Acknowledgments

Many have contributed to the ideas and concepts behind these specifications. William Carlson, Jesse Draper, David Culler, Katherine Yelick, Eugene Brooks, and Karen Warren are the authors of the initial UPC language concepts and specifications. Tarek El-Ghazawi, William Carlson, and Jesse Draper are the authors of the first formal version of the specifications. Because of the numerous contributions to the specifications, no explicit authors are currently mentioned. We also would like to acknowledge the role of the participants in the first UPC workshop, held in May 2000 in Bowie, Maryland, and in which the specifications of this version were discussed. In particular we would like to acknowledge the support and participation of Compaq, Cray, HP, Sun, and CSC. We would like also to acknowledge the abundant input of Kevin Harris and Sébastien Chauvin and the efforts of Lauren Smith. The efforts of Brian Wibecan and Greg Fischer were invaluable in bringing these specifications to version 1.0.

Version 1.1 is the result of the contributions of many in the UPC community, most importantly the participants in the second UPC workshop held in March 2002 in Washington, DC. In addition to the continued support of all those mentioned above, the efforts of Dan Bonachea were invaluable in this effort.

Version 1.2 is also the result of many contributors. Worthy of special note (in addition to the continued support of those mentioned above) are the substantial contributions to many aspects of the specifications by Jason Duell; add note crediting collective, io, and mem model authors when included in 1.2.

Members of the UPC consortium may be contacted via the world wide web at http://www.upcworld.org or http://upc.gwu.edu, where an archived mailing list may be joined. Comments on these specifications are always welcome.
# Contents

Introduction 5

1 Scope 5

2 Normative references 5

3 Terms, definitions and symbols 6

4 Conformance 8

5 Environment 9

5.1 Conceptual models 9

5.1.1 Translation environment 9

5.1.2 Execution environment 9

6 Language 11

6.1 Notations 11

6.2 Predefined identifiers 12

6.2.1 THREADS 12

6.2.2 MYTHREAD 12

6.2.3 UPC_MAX_BLOCK_SIZE 12

6.3 Expressions 12

6.3.1 The upc.localsizeof operator 12

6.3.2 The upc.blocksiz eof operator 13

6.3.3 The upc.elemsizeof operator 14

6.3.4 Pointer-to-shared arithmetic 14

6.3.5 Cast and assignment expressions 16

6.3.6 Address operators 17

6.4 Declarations 17

6.4.1 Type qualifiers 18

6.4.2 The shared and reference type qualifiers 18

6.4.3 Declarators 20

6.5 Statements and blocks 23

6.5.1 Barrier statements 24

6.5.2 Iteration statements 26

6.6 Preprocessing directives 28

6.6.1 UPC pragmas 28
6.6.2 Predefined macro names .................. 29

7 Library ........................................ 29
  7.1 Standard headers ........................... 29
  7.2 UPC utilities <upc.h> ....................... 30
    7.2.1 Termination of all threads .......... 30
    7.2.2 Shared memory allocation functions .. 30
    7.2.3 Pointer-to-shared manipulation functions .... 33
    7.2.4 Lock functions ....................... 35
    7.2.5 Shared string handling functions ..... 38

References .................................... 41

A UPC versus C Standard section numbering .... 41
Introduction

1 UPC is a parallel extension to the C Standard. UPC follows the distributed shared-memory programming paradigm. The first version of UPC, known as version 0.9, was published in May of 1999 as technical report [CARLSON99] at the Institute for Defense Analyses Center for Computing Sciences.

2 Version 1.0 of UPC was initially discussed at the UPC workshop, held in Bowie, Maryland, 18-19 May, 2000. The workshop had about 50 participants from industry, government, and academia. This version was adopted with modifications in the UPC mini workshop meeting held during Supercomputing 2000, in November 2000, in Dallas, and finalized in February 2001.

3 Version 1.1 of UPC was initially discussed at the UPC workshop, held in Washington, DC, 3-5 March, 2002, and finalized in October 2003.

4 Version 1.2 of UPC was initially discussed at the UPC workshop held in Phoenix, AZ, 20 November 2003.

1 Scope

1 This document focuses only on the UPC specifications that extend the C Standard to an explicit parallel C based on the distributed shared memory model. All C specifications as per ISO/IEC 9899 [ISO/IEC00] are considered a part of these UPC specifications, and therefore will not be addressed in this document.

2 Small parts of the C Standard [ISO/IEC00] may be repeated for self-containment and clarity of a subsequent UPC extension definition.

2 Normative references

1 The following document and its identified normative references constitute provisions of these UPC specifications.

2 ISO/IEC 9899: 1999(E), Programming languages - C [ISO/IEC00]
The relationship between the section numbering used in the C Standard [ISO/IEC00] and that used in this document is given in Appendix A and noted at the beginning of each corresponding section.

3 Terms, definitions and symbols

For the purpose of these specifications the following definitions apply.

Other terms are defined where they appear in italic type or on the left hand side of a syntactical rule.

3.1 thread
an instance of execution initiated by the execution environment at program startup.

3.2 object
region of data storage in the execution environment which can represent values.

3.2.1 shared object
an object created using a shared-qualified declarator or constructed by a library function defined to create shared objects.

NOTE All threads may access shared objects.¹

3.2.1 private object
any object which is not a shared object.

NOTE Each thread declares and creates its own private objects which no other thread can access.²

¹The file scope declaration shared int x; creates a single object which any thread may access.
²The file scope declaration int y; creates a separate object for each thread to access.
affinity
logical association between shared objects and threads. Each element of data
storage that contains shared objects has affinity to exactly one thread.

pointer-to-shared
a pointer whose referenced type is shared-qualified.

pointer-to-local
a pointer whose referenced type is not shared-qualified.

access
<execution-time action> to read or modify the value of an object by a thread.

shared access
an access using an expression whose type is shared-qualified.

strict shared read
a shared read access which is determined to be strict according to section
6.4.2 of this specification.

strict shared write
a shared modify access which is determined to be strict according to section
6.4.2 of this specification.

relaxed shared read
a shared read access which is determined to be relaxed according to section
6.4.2 of this specification.

relaxed shared write
a shared modify access which is determined to be relaxed according to section
6.4.2 of this specification.
1 local access
an access using an expression whose type is not shared-qualified.

3.7

1 collective
a constraint placed on some language operations which requires evaluation of such operations to be matched\(^3\) across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.

3.8

1 single-valued
an operand to a collective operation, which has the same value on every thread. The behavior of the operation is otherwise undefined.

3.9

1 phase
an unsigned integer value associated with a pointer-to-shared which indicates the element-offset within an affinity block; used in pointer-to-shared arithmetic to determine affinity boundaries.

4 Conformance

1 In this document, “shall” is to be interpreted as a requirement on a UPC implementation; conversely, “shall not” is to be interpreted as a prohibition.

2 If a “shall” or “shall not” requirement of a constraint is violated, the behavior will be undefined. Undefined behavior is indicated by “undefined behavior” or by the omission of any explicit definition of behavior from the UPC specification.

\(^3\)A collective operation need not provide any actual synchronization between threads, unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely upon such unspecified synchronization for correctness.
5 Environment

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Threads environment

1 A UPC program is translated under either a “static THREADS” environment or a “dynamic THREADS” environment. Under the static THREADS environment, the number of threads to be used in execution is indicated to the translator in an implementation-defined manner. If the actual execution environment differs from this number of threads, the behavior of the program is undefined.

5.1.2 Execution environment

1 This subsection provides the UPC parallel extensions of [ISO/IEC00: Sec. 5.1.2]

2 A UPC program consists of a set of threads which may declare, create, and access both shared and private objects. Accesses to these objects are defined as either local or shared, based on the type of the access. Each thread’s local accesses behave independently and exactly as described in [ISO/IEC00]. All shared accesses behave as described herein.

3 There is an implicit upc_barrier at program startup and termination. Except as explicitly specified by upc_barrier operations or by certain library functions (all of which are explicitly documented), there are no other barrier synchronization guarantees among the threads.

Forward references: upc_barrier (6.5.1).

5.1.2.1 Program startup

1 In the execution environment of a UPC program, derived from the hosted environment as defined in the C Standard [ISO/IEC00], each thread calls the
UPC program’s main() function\(^4\).

5.1.2.2 Program termination

1 A program is terminated by the termination of all the threads\(^5\) or a call to the function upc_global_exit().

2 Thread termination follows the C Standard definition of program termination in [ISO/IEC00: Sec. 5.1.2.2.3]. A thread is terminated by reaching the } that terminates the main function, by a call to the exit function, or by a return from the initial main. Note that thread termination does not imply the completion of all I/O and that shared data allocated by a thread either statically or dynamically shall not be freed before UPC program termination.

Forward references: upc_global_exit (7.2.1).

5.1.2.3 Program execution

1 Shared accesses shall be either strict or relaxed. The behavior of a relaxed shared reads and writes is the same as in the C Standard execution model as applied to an individual thread. This implies that translators are free to reorder and/or ignore accesses (including shared accesses) as long as the restrictions found in [ISO/IEC00: Sec. 5.1.2.3] are observed.

2 A further restriction applies to strict shared reads and writes. For each such access, the restrictions found in [ISO/IEC00: Sec. 5.1.2.3] must be observed with respect to all threads if that access is eliminated (or reordered with respect to any other shared access performed by that thread).

3 Equally, the behavior of strict shared reads and writes can be defined as follows. Label each shared access S(i,j) or R(i,j), where S represents a strict shared read or write, R represents a relaxed shared read or write, i is the thread number making the access, j is an integer which monotonically increases as the evaluation of the program (in the abstract machine) proceeds from startup through termination. The “abstract order” is a partial ordering of all accesses by all threads such that an access x(a,b) occurs before y(c,d)

\(^4\)e.g., in the program main(){ printf("hello"); } , each thread prints hello.

\(^5\)A barrier is automatically inserted at thread termination.
in the ordering if \(a==c\) and \(b < d\). The “actual order(k)” for thread k is another partial order in which \(x(a,b)\) occurs before \(y(c,d)\) if thread k observes the x access before it observes the y access. A thread observes all accesses present in the abstract order which affect either the data written to files by it or its input and output dynamics as described in [ISO/IEC00: Sect 5.1.2.3]. The least requirements on a conforming implementation are that:

- \(x(a,b)\) must “occur before” \(y(c,d)\) in actual order(e) if \(a == e\) and \(a == e\) and \(b < d\)
- \(x(a,b)\) must “occur before” \(y(c,d)\) in actual order(e) if \(a == c\) and \(b < d\) and ((\(x == S\)) or (\(y == S\))), for all e

UNLESS such a restriction has no effect on either the data written into files at program termination OR the input and output dynamics requirements described in [ISO/IEC00: Sec. 5.1.2.3].

6 Language

6.1 Notations

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by italic type, and literal words and character set members (terminals) by bold type. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words “one of”. An optional symbol is indicated by the subscript “opt”, so that

\[
\{ \text{expression}_{\text{opt}} \}
\]

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.
6.2 Predefined identifiers

1 This subsection provides the UPC parallel extensions of section 6.4.2.2 in [ISO/IEC00].

6.2.1 THREADS

1 THREADS is a value of type int; it specifies the number of threads and has the same value on every thread. Under the static THREADS translation environment, THREADS is an integer constant suitable for use in #if preprocessing directives.

6.2.2 MYTHREAD

1 MYTHREAD is a value of type int; it specifies the unique thread index. The range of possible values is \(0..\text{THREADS}-1\)\(^6\).

6.2.3 UPC_MAX_BLOCK_SIZE

1 UPC_MAX_BLOCK_SIZE is a predefined integer constant value. It indicates the maximum value\(^7\) allowed in a layout qualifier for shared data. It shall be suitable for use in #if preprocessing directives.

6.3 Expressions

1 This subsection provides the UPC parallel extensions of section 6.5 in [ISO/IEC00].

6.3.1 The upc.localsizeof operator

\[
\text{upc.localsizeof} \quad \text{unary-expression} \\
\text{upc.localsizeof ( type-name )}
\]

\(^6\)e.g., the program main(){ printf("%d ",MYTHREAD); } , prints the numbers 0 thru THREADS-1, in some order.

\(^7\) e.g., shared [UPC_MAX_BLOCK_SIZE+1] char x[UPC_MAX_BLOCK_SIZE+1] and shared [*] char x[(UPC_MAX_BLOCK_SIZE+1)*THREADS] are compile errors.
Constraints

1 The `upc_localsizes` operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the `sizeof` operator [ISO/IEC00 Section 6.5.3.4] also apply to this operator.

Semantics

1 The `upclocalsizes` operator returns the size, in bytes, of the local portion of its operand, which may be a shared object or a shared-qualified type. It returns the same value on all threads; the value is the maximum of the size allocated to objects with affinity to any single thread. The result of `upc_localsizes` is a compile-time constant.

2 The type of the result is `size_t`.

3 If the the operand is an expression, that expression is not evaluated.

6.3.2 The `upc_blocksizeof` operator

```c
upc_blocksizeof unary-expression
upc_blocksizeof ( type-name )
```

Constraints

1 The `upc_blocksizeof` operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the `sizeof` operator [ISO/IEC00 Section 6.5.3.4] also apply to this operator.

Semantics

1 The `upc_blocksizeof` operator returns the block size of the operand, which may be a shared object or a shared-qualified type. The block size is the value specified in the layout qualifier of the type declaration. If there is no layout qualifier, the block size is 1. The result of `upc_blocksizeof` is a compile-time constant.

2 If the operator of `upc_blocksizeof` has indefinite block size, the value of `upc_blocksizeof` is 0.

3 The type of the result is `size_t`.

4 If the the operand is an expression, that expression is not evaluated.

Forward references: indefinite block size (6.4.2).
6.3.3 The upc_elemsizeof operator

\begin{verbatim}
upc_elemsizeof unary-expression
upc_elemsizeof ( type-name )
\end{verbatim}

Constraints

1 The \texttt{upc\_elemsizeof} operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the \texttt{sizeof} operator [ISO/IEC00 Section 6.5.3.4] also apply to this operator.

Semantics

1 The \texttt{upc\_elemsizeof} operator returns the size, in bytes, of the highest-level (leftmost) type that is not an array. For non-array objects, \texttt{upc\_elemsizeof} returns the same value as \texttt{sizeof}. The result of \texttt{upc\_elemsizeof} is a compile-time constant.

2 The type of the result is \texttt{size\_t}.

3 If the operand is an expression, that expression is not evaluated.

6.3.4 Pointer-to-shared arithmetic

1 When an expression that has integer type is added to or subtracted from a pointer-to-shared, the result has the type of the pointer-to-shared operand. If the pointer-to-shared operand points to an element of a shared array object, and the shared array is large enough, the result points to an element of the shared array. If the shared array is declared with indefinite block size, the result of the pointer-to-shared arithmetic is identical to that described for normal C pointers in [ISO/IEC00 sec. 6.5.6], except that the thread of the new pointer shall be the same as that of the original pointer and the phase component is defined to always be zero. If the shared array has a definite block size, then the following example describes pointer arithmetic:

\begin{verbatim}
shared [B] int *p, *p1; /* B a positive integer */
int i;

p1 = p + i;
\end{verbatim}
After this assignment the following equations must hold in any UPC implementation. In each case the \texttt{div} operator indicates integer division rounding towards negative infinity and the \texttt{mod} operator returns the nonnegative remainder:

\begin{align*}
\text{upc\_phaseof}(p1) &== (\text{upc\_phaseof}(p) + i) \mod B \\
\text{upc\_threadof}(p1) &== (\text{upc\_threadof}(p) + (\text{upc\_phaseof}(p) + i) \div B) \mod \text{THREADS}
\end{align*}

In addition, the correspondence between shared and local addresses and arithmetic is defined using the following constructs:

\begin{verbatim}
T *P1, *P2;
shared T *S1, *S2;

P1 = (T*) S1; /* legal if S1 has affinity to MYTHREAD */
P2 = (T*) S2; /* legal if S2 has affinity to MYTHREAD */
\end{verbatim}

For all \texttt{S1} and \texttt{S2} that point to two distinct elements of the same shared array object which have affinity to the same thread:

- \texttt{S1} and \texttt{P1} shall point to the same object.
- \texttt{S2} and \texttt{P2} shall point to the same object.
- The expression \((\text{upc\_addrfield}(S2) - \text{upc\_addrfield}(S1))\) shall evaluate to the same value as \(((P2 - P1) \times \text{sizeof}(T))\).
- If \texttt{S1} < \texttt{S2} then \texttt{upc\_addrfield}(S1) shall be < \texttt{upc\_addrfield}(S2) otherwise \texttt{upc\_addrfield}(S1) shall be > \texttt{upc\_addrfield}(S2)

Two compatible pointers-to-shared which point to the same object (i.e. having the same address and thread components) shall compare as equal according to \texttt{==} and \texttt{!=}, regardless of whether the phase components match.

\textbf{Forward references:} \texttt{upc\_threadof (7.2.3.1)}, \texttt{upc\_phaseof (7.2.3.2)}, \texttt{upc\_addrfield (7.2.3.4)}.

\textsuperscript{8}The C \texttt{"\%"} and \texttt{"/"} operators do not have the necessary properties
6.3.5 Cast and assignment expressions

Constraints

1 A shared type qualifier shall not appear in a type cast where the corresponding pointer component of the type of the expression being cast is not shared-qualified. An exception is made when the constant expression 0 is cast, the result is called the \textit{null pointer-to-shared}.\footnote{I.e., pointers-to-local cannot be cast to pointers-to-shared.} \footnote{[ISO/IEC00] sec 6.3.2.3 and 6.5.16.1 imply that an implicit cast is allowed for zero and that all null pointers-to-shared compare equal.}

Semantics

1 The casting or assignment from one pointer-to-shared to another in which either the type size or block size differs results in a pointer with a zero phase, unless one of the types is a qualified or unqualified version of \texttt{shared \texttt{void}*}, the generic pointer-to-shared.\footnote{The operators \texttt{upc\_threadof} and \texttt{upc\_phaseof} evaluate to zero for all null pointers-to-shared.}

2 If a generic pointer-to-shared is cast to a non-generic pointer-to-shared type with indefinite block size or with block size of one, the result is a pointer with a phase of zero. Otherwise, if the phase of the former pointer value is not within the range of possible phases of the latter pointer type, the result is undefined.

3 If a non-null pointer-to-shared is cast to a pointer-to-local and the affinity of the pointed-to shared object is not to the current thread, the result is undefined.

4 If a null pointer-to-shared is cast to a pointer-to-local, the result is a null pointer.

5 Shared objects with affinity to a given thread can be accessed by either pointers-to-shared or pointers-to-local of that thread.

6 \textbf{EXAMPLE 1:}

```c
int i, *p;
shared int *q;
q = (shared int *)p; /* is not allowed */
```
if (upc_threadof(q) == MYTHREAD)
  p = (int *) q; /* is allowed */

6.3.6 Address operators

Semantics
1 When the unary & is applied to a shared structure element of type T, the result has type shared [] T *.

6.4 Declarations

1 UPC extends the declaration ability of C to allow shared types, shared data layout across threads, and ordering constraint specifications.

Constraints
1 The declaration specifiers in a given declaration shall not include, either directly or through one or more typedefs, both strict and relaxed.
2 The declaration specifiers in a given declaration shall not specify more than one block size, either directly or indirectly through one or more typedefs.

Syntax
1 The following is the declaration definition as per [ISO/IEC00] section 6.7, repeated here for self-containment and clarity of the subsequent UPC extension specifications.
2 declaration:
   declaration-specifiers init-declarator-listopt ;
3 declaration-specifiers:
   storage-class-specifier declaration-specifiersopt
   type-specifier declaration-specifiersopt
   type-qualifier declaration-specifiersopt
   function-specifier declaration-specifiersopt
4 init-declarator-list:
   init-declarator
init-declarator-list, init-declarator

5 init-declarator:
   declarator
   declarator = initializer

Forward references: strict and relaxed type qualifiers (6.4.2).

6.4.1 Type qualifiers

1 This subsection provides the UPC parallel extensions of section 6.7.3 in [ISO/IEC00].

Syntax

1 type-qualifier:
   const
   restrict
   volatile
   shared-type-qualifier
   reference-type-qualifier

6.4.2 The shared and reference type qualifiers

Syntax

1 shared-type-qualifier:
   shared layout-qualifier_{opt}

2 reference-type-qualifier:
   relaxed
   strict

3 layout-qualifier:
   [constant-expression_{opt}]
   [ * ]

Constraints

18
A reference type qualifier shall appear in a qualifier list only when the list also contains a shared type qualifier.

A shared type qualifier can appear anywhere a type qualifier can appear except that it shall not appear in the specifier qualifier list of a structure declaration unless it qualifies a pointer type.

A layout qualifier of \([*]\) shall not appear in the declaration specifiers of a pointer.

A layout qualifier shall not appear in the type qualifiers for a pointer to void type.

**Semantics**

1. Shared accesses shall be either strict or relaxed. Strict and relaxed shared accesses behave as described in section 5.1.2.3 of this document.

2. An access shall be determined to be strict or relaxed as follows. If the referenced type is strict-qualified or relaxed-qualified, the access shall be strict or relaxed, respectively. Otherwise the access shall be determined to be strict or relaxed by the UPC pragma rules, as described in section 6.6.1 of this document.

3. The layout qualifier dictates the blocking factor for the type being shared qualified. This factor is the nonnegative number of consecutive elements (when evaluating pointer-to-shared arithmetic and array declarations) which have affinity to the same thread. If the optional constant expression is 0 or is not specified, all objects have affinity to the same thread. If there is no layout qualifier, the blocking factor has the default value (1). The blocking factor is also referred to as the block size.

4. A layout qualifier indicating that all array elements have affinity to the same thread is said to specify indefinite block size.

5. The block size is a part of the type compatibility\(^{14}\)

6. The generic pointer-to-shared is assignment compatible with any pointer-to-shared type, regardless of block size.

7. If the layout qualifier is of the form \(\[\,*\]\), the shared object is distributed as if it had a block size of

\(^{14}\)This is a powerful statement which allows, for example, that in an implementation `sizeof(shared int *)` may differ from `sizeof (shared [10] int *)` and if T and S are pointer-to-shared types with different block sizes, then T* and S* cannot be aliases.
where ‘a’ is the array being distributed.

8 EXAMPLE 1: declaration of a shared scalar

    strict shared int y;

strict shared is the type qualifier.

9 EXAMPLE 2: automatic storage duration

    void foo (void) {
      shared int x; /* a shared automatic variable is not allowed */
      shared int* y; /* a pointer-to-shared is allowed */
      int * shared z; /* a shared automatic variable is not allowed*/
      ... }

10 EXAMPLE 3: inside a structure

    struct foo {
      shared int x; /* this is not allowed */
      shared int* y; /* a pointer-to-shared is allowed */
    };

Forward references: shared array (6.4.3.1)

6.4.3 Declarators

Syntax

1 The following is the declarator definition as per [ISO/IEC00] section 6.7.5, repeated here for self-containment and clarity of the subsequent UPC extension specifications.

2 declarator:

    pointer_opt direct-declarator

3 direct-declarator:
identifier
( declarator )
direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]
direct-declarator [ static type-qualifier-list_opt assignment-expression ]
direct-declarator [ type-qualifier-list static assignment-expression ]
direct-declarator [ type-qualifier-list_opt * ]
direct-declarator ( parameter-type-list )
direct-declarator ( identifier-list_opt )

4  pointer:
   * type-qualifier-list_opt
   * type-qualifier-list_opt pointer

5  type-qualifier-list:
   type-qualifier
   type-qualifier-list type-qualifier

Constraints
1  No type qualifier list shall specify more than one block size, either directly or indirectly through one or more typedefs.\(^{15}\)
2  No type qualifier list shall include both *strict* and *relaxed* either directly or indirectly through one or more typedefs.
3  *shared* shall not appear in a declarator which has automatic storage duration, unless it qualifies a pointer type.

Semantics
1  All static non-array shared objects have affinity with thread zero.
2  Only pointer type members of a structure or union may be shared-qualified.\(^{16}\)

\(^{15}\)While layout qualifiers are most often seen in array or pointer declarators, they are legal in all declarators. For example, *shared* [3] int y is a legal declarator.

\(^{16}\)E.g., struct S1 { shared char * p1; }; is legal, while struct S2 { char * shared p2; }; is not.
6.4.3.1 Array declarators

This subsection provides the UPC parallel extensions of section 6.7.5.2 in [ISO/IEC00].

Constraints

When a UPC program is translated in the “dynamic THREADS” environment and the type of the array is shared-qualified but not indefinite layout-qualified, the THREADS lvalue shall occur exactly once in one dimension of the array declarator (including through typedefs). Further, in such cases, the THREADS lvalue shall only occur either alone or when multiplied by a constant expression.

Semantics

Elements of shared arrays are distributed in a round robin fashion, by chunks of block-size elements, such that the i-th element has affinity with thread (floor (i/block_size) mod THREADS).

In an array declaration, the type qualifier applies to the elements.

For any shared array, a, upc_phaseof (&a) is zero.

EXAMPLE 1: declarations legal in either static or dynamic translation environments:

```c
shared int x [10*THREADS];
shared [] int x [10];
```

EXAMPLE 2: declarations legal only in static translation environment:

```c
shared int x [10+THREADS];
shared [] int x [THREADS];
shared int x [10];
```

EXAMPLE 3: declaration of a shared array

```c
shared [3] int x [10];
```
shared [3] is the type qualifier of an array, x, of 10 integers. [3] is the layout qualifier.

7 EXAMPLE 4:

```c
typedef int S[10];
shared [3] S T[3*THREADS];
```

shared [3] applies to the underlying type of T, which is int, regardless of the typedef. The array is blocked as if it were declared:

```c
shared [3] int T[3*THREADS][10];
```

8 EXAMPLE 5:

```c
shared [] double D[100];
```

All elements of the array D have affinity to thread 0. No THREADS dimension is allowed in the declaration of D.

```c
shared [] long *p;
x = p[i];
```

All elements accessed by subscripting or otherwise dereferencing p have affinity to the same thread. That thread is determined by the assignment which sets p.

6.5 Statements and blocks

1 This subsection provides the UPC parallel extensions of section 6.8 in [ISO/IEC00].

Syntax

1 statement:

```plaintext
labeled-statement
compound-statement
expression-statement
```
6.5.1 Barrier statements

Syntax
1  synchronization-statement:
   upc_notify expression opt ;
   upc_wait expression opt ;
   upc_barrier expression opt ;
   upc_fence ;

Constraints
1  expression shall be an integer expression.
2  Each thread shall execute an alternating sequence of upc_notify and upc_wait statements, starting with a upc_notify and ending with a upc_wait statement. After a thread executes upc_notify the next collective operation it executes must be a upc_wait.\(^{17}\) A synchronization phase consists of the execution of all statements between the completion of one upc_wait and the start of the next.

Semantics
1  A upc_wait statement completes, and the thread enters the next synchronization phase, only after all threads have completed the upc_notify statement in the current synchronization phase.\(^{18}\) upc_wait and upc_notify are collective operations.

\(^{17}\)This effectively prohibits issuing any collective operations between a upc_notify and a upc_wait.

\(^{18}\)Therefore, all threads are entering the same synchronization phase as they complete the upc_wait statement.
The **upc_fence** statement is equivalent to a *null* strict access. This insures that all shared accesses issued before the fence are complete before any after it are issued.\(^{19}\)

A null strict access is implied before a **upc_notify** statement and after a **upc_wait** statement.\(^{20}\)

The **upc_wait** statement will generate a runtime error if the value of its expression does not equal the value of the expression by the **upc_notify** statement for the current synchronization phase. No error will be generated if either statement does not have an expression.

The **upc_wait** statement will generate a runtime error if the value of its expression differs from any expression on the **upc_wait** and **upc_notify** statements issued by any thread in the current synchronization phase. No error will be generated from a “difference” involving a statement for which no expression is given.

The **upc_barrier** statement is equivalent to the compound statement\(^ {21}\):

\[
\{ \text{upc.notify barrier_value; upc.wait barrier_value; } \}
\]

The barrier operations at thread startup and termination have a value of *expression* which is not in the range of user expressible values.

**EXAMPLE 1:** The following will result in a runtime error:

\[
\{ \text{upc.notify; upc.barrier; upc.wait; } \}
\]

as it is equivalent to

\[
\{ \text{upc.notify; upc.notify; upc.wait; upc.wait; } \}
\]

\(^{19}\)One implementation of **upc_fence** may be achieved by a null strict access: { static shared strict int x; x = x; }

\(^{20}\)This implies that shared accesses executed after the **upc_notify** and before the **upc_wait** may occur in either the synchronization phase containing the **upc_notify** or the next on different threads.

\(^{21}\)This equivalence is explicit with respect to matching expressions in semantic 6 and collective status in semantic 1.
6.5.2 Iteration statements

This subsection provides the UPC parallel extensions of section 6.8.5 in [ISO/IEC00].

Syntax

iteration-statement:

while ( expression ) statement
do statement while ( expression ) ;
for ( expression_opt; expression_opt; expression_opt ) statement
for ( declaration-expression_opt; expression_opt ) statement
upc forall ( expression_opt; expression_opt; expression_opt; affinity_opt )

statement

affinity:

expression_opt
continue

Constraints:

The expression for affinity shall have pointer-to-shared type or integer type.

Semantics:

upc forall is a collective operation in which, for each execution of the loop body, the controlling expression and affinity expression are single-valued.\(^{22}\)

The affinity field specifies the executions of the loop body which are to be performed by a thread.

When affinity is of pointer-to-shared type, the loop body of the upc forall statement is executed for each iteration in which the value of MYTHREAD equals the value of upc_threadof(affinity). Each iteration of the loop body is executed by precisely one thread.

When affinity is an integer expression, the loop body of the upc forall statement is executed for each iteration in which the value of MYTHREAD equals the value affinity mod THREADS.

\(^{22}\)Note that single-valued implies that all thread agree on the total number of iterations, their sequence, and which threads execute each iteration.
When affinity is continue or not specified, each loop body of the upc_forall statement is performed by every thread and semantic 1 does not apply.

If the loop body of a upc_forall statement contains one or more upc_forall statements, either directly or through one or more function calls, the construct is called a “nested upc_forall” statement. In a “nested upc_forall”, the outermost upc_forall statement that has an affinity expression which is not continue is called the “controlling upc_forall” statement. All upc_forall statements which are not “controlling” in a “nested upc_forall” behave as if their affinity expressions were continue.

If the execution of any loop body of a upc_forall statement produces a side-effect which affects the execution of another loop body of the same upc_forall statement which is executed by a different thread\(^23\), the behavior is undefined.

If any thread terminates or executes a upc_barrier, upc_notify, or upc_wait statement within the dynamic scope of a upc_forall statement, the result is undefined. If any thread terminates a upc_forall statement using a break, goto, or return statement, or the longjmp function, the result is undefined. If any thread enters the body of a upc_forall statement using a goto statement, the result is undefined.\(^24\)

**EXAMPLE 1: Nested upc_forall:**

```c
main () {
    int i, j, k;
    shared float *a, *b, *c;

    upc_forall(i=0; i<N; i++; continue)
        upc_forall(j=0; j<N; j++; &a[j])
            upc_forall (k=0; k<N; k++; &b[k])
            a[j] = b[k] * c[i];
}
```

This example executes all iterations of the “i” and “k” loops on every thread,

\(^23\)This semantic implies that side effects on the same thread have defined behavior, just like in the for statement.

\(^24\)The continue statement behaves as defined in [ISO/IEC 00: Section 6.8.6.2]; equivalent to a goto the end of the loop body.
and executes iterations of the “j” loop on those threads where \texttt{upc\_threadof} (&a[j]) equals the value of \texttt{MYTHREAD}.

6.6 Preprocessing directives

1 This subsection provides the UPC parallel extensions of section 6.10 in [ISO/IEC00].

6.6.1 UPC pragmas

Semantics

1 If the preprocessing token \texttt{upc} immediately follows the \texttt{pragma}, then no macro replacement is performed and the directive shall have one of the following forms:

\verbatim
#pragma upc strict

#pragma upc relaxed
\endverbatim

2 These pragmas affect the strict or relaxed categorization of shared accesses where the referenced type is neither strict-qualified nor relaxed-qualified. Such accesses shall be strict if a strict pragma is in effect, or relaxed if a relaxed pragma is in effect.

3 Shared accesses which are not categorized by either referenced type or by these pragmas behave in an implementation defined manner in which either all such accesses are strict or all are relaxed. Users wishing portable programs are strongly encouraged to categorize all shared accesses either by using type qualifiers, these directives, or by including \texttt{upc\_strict.h} or \texttt{upc\_relaxed.h}.

4 The pragmas shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When they are outside external declarations, they apply until another such pragma or the end of the translation unit. When inside a compound statement, they apply until the end of the compound statement; at the end of the compound statement the state of the pragmas is restored to that preceding the compound statement. If these pragmas are used in any other context, their behavior is undefined.
6.6.2 Predefined macro names

1 The following macro name shall be defined by the implementation:

    __UPC__   The integer constant 1, indicating a conforming implementation.
    __UPC_VERSION__   The integer constant 200311L.

2 The following macro names are conditionally defined by the implementation:

    __UPC_DYNAMIC_THREADS__.   The integer constant 1 in the dynamic THREADS
                              translation environment, otherwise undefined.
    __UPC_STATIC_THREADS__.   The integer constant 1 in the static THREADS
                              translation environment, otherwise undefined.

7 Library

7.1 Standard headers

1 This subsection provides the UPC parallel extensions of section 7.1.2 in
   [ISO/IEC00].

2 The standard headers are

    <upc_strict.h>
    <upc_relaxed.h>
    <upc.h>

3 upc_strict.h shall contain at least:

    #pragma upc strict
    #include <upc.h>

4 upc_relaxed.h shall contain at least:

    #pragma upc relaxed
    #include <upc.h>
7.2 UPC utilities <upc.h>

This subsection provides the UPC parallel extensions of section 7.20 in [ISO/IEC00]. All of the characteristics of library functions described in section 7.1.4 in [ISO/IEC00] apply to these as well.

7.2.1 Termination of all threads

Synopsis

```c
#include <upc.h>

void upc_global_exit(int status);
```

Description

`upc_global_exit()` flushes all I/O, releases all storage, and terminates the execution for all active threads.

7.2.2 Shared memory allocation functions

The UPC memory allocation functions return, if successful, a pointer-to-shared which is suitably aligned so that it may be assigned to a pointer-to-shared of any type. The pointer has zero phase and points to the start of the allocated space. If the space cannot be allocated, a null pointer-to-shared is returned.

7.2.2.1 The `upc_global_alloc` function

Synopsis

```c
#include <upc.h>

shared void *upc_global_alloc(size_t nblocks, size_t nbytes);
```

nblocks : number of blocks
nbytes : block size
Description
1 The `upc_global_alloc` allocates shared space compatible with the declaration:

   `shared [nbytes] char[nblocks * nbytes].`

2 The `upc_global_alloc` function is not a collective function. If called by multiple threads, all threads which make the call get different allocations. If `nblocks*nbytes` is zero, the result is a null pointer-to-shared.

7.2.2.2 The `upc_all Alloc` function

Synopsis
1

   ```
   #include <upc.h>

   shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
   nblocks : number of blocks
   nbytes : block size
   ```

Description
1 `upc_all_alloc` is a collective function with single-valued arguments.
2 The `upc_all_alloc` function allocates shared space compatible with the following declaration:

   `shared [nbytes] char[nblocks * nbytes].`

3 The `upc_all_alloc` function returns the same pointer value on all threads. If `nblocks*nbytes` is zero, the result is a null pointer-to-shared.
4 The dynamic lifetime of an allocated object extends from the time any thread completes the call to `upc_all_alloc` until any thread has deallocated the object.
7.2.2.3 The upc_alloc function

Synopsis
1

```c
#include <upc.h>

shared void *upc_alloc(size_t nbytes);
  nbytes : total number of bytes to allocate
```

Description
1 The upc_alloc function allocates shared space of at least nbytes bytes with affinity to the calling thread.

2 upc_alloc is similar to malloc() except that it returns a pointer-to-shared value. It is not a collective function. If nbytes is zero, the result is a null pointer-to-shared.

7.2.2.4 The upc_local_alloc function deprecated

Synopsis
1

```c
#include <upc.h>

shared void *upc_local_alloc(size_t nblocks, size_t nbytes);
  nblocks : number of blocks
  nbytes : block size
```

Description
1 The upc_local_alloc function is deprecated and should not be used. UPC programs should use the upc_alloc function instead. Support may be removed in future versions of this specification.

2 The upc_local_alloc function allocates shared space of at least nblocks * nbytes bytes with affinity to the calling thread. If nblocks*nbytes is zero, the result is a null pointer-to-shared.

3 upc_local_alloc is similar to malloc() except that it returns a pointer-to-shared value. It is not a collective function.
7.2.2.5 The upc_free function

Synopsis
1  #include <upc.h>

    void upc_free(shared void *ptr);

Description
1  The upc_free function frees the dynamically allocated shared storage 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_alloc, upc_global_alloc, upc_all_alloc, or upc_local_alloc, function, or if the space has been deallocated by a previous call, by any thread,\textsuperscript{25} to upc_free, the behavior is undefined.

7.2.3 Pointer-to-shared manipulation functions

7.2.3.1 The upc_threadof function

Synopsis
1  #include <upc.h>

    size_t upc_threadof(shared void *ptr);

Description
1  The upc_threadof function returns the index of the thread that has affinity to the shared object pointed to by ptr.

7.2.3.2 The upc_phaseof function

\textbf{Synopsis}\textsuperscript{\textsuperscript{29}}
1  i.e., only one thread may call upc_free for each allocation
#include <upc.h>

size_t upc_PHASEOF(shared void *ptr);

Description
1 The upc_PHASEOF function returns the phase component of the pointer-to-shared argument.

7.2.3.3 The upc_RESETPHASE function

Synopsis
1 #include <upc.h>

shared void *upc_RESETPHASE(shared void *ptr);

Description
1 The upc_RESETPHASE function returns a pointer-to-shared which is identical to its input except that it has zero phase.

7.2.3.4 The upc_ADDRFIELD function

Synopsis
1 #include <upc.h>

size_t upc_ADDRFIELD(shared void *ptr);

Description
1 The upc_ADDRFIELD function returns an implementation-defined value reflecting the “local address” of the object pointed to by the pointer-to-shared argument.
7.2.3.5 The upc_affinitysize function

Synopsis

```c
#include <upc.h>

size_t upc_affinitysize(size_t totalsize, size_t nbytes, size_t threadid);
```

totalsize: the total size of the allocation in bytes
nbytes: the number of bytes in a block
threadid: the thread whose affinitysize is to be evaluated

Description

`upc_affinitysize` is a convenience function which calculates the exact size of the local portion of the data in a shared object with affinity to `threadid`.

In the case of a dynamically allocated shared object, the `totalsize` argument shall be `nbytes*nblocks` and the `nbytes` argument shall be `nbytes`, where `nblocks` and `nbytes` are exactly as passed to `upc_global_alloc` or `upc_all_alloc` when the object was allocated.

In the case of a statically allocated shared object with declaration:

```c
shared [b] t d[s];
```

the `totalsize` argument shall be `s * sizeof (t)` and the `nbytes` argument shall be `b * sizeof (t)`. If block size is unspecified, `nbytes` shall be 1. If the block size is indefinite, `nbytes` shall be 0.

`threadid` shall be a value in `0..(THREADS-1)`.

7.2.4 Lock functions

7.2.4.1 Type

The type declared is

```c
upc_lock_t
```

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

Synopsis
1    #include <upc.h>

    upc_lock_t *upc_global_lock_alloc(void);

Description
1    The upc_global_lock_alloc function dynamically allocates a lock and re-
turns a pointer to it. The lock is created in an unlocked state.
2    The upc_global_lock_alloc function is not a collective function. If called
by multiple threads, all threads which make the call get different allocations.

7.2.4.3 The upc_all_lock_alloc function

Synopsis
1    #include <upc.h>

    upc_lock_t *upc_all_lock_alloc(void);

Description
1    The upc_all_lock_alloc function dynamically allocates a lock and returns
a pointer to it. The lock is created in an unlocked state.
2    The upc_all_lock_alloc is a collective function. The return value on every
thread points to the same lock object.

7.2.4.4 The upc_lock_free function

Synopsis
1    #include <upc.h>

    void upc_lock_free(upc_lock_t *ptr);
Description
1 The **upc_lock_free** function frees all resources associated with the dynamically allocated **upc_lock_t** 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_global_lock_alloc** or **upc_all_lock_alloc** function, or if the lock has been deallocated by a previous call to **upc_lock_free**, the behavior is undefined.
2 **upc_lock_free** succeeds regardless of whether the referenced lock is currently unlocked or currently locked (by any thread).
3 Any subsequent calls to locking functions from any threads using **ptr** have undefined effects.

7.2.4.5 The **upc_lock** function

Synopsis
1
2
3

```c
#include <upc.h>

void upc_lock(upc_lock_t *ptr);
```

Description
1 The **upc_lock** function locks a shared variable, of type **upc_lock_t**, pointed to by the pointer given as argument.
2 If the lock is not used by another thread, then the thread making the call gets the lock and the function returns. Otherwise, the function keeps trying to get access to the lock.
3 A null strict access is implied after a call to **upc_lock**.
4 If the calling thread is already holding the lock pointed to by **ptr** (i.e., it has previously locked it using **upc_lock()** or **upc_lock_attempt()**, but not unlocked it), the result is undefined.

7.2.4.6 The **upc_lock_attempt** function

Synopsis
1
2
3

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

int upc_lock_attempt(upc_lock_t *ptr);

Description

1 The upc_lock_attempt function tries to lock a shared variable, of type upc_lock_t, pointed to by the pointer given as argument.

2 If the lock is not used by another thread, then the thread making the call gets the lock and the function returns 1. Otherwise, the function returns 0.

3 A null strict access is implied after a call to upc_lock_attempt() that returns 1.

4 If the calling thread is already holding the lock pointed to by ptr (i.e., it has previously locked it using upc_lock() or upc_lock_attempt(), but not unlocked it), the result is undefined.

7.2.4.7 The upc_unlock function

Synopsis

1 #include <upc.h>

   void upc_unlock(upc_lock_t *ptr);

Description

1 The upc_unlock function frees the lock and does not return any value.

2 A null strict access is implied before a call to upc_unlock().

7.2.5 Shared string handling functions

7.2.5.1 The upc_memcpy function

Synopsis

1 #include <upc.h>

   void upc_memcpy(shared void *dst, shared const void *src, size_t n);
Description
1 The `upc_memcpy` function copies `n` characters from a shared object having affinity with one thread to a shared object having affinity with the same or another thread. If copying takes place between objects that overlap, the behavior is undefined.

2 The `upc_memcpy` function treats the `dst` and `src` pointers as if they had type:

   shared [] char[n]

   The effect is equivalent to copying the entire contents from one shared array object with this type (the `src` array) to another shared array object with this type (the `dst` array).

7.2.5.2 The `upc_memget` function

Synopsis
1

   #include <upc.h>

   void upc_memget(void *dst, shared const void *src, size_t n);

Description
1 The `upc_memget` function copies `n` characters from a shared object with affinity to any single thread to an object on the calling thread. If copying takes place between objects that overlap, the behavior is undefined.

2 The `upc_memget` function treats the `src` pointer as if it had type:

   shared [] char[n]

   The effect is equivalent to copying the entire contents from one shared array object with this type (the `src` array) to an array object (the `dst` array) declared with the type

   char[n]
7.2.5.3 The upc_memput function

Synopsis

```c
#include <upc.h>

void upc_memput(shared void *dst, const void *src, size_t n);
```

Description

1 The `upc_memput` function copies `n` characters from the an object on the calling thread to a shared object with affinity to any single thread. If copying takes place between objects that overlap, the behavior is undefined.

2 The `upc_memput` function is equivalent to copying the entire contents from an array object (the `src` array) declared with the type

```
char[n]
```

to a shared array object (the `dst` array) with the type

```
shared [] char[n]
```

7.2.5.4 The upc_memset function

Synopsis

```c
#include <upc.h>

void upc_memset(shared void *dst, int c, size_t n);
```

Description

1 The `upc_memset` function copies the value of `c`, converted to an `unsigned char`, to a shared object with affinity to any single thread. The number of bytes set is `n`.

2 The `upc_memset` function treats the `dst` pointer as if had type:

```
shared [] char[n]
```

The effect is equivalent to setting the entire contents of a shared array object with this type (the `dst` array) to the value `c`. 

40
References


A UPC versus C Standard section numbering

<table>
<thead>
<tr>
<th>UPC specifications subsection</th>
<th>C Standard specifications subsection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6.1</td>
<td>6.1</td>
</tr>
<tr>
<td>6.2</td>
<td>6.4.2.2</td>
</tr>
<tr>
<td>6.3</td>
<td>6.5</td>
</tr>
<tr>
<td>6.3.6</td>
<td>6.5.3.1</td>
</tr>
<tr>
<td>6.4</td>
<td>6.7</td>
</tr>
<tr>
<td>6.4.1</td>
<td>6.7.3</td>
</tr>
<tr>
<td>6.4.3</td>
<td>6.7.5</td>
</tr>
<tr>
<td>6.5</td>
<td>6.8</td>
</tr>
<tr>
<td>6.6</td>
<td>6.10</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7.1</td>
<td>7.1.2</td>
</tr>
</tbody>
</table>

Table A1. Mapping UPC subsections to C Standard specifications subsections