_all_fget_size` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_fget_size(upc_file_t *fd);
```

Description

2 `upc_all_fget_size` returns the current size in bytes of the file associated

with `fd` on success. On error, it returns –1 and sets `errno` appropriately.

7.8.3.7 The `upc_all_fpreallocate` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
int upc_all_fpreallocate(upc_file_t *fd, upc_off_t size);
```

Description

2 `upc_all_fpreallocate` ensures that storage space is allocated for the first `size` bytes of the file associated with `fd`. The file handle must be open for writing.

3 Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, `upc_all_fpreallocate` has the same effect as writing undefined data.

4 If `size` is greater than the current file size, the file size increases to `size`. If `size` is less than or equal to the current file size, the file size is unchanged.

5 Calling this function does not affect the individual or common file pointers.

6 The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.8.3.8 The `upc_all_fcntl` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
int upc_all_fcntl(upc_file_t *fd,  int cmd, void *arg);
```

Description

2 `upc_all_fcntl` performs one of various miscellaneous operations related to the file specified by `fd`, as determined by `cmd`. The valid commands `cmd` and their associated argument `arg` are explained below.

`UPC_GET_CA_SEMANTICS` Get the current consistency and atomicity semantics for `fd`. The argument `arg` is ignored. The return value is `UPC_STRONG_CA` for strong consistency and atomicity semantics and 0 for the default weak consistency and atomicity semantics.

`UPC_SET_WEAK_CA_SEMANTICS` Executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the weak consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_SET_STRONG_CA_SEMANTICS` Executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the strong consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_GET_FP` Get the type of the current file pointer for `fd`. The argument `arg` is ignored. The return value is either `UPC_COMMON_FP` in case of a common file pointer, or `UPC_INDIVIDUAL_FP` for individual file pointers.

`UPC_SET_COMMON_FP` Executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `fd` to a common file pointer (or leaves it unchanged if that mode is already selected), and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_SET_INDIVIDUAL_FP` Executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `fd` to an individual file pointer (or leaves the mode unchanged if that mode is already selected), and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_GET_FL` Get all the flags specified during the `upc_all_fopen` call for `fd`, as modified by any subsequent mode changes using the `upc_all_fcntl(UPC_SET_*)` commands. The argument `arg` is ignored. The return value has same format as the `flags` parameter in `upc_all_fopen`.

`UPC_GET_FN` Get the file name provided by each thread in the `upc_all_fopen` call that created `fd`. The argument `arg` is a valid (`const char**`) pointing to a (`const char*`) location in which a pointer to file name will be written. Writes a (`const char*`) into `*arg` pointing to the file-name in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fcntl` call on that file handle.

`UPC_GET_HINTS` Retrieve the hints applicable to `fd`. The argument `arg` is a valid (`const upc_hint_t**`) pointing to a (`const upc_hint_t*`) location in which a pointer to the hints array will be written. Writes a (`const upc_hint_t*`) into `*arg` pointing to an array of `upc_hint_t`'s in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fnctl` call on that file handle. The number of hints in the array is returned by the call. The hints in the array may be a subset of those specified at file open time, if the implementation ignored some unrecognized or unsupported hints.

`UPC_SET_HINT` Executes an implicit `upc_all_fsync` on `fd` and sets a new hint to `fd`. The argument `arg` points to one single-valued `upc_hint_t` hint to be applied. If the given hint key has already been applied to `fd`, the current value for that hint is replaced with the provided value. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.

`UPC_ASYNC_OUTSTANDING` Returns 1 if there is an asynchronous operation outstanding on `fd`, or 0 otherwise.

3 In case of a non valid `fd`, `upc_all_fcntl` returns -1 and sets `errno` appropriately.

4 It *is* allowed to call `upc_all_fcntl(UPC_ASYNC_OUTSTANDING)` when an asynchronous operation is outstanding (but it is still disallowed to call `upc_all_fcntl` with any other argument when an asynchronous operation is outstanding).

7.8.4 Reading Data

Common Constraints

1 No function in this section 7.8.4 may be called while an asynchronous operation is pending on the file handle.

7.8.4.1 The `upc_all_fread_local` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fread_local(upc_file_t *fd, void *buffer,
         size_t size, size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fread_local` reads data from a file into a local buffer on each thread.

3 This function can be called only if the current file pointer type is an individual file pointer, and the file handle is open for reading.

4 `buffer` is a pointer to an array into which data will be read, and each thread may pass a different value for `buffer`.

5 Each thread reads `(size*nmemb)` bytes of data from the file at the position indicated by its individual file pointer into the buffer. Each thread may pass a different value for `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O.

6 On success, the function returns the number of bytes read into the local buffer of the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns –1 and sets `errno` appropriately.

7.8.4.2 The `upc_all_fread_shared` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fread_shared(upc_file_t *fd,
         shared void *buffer, size_t blocksize, size_t size,
         size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fread_shared` reads data from a file into a shared buffer in memory.

3 The function can be called when the current file pointer type is either a common file pointer or an individual file pointer. The file handle must be open for reading.

4 `buffer` is a pointer to an array into which data will be read. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of 0 indicates an indefinite blocking factor.

5 In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. Each thread reads `(size*nmemb)` bytes of data from the file at the position indicated by its individual file pointer into its buffer. Each thread may pass a different value for `blocksize`, `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes read by the calling thread, and the individual file pointer of the thread is incremented by that amount.

6 In the case of a common file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter, and the same value for `blocksize`, `size` and `nmemb`. The effect is that `(size*nmemb)` bytes of data are read from the file at the position indicated by the common file pointer into the buffer. If `size` or `nmemb` is zero, the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes read by all threads, and the common file pointer is incremented by that amount.

7 If reading with individual file pointers results in overlapping reads into the shared buffer, the result is determined by whether the file was opened with the `UPC_STRONG_CA` flag or not (see Section 7.8.3.1).

8 The function returns –1 on error and sets `errno` appropriately.

7.8.5 Writing Data

Common Constraints

1 No function in this section 7.8.5 may be called while an asynchronous operation is pending on the file handle.

7.8.5.1 The `upc_all_fwrite_local` function

Synopsis

1

```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fwrite_local(upc_file_t *fd, void *buffer,
         size_t size, size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fwrite_local` writes data from a local buffer on each thread into a file.

3 This function can be called only if the current file pointer type is an individual file pointer, and the file handle is open for writing.

4 `buffer` is a pointer to an array from which data will be written, and each thread may pass a different value for `buffer`.

5 Each thread writes `(size*nmemb)` bytes of data from the buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O.

6 If any of the writes result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see 7.8.3.1).

7 On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns –1 and sets `errno` appropriately.

7.8.5.2 The `upc_all_fwrite_shared` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fwrite_shared(upc_file_t *fd,
         shared void *buffer, size_t blocksize,  size_t size,
         size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fwrite_shared` writes data from a shared buffer in memory to a file.

3 The function can be called if the current file pointer type is either a common

file pointer or an individual file pointer. The file handle must be open for writing.

4 `buffer` is a pointer to an array from which data will be written. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of 0 indicates an indefinite blocking factor.

5 In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. Each thread writes `(size*nmemb)` bytes of data from its buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for `blocksize`, `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount.

6 In the case of a common file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter, and the same value for `blocksize`, `size` and `nmemb`. The effect is that `(size*nmemb)` bytes of data are written from the buffer to the file at the position indicated by the common file pointer. If `size` or `nmemb` is zero, the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes written by all threads, and the common file pointer is incremented by that amount.

7 If writing with individual file pointers results in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.8.3.1).

8 The function returns –1 on error and sets `errno` appropriately.

7.8.6 List I/O

Common Constraints

1 List I/O functions take a list of addresses/offsets and corresponding lengths in memory and file to read from or write to.

2 List I/O functions can be called regardless of whether the current file pointer type is individual or common.

3 File pointers are not updated as a result of a list I/O read/write operation.

4 Types declared in `<upc_io.h>` are

```
struct upc_local_memvec
```

which contains at least the initial members, in this order:

```
void *baseaddr;
size_t len;
```

and is a memory vector element pointing to a contiguous region of local memory.

5

```
struct upc_shared_memvec
```

which contains at least the initial members, in this order:

```
shared void *baseaddr;
size_t blocksize;
size_t len;
```

and is a memory vector element pointing to a blocked region of shared memory.

6

```
struct upc_filevec
```

which contains at least the initial members, in this order:

```
upc_off_t offset;
size_t len;
```

and is a file vector element pointing to a contiguous region of a file.

For all cases these vector element types specify regions which are `len` bytes long. If `len` is zero, the entry is ignored. `blocksize` is the block size of the shared buffer in bytes. A `blocksize` of 0 indicates an indefinite blocking factor.

7 The `memvec` argument passed to any list I/O *read* function by a single thread must not specify overlapping regions in memory.

8 The base addresses passed to `memvec` can be in any order.

9 The `filevec` argument passed to any list I/O *write* function by a single thread must not specify overlapping regions in the file.

10 The offsets passed in `filevec` must be in monotonically non-decreasing order.

11 No function in this section (7.8.6) may be called while an asynchronous operation is pending on the file handle.

12 No function in this section (7.8.6) implies the presence of barriers at entry or exit. However, the programmer is advised to use a barrier after calling `upc_all_fread_list_shared` to ensure that the entire shared buffer has been filled up, and similarly, use a barrier before calling `upc_all_fwrite_list_shared` to ensure that the entire shared buffer is up-to-date before being written to the file.

13 For all the list I/O functions, each thread passes an independent set of memory and file vectors. Passing the same vectors on two or more threads specifies redundant work. The file pointer is a single-valued argument, all other arguments to the list I/O functions are NOT single-valued.

14 EXAMPLE 1: a collective list I/O read operation. The list I/O functions allow the user to specify noncontiguous accesses both in memory and file in the form of lists of explicit offsets and lengths in the file and explicit address and lengths in memory. None of the file pointers are used or updated in this case.

```
#include <upc.h>
#include <upc_io.h>
char buffer[12];
struct upc_local_memvec memvec[2] = {(&buffer[0],4},{&buffer[7],3}};
struct upc_filevec filevec[2];
upc_file_t *fd;

fd = upc_all_fopen( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL );
filevec[0].offset = MYTHREAD*5;
filevec[0].len = 2;
filevec[1].offset = 10+MYTHREAD*5;
filevec[1].len = 5;

upc_all_fread_list_local( fd, 2, &memvec, 2, &filevec,
                                            UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

7.8.6.1 The `upc_all_fread_list_local` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fread_list_local(upc_file_t *fd,
         size_t memvec_entries, struct upc_local_memvec const *memvec,
         size_t filevec_entries, struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fread_list_local` reads data from a file into local buffers in memory. The file handle must be open for reading.

3 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

4 The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

5 On success, the function returns the number of bytes read by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.8.6.2 The `upc_all_fread_list_shared` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fread_list_shared(upc_file_t *fd,
         size_t memvec_entries, struct upc_shared_memvec const *memvec,
         size_t filevec_entries, struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fread_list_shared` reads data from a file into various locations of a shared buffer in memory. The file handle must be open for reading.

3 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored

and no locations are specified for I/O.

4 The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

5 If any of the reads from different threads result in overlapping regions in memory, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.8.3.1).

6 On success, the function returns the number of bytes read by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.8.6.3 The `upc_all_fwrite_list_local` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fwrite_list_local(upc_file_t *fd,
         size_t memvec_entries, struct upc_local_memvec const *memvec,
         size_t filevec_entries, struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fwrite_list_local` writes data from local buffers in memory to a file. The file handle must be open for writing.

3 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

4 The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

5 If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.8.3.1).

6 On success, the function returns the number of bytes written by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.8.6.4 The `upc_all_fwrite_list_shared` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fwrite_list_shared(upc_file_t *fd,
         size_t memvec_entries, struct upc_shared_memvec const *memvec,
         size_t filevec_entries, struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fwrite_list_shared` writes data from various locations of a shared buffer in memory to a file. The file handle must be open for writing.

3 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

4 The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.

5 If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.8.3.1).

6 On success, the function returns the number of bytes written by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.8.7 Asynchronous I/O

Common Constraints

1 Only one asynchronous I/O operation can be outstanding on a UPC-IO file handle at any time. If an application attempts to initiate a second asyn-

chronous I/O operation while one is still outstanding on the same file handle the behavior is undefined – however, high-quality implementations will issue a fatal error.

2 For asynchronous read operations, the contents of the destination memory are undefined until after a successful `upc_all_fwait_async` or `upc_all_ftest_async` on the file handle. For asynchronous write operations, the source memory may not be safely modified until after a successful `upc_all_fwait_async` or `upc_all_ftest_async` on the file handle.

3 An implementation is free to block for completion of an operation in the asynchronous initiation call or in the `upc_all_ftest_async` call (or both). High-quality implementations are recommended to minimize the amount of time spent within the asynchronous initiation or `upc_all_ftest_async` call.

4 In the case of list I/O functions, the user may modify or free the lists after the asynchronous I/O operation has been initiated.

5 The semantics of the flags of type `upc_flag_t` when applied to the async variants of the fread/fwrite functions should be interpreted as follows: constraints that reference entry to a function call correspond to entering the fread_async-/fwrite_async call that initiates the asynchronous operation, and constraints that reference returning from a function call correspond to returning from the `upc_all_fwait_async()` or successful `upc_all_ftest_async()` call that completes the asynchronous operation. Also, note that the flags which govern an asynchronous operation are passed to the library during the asynchronous initiation call.

7.8.7.1 The `upc_all_fread_local_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
void upc_all_fread_local_async(upc_file_t *fd, void *buffer,
     size_t size, size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fread_local_async` initiates an asynchronous read from a file into a local buffer on each thread.

3 The meaning of the parameters and restrictions are the same as for the

blocking function, `upc_all_fread_local`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.2 The `upc_all_fread_shared_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
void upc_all_fread_shared_async(upc_file_t *fd,
     shared void *buffer, size_t blocksize, size_t size,
     size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fread_shared_async` initiates an asynchronous read from a file into a shared buffer.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_shared`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.3 The `upc_all_fwrite_local_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
void upc_all_fwrite_local_async(upc_file_t *fd, void *buffer,
     size_t size, size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fwrite_local_async` initiates an asynchronous write from a local buffer on each thread to a file.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_local`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.4 The upc_all_fwrite_shared_async function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    void upc_all_fwrite_shared_async(upc_file_t *fd,
         shared void *buffer, size_t blocksize,size_t size,
         size_t nmemb, upc_flag_t flags);
```

Description

2 `upc_all_fwrite_shared_async` initiates an asynchronous write from a shared buffer to a file.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_shared`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.5 The upc_all_fread_list_local_async function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    void upc_all_fread_list_local_async(upc_file_t *fd,
         size_t memvec_entries, struct upc_local_memvec const *memvec,
         size_t filevec_entries,  struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fread_list_local_async` initiates an asynchronous read of data from a file into local buffers in memory.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_list_local`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.6 The `upc_all_fread_list_shared_async` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    void upc_all_fread_list_shared_async(upc_file_t *fd,
         size_t memvec_entries, struct upc_shared_memvec const *memvec,
         size_t filevec_entries,  struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fread_list_shared_async` initiates an asynchronous read of data from a file into various locations of a shared buffer in memory.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fread_list_shared`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.7 The `upc_all_fwrite_list_local_async` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    void upc_all_fwrite_list_local_async(upc_file_t *fd,
         size_t memvec_entries, struct upc_local_memvec const *memvec,
         size_t filevec_entries, struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fwrite_list_local_async` initiates an asynchronous write of data from local buffers in memory to a file.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_list_local`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.8 The `upc_all_fwrite_list_shared_async` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    void upc_all_fwrite_list_shared_async(upc_file_t *fd,
         size_t memvec_entries, struct upc_shared_memvec const *memvec,
         size_t filevec_entries, struct upc_filevec const *filevec,
         upc_flag_t flags);
```

Description

2 `upc_all_fwrite_list_shared_async` initiates an asynchronous write of data from various locations of a shared buffer in memory to a file.

3 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_fwrite_list_shared`.

4 The status of the initiated asynchronous I/O operation can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

7.8.7.9 The `upc_all_fwait_async` function

Synopsis

1
```
    #include <upc.h>
    #include <upc_io.h>
    upc_off_t upc_all_fwait_async(upc_file_t *fd)
```

Description

2 `upc_all_fwait_async` completes the previously issued asynchronous I/O operation on the file handle `fd`, blocking if necessary.

3 It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.

4 On success, the function returns the number of bytes read or written by the asynchronous I/O operation as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns –1 and sets `errno` appropriately, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

7.8.7.10 The `upc_all_ftest_async` function

Synopsis

1
```
#include <upc.h>
#include <upc_io.h>
upc_off_t upc_all_ftest_async(upc_file_t *fd, int *flag)
```

Description

2 `upc_all_ftest_async` tests whether the outstanding asynchronous I/O operation associated with `fd` has completed.

3 If the operation has completed, the function sets `flag=1` and the asynchronous operation becomes no longer outstanding;[9] otherwise it sets `flag=0`. The same value of `flag` is set on all threads.

4 If the operation was completed, the function returns the number of bytes that were read or written as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns –1 and sets `errno` appropriately, and sets the `flag=1`, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

5 It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.

[9]This implies it is disallowed to call `upc_all_fwait_async` or `upc_all_ftest_async` immediately after a successful `upc_all_ftest_async` on that file handle.

7.9 UPC Non-Blocking Transfer Operations `<upc_nb.h>`

1 Implementations that support this interface shall predefine the feature macro `__UPC_NB__` to the value 1.

7.9.1 Standard header

1 The standard header is

`<upc_nb.h>`

2 Unless otherwise noted, all of the functions, types and macros specified in Section 7.9 are declared by the header `<upc_nb.h>`.

7.9.2 Common Requirements

1 The following requirements apply to all of the functions defined in Section 7.9.

2 This section defines extensions to the `upc_mem*` functions defined in [UPC Language Specifications, Section 7.2.5]. Data transfer effects are as specified in that document.

3 `<upc_nb.h>` defines two non-blocking variants for each `upc_mem*` function. The `_nb` function name suffix denotes the *explicit-handle* variant, whereas the `_nbi` function name suffix denotes the *implicit-handle* variant. These functions are jointly referred to as *transfer initiation functions*. A thread which invokes one of these functions is referred to as the *initiating thread* for the transfer.

4 A transfer initiation function returns as soon as possible after initiating the transfer and may return prior to the effects of the transfer being completed. [10]

[10] Each call to a transfer initiation function shall either initiate an asynchronous transfer or perform the transfer synchronously within the initiation call (and return `UPC_COMPLETE_HANDLE` in the case of an explicit-handle initiation). The conditions governing this decision are unspecified. For example, an implementation might choose to perform a synchronous transfer when all affected memory has affinity to the initiating thread. Implementations are encouraged to perform asynchronous transfers and return quickly whenever possible to allow the caller to overlap unrelated computation and com-

Generally the initiating thread must later take explicit action to synchronize the completion of the transfer.

5 The explicit-handle variant returns a handle that gives the initiating thread explicit control and responsibility to manage completion of the transfer. The initiating thread shall eventually invoke a successful `upc_sync[_attempt]` function call upon each such handle, to synchronize completion of the associated transfer and allow the implementation to reclaim resources that may be associated with the handle.

6 The implicit-handle variant allows the program to synchronize completion of an implicit group of transfers together, at the next call to `upc_synci[_attempt]` by the initiating thread.

7 Each call entry to a transfer initiation function defines the beginning of an abstract interval referred to as the *transfer interval* for the transfer operation being performed. The *transfer interval* extends until the return of the successful `upc_sync[_attempt]` or `upc_synci[_attempt]` call which synchronizes the completion of the transfer.

8 Each non-blocking transfer proceeds independently of all other operations and actions by any thread until the end of the transfer interval. In particular, the transfer interval may extend beyond strict operations and other forms of inter-thread synchronization.

9 The order in which non-blocking transfers complete is unspecified - the implementation may 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 guarantees are those explicitly enforced using the synchronization functions, i.e. all the accesses comprising the transfer are guaranteed to occur during the transfer interval.

10 Throughout the transfer interval, the contents of all destination memory specified by the transfer are undefined. Specifically, concurrent reads to these locations from any thread will observe an indeterminate value.

11 If any of the source or destination memory specified by a transfer is modified by any thread during the transfer interval, then the results of the transfer are undefined. Specifically, concurrent writes to these locations will result in indeterminate values in the destination memory which persist after synchro-

munication.

nization.

12 The source memory specified in a transfer is not modified by the transfer. Concurrent reads of source memory areas by any thread are permitted and behave as usual. Multiple concurrent transfers initiated by any thread are permitted to specify overlapping source memory areas. If a transfer specifies destination memory which overlaps its own source, or the source or destination of a concurrent transfer initiated by any thread, the resulting values in all destination memory specified by the affected transfers are indeterminate.

13 The memory consistency semantics of all transfers performed by the library are as described in [UPC Language Specifications, Section B.3.2.1]. Specifically, the effect on conflicting accesses issued *outside* the transfer interval is as if the transfer were performed by a set of relaxed shared reads and relaxed shared writes of unspecified size and order, issued at unspecified times anywhere within the transfer interval by the initiating thread. Conflicting accesses *inside* the transfer interval have undefined results, as specified in the preceding paragraphs. [11] Here *inside* and *outside* are defined by the `Precedes()` program order for accesses issued by the initiating thread; accesses issued by other threads are considered *inside* unless every possible and valid $<_{strict}$ relationship orders them outside the transfer interval. [12]

[11]The restrictions described in the three preceding paragraphs are a direct consequence of [UPC Language Specifications, Section B.3.2.1], and also apply to the blocking `upc_mem*` functions. They are explicitly stated here for clarity.

[12] Stated differently, a successful `upc_sync[_attempt]` or `upc_synci[_attempt]` call completes transfers with respect to the initiating thread, and subsequent relaxed accesses issued by the initiating thread are guaranteed to observe the effects of the synchronized transfer(s).

Similarly, a successful `upc_sync[_attempt]` or `upc_synci[_attempt]` call followed by a strict operation ensures the effects of the synchronized transfer(s) will be observed by all threads prior to observing the strict operation.

7.9.3 Explicit Handle Type

1 An implementation shall define the following type and value:

```
type  upc_handle_t
value UPC_COMPLETE_HANDLE
```

2 `UPC_COMPLETE_HANDLE` shall have type `upc_handle_t`. All of its bits shall be 0.

3 Any handle value other than `UPC_COMPLETE_HANDLE` is valid only for the initiating thread which obtained it from an explicit-handle transfer initiation function. Different threads shall not use it for any purpose.

4 Every handle value returned from an explicit-handle transfer initiation function call shall eventually be passed to a successful `upc_sync[_attempt]` call. It is an error to discard a handle value and never synchronize it unless the value is `UPC_COMPLETE_HANDLE`.

5 Once a handle value is successfully synchronized, it becomes invalid and shall not be used for any purpose.

7.9.4 Explicit-handle transfer initiation functions

7.9.4.1 The `upc_memcpy_nb` function

Synopsis

1
```
#include <upc_nb.h>
upc_handle_t upc_memcpy_nb( shared void * restrict dst,
                            shared const void * restrict src,
                            size_t n );
```

Description

2 The transfer initiated by `upc_memcpy_nb(dst, src, n)` shall have the same effects as `upc_memcpy(dst, src, n)`. If the returned value is `UPC_COMPLETE_HANDLE`, then these effects were performed synchronously and the transfer is complete. Otherwise, the transfer interval extends until the return of a successful `upc_sync[_attempt]` call upon the returned handle.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

4 The following two code sequences demonstrate the relationship between `upc_memcpy` and `upc_memcpy_nb`. Both transfers ultimately result in the same data movement.

```
upc_memcpy( dst, src, n ); // perform an explicitly synchronous transfer
...  // code that may access dst and src regions
...  // accesses by THIS thread guaranteed to observe the effects
upc_fence; // any strict operation
...  // subsequent accesses by ANY thread guaranteed to observe the effects
```

```
upc_handle_t handle = upc_memcpy_nb( dst, src, n ); // initiate a transfer
...  // code that must not read dst region or modify either region
upc_sync( handle );   // sync the handle, ending the transfer interval
...  // accesses by THIS thread guaranteed to observe the effects
upc_fence; // any strict operation
...  // subsequent accesses by ANY thread guaranteed to observe the effects
```

7.9.4.2 The `upc_memget_nb` function

Synopsis

1
```
#include <upc_nb.h>
upc_handle_t upc_memget_nb( void * restrict dst,
                            shared const void * restrict src,
                            size_t n );
```

Description

2 The transfer initiated by `upc_memget_nb(dst, src, n)` shall have the same effects as `upc_memget(dst, src, n)`. If the returned value is `UPC_COMPLETE_HANDLE`, then these effects were performed synchronously and the transfer is complete. Otherwise, the transfer interval extends until the return of a successful `upc_sync[_attempt]` call upon the returned handle.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

7.9.4.3 The `upc_memput_nb` function

Synopsis

1
```
#include <upc_nb.h>
upc_handle_t upc_memput_nb( shared void * restrict dst,
                            const void * restrict src,
                            size_t n );
```

Description

2 The transfer initiated by `upc_memput_nb(dst, src, n)` shall have the same effects as `upc_memput(dst, src, n)`. If the returned value is `UPC_COMPLETE_HANDLE`, then these effects were performed synchronously and the transfer is complete. Otherwise, the transfer interval extends until the return of a successful `upc_sync[_attempt]` call upon the returned handle.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

7.9.4.4 The `upc_memset_nb` function

Synopsis

1
```
#include <upc_nb.h>
upc_handle_t upc_memset_nb( shared void *dst, int c, size_t n );
```

Description

2 The transfer initiated by `upc_memset_nb(dst, c, n)` shall have the same effects as `upc_memset(dst, c, n)`. If the returned value is `UPC_COMPLETE_HANDLE`, then these effects were performed synchronously and the transfer is complete. Otherwise, the transfer interval extends until the return of a successful `upc_sync[_attempt]` call upon the returned handle.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

7.9.5 Implicit-handle transfer initiation functions

7.9.5.1 The `upc_memcpy_nbi` function

Synopsis

1
```
#include <upc_nb.h>
void upc_memcpy_nbi( shared void * restrict dst,
                     shared const void * restrict src,
                     size_t n );
```

Description

2 The transfer initiated by `upc_memcpy_nbi(dst, src, n)` shall have the same effects as `upc_memcpy(dst, src, n)`. The transfer interval extends until the return of the next successful `upc_synci[_attempt]` call performed by the initiating thread.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

4 The following two code sequences demonstrate the relationship between `upc_memcpy` and `upc_memcpy_nbi`. Both transfers ultimately result in the same data movement.

```
upc_memcpy( dst, src, n ); // perform an explicitly synchronous transfer
...  // code that may access dst and src regions
...  // accesses by THIS thread guaranteed to observe the effects
upc_fence; // any strict operation
...  // subsequent accesses by ANY thread guaranteed to observe the effects
```

```
upc_memcpy_nbi( dst, src, n ); // initiate a transfer
...  // code that must not read dst region or modify either region
upc_synci(); // sync all implicit-handle ops, ending the transfer interval(s)
...  // accesses by THIS thread guaranteed to observe the effects
upc_fence; // any strict operation
...  // subsequent accesses by ANY thread guaranteed to observe the effects
```

7.9.5.2 The `upc_memget_nbi` function

Synopsis

1
```
#include <upc_nb.h>
void upc_memget_nbi( void * restrict dst,
                     shared const void * restrict src,
                     size_t n );
```

Description

2 The transfer initiated by `upc_memget_nbi(dst, src, n)` shall have the same effects as `upc_memget(dst, src, n)`. The transfer interval extends until the return of the next successful `upc_synci[_attempt]` call performed by the initiating thread.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

7.9.5.3 The `upc_memput_nbi` function

Synopsis

1
```
#include <upc_nb.h>
void upc_memput_nbi( shared void * restrict dst,
                     const void * restrict src,
                     size_t n );
```

Description

2 The transfer initiated by `upc_memput_nbi(dst, src, n)` shall have the same effects as `upc_memput(dst, src, n)`. The transfer interval extends until the return of the next successful `upc_synci[_attempt]` call performed by the initiating thread.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

7.9.5.4 The `upc_memset_nbi` function

Synopsis

1
```
#include <upc_nb.h>
void upc_memset_nbi( shared void *dst, int c, size_t n );
```

Description

2 The transfer initiated by `upc_memset_nbi(dst, c, n)` shall have the same effects as `upc_memset(dst, c, n)`. The transfer interval extends until the return of the next successful `upc_synci[_attempt]` call performed by the initiating thread.

3 All of the common requirements listed in Section 7.9.2 apply to this function.

7.9.6 Explicit-handle synchronization functions

7.9.6.1 The `upc_sync_attempt` function

Synopsis

1
```
#include <upc_nb.h>
int upc_sync_attempt( upc_handle_t handle );
```

Description

2 `handle` shall be a valid handle value returned by an explicit-handle transfer initiation function to the current thread, or the value `UPC_COMPLETE_HANDLE`.

3 The `upc_sync_attempt` function always returns immediately, without blocking. It returns non-zero if the transfer associated with `handle` is complete, thereby ending the transfer interval. Otherwise, it returns 0.

4 If `handle == UPC_COMPLETE_HANDLE` then the `upc_sync_attempt` function returns non-zero. Otherwise, if the function returns non-zero then the handle value is consumed and shall not be subsequently used for any purpose.

7.9.6.2 The `upc_sync` function

Synopsis

1
```
#include <upc_nb.h>
void upc_sync( upc_handle_t handle );
```

Description

2 `handle` shall be a valid handle value returned by an explicit-handle transfer initiation function to the current thread, or the value `UPC_COMPLETE_HANDLE`.

3 The `upc_sync` function does not return until the transfer associated with the `handle` is complete, ending the transfer interval.

4 If `handle == UPC_COMPLETE_HANDLE` then the `upc_sync` function returns immediately. Otherwise, the handle value is consumed by this function and shall not be subsequently used for any purpose.

7.9.7 Implicit-handle synchronization functions

7.9.7.1 The `upc_synci_attempt` function

Synopsis

1
```
#include <upc_nb.h>
int upc_synci_attempt( void );
```

Description

2 The `upc_synci_attempt` function always returns immediately, without blocking. It returns non-zero if all implicit-handle transfers previously initiated by the calling thread (but not yet synchronized) are complete, thereby ending those transfer intervals. Otherwise, it returns 0.

3 If there are no such pending implicit-handle transfers, the function returns non-zero.

4 The `upc_synci_attempt` function does not complete explicit-handle transfers.

7.9.7.2 The `upc_synci` function

Synopsis

1
```
#include <upc_nb.h>
void upc_synci( void );
```

Description

2 The `upc_synci` function does not return until all implicit-handle transfers previously initiated by the calling thread (but not yet synchronized) are complete, thereby ending those transfer intervals.

3 If there are no such pending implicit-handle transfers, the function returns immediately.

4 The `upc_synci` function does not complete explicit-handle transfers.

Index