SC22/WG14 N403
X3J11/95-004

Technical Report

Numerical C Extensions

Committee X3J11

April 1995

Foreword

In 1983, committee X3J11 was formed to produce an ANSI standard for the C language. However, by 1986, it was clear that many good suggestions and proposals would not make it into that standard. As a result, for several years, I chaired ad-hoc evening sessions at X3J11 meetings, at which members discussed issues outside the scope of the emerging standard, as well as their existing and planned extensions.

In order to provide a more formal forum for some of these discussions, I formed the Numerical C Extensions Group (NCEG), announced its existence, and asked Tom MacDonald of Cray Research to host an organizational meeting in Minneapolis, Minnesota. This meeting took place May 10–11, 1989, and was attended by 30 people. The backgrounds of the attendees were diverse and included representatives from hardware vendors involved in supercomputing, digital signal processing, IEEE (as well as proprietary) floating-point, as well as users and academia. The companies represented included: Analog Devices, Apple, CDC, Convex, Cray, Digital Equipment Corporation, Hewlett-Packard, IBM, Information and Graphics Systems, Intermetrics, Lawrence Livermore Labs., Microsoft, Polaroid, Solborne, Supercomputer Systems, Thinking Machines, Unisys, University of Minnesota, and the US Army. Dennis Ritchie from AT&T also attended as did a number of independent consultants.

The main purpose of the initial meeting was to identify and prioritize the principal technical issues. Attendees then voted on each topic, indicating high or medium priority. The high priority votes were weighted twice as much as the medium, and the following list of priorities resulted:

Main Numerical Issues

Topic	*Priority*
Aliasing	29
Vectorization support	27
Complex arithmetic	27
Variable-size arrays	25
IEEE issues	24
Exceptions/`errno`	24
`float`/`long double` library	23
Parallel support	22
`<math.h>` improvements	21
Array syntax	19
Addition of math functions	17
Aggregate initializers	15
Inter-language issues	15
Wide accumulators	10
Math function precision	9
Non-zero-based arrays	8
Numerical representation	6
New data types	4
New operators	4
Function overloading	4

Another topic, "Arrays as first class objects" had a high priority (21) but after considerable debate was dropped from the list. It was agreed that its addition would likely cause great confusion among existing C programmers. (A new topic "Extended Integers" was added at a later meeting.)

The bulk of the agenda time was then given to the top ten topics, each getting 20–30 minutes. For each of these topics, various attendees volunteered to be primary and alternate coordinators. From that point on, meeting rules and procedures emulated those used by committee X3J11.

From the beginning, there was consensus that NCEG become affiliated with a recognized standards organization. The final proposal was that we become a working group within X3J11. This took place early in 1991, when NCEG became officially known as committee X3J11.1. Our charter was to produce a Technical Report (TR), a non-binding report on our findings and recommendations with respect to numerical C extensions. This document is that report.

In January, 1994, committee X3J11.1 was folded into its parent committee X3J11. Work on the TR was coming to a close and it was felt that progress would be helped by elevating subgroup X3J11.1's work to the full committee. At the December, 1994, meeting of X3J11, the final subgroup contributions were voted into the TR. The final report was produced in April, 1995, and forwarded to ANSI for processing.

During 1994, committee SC22/WG14 (ISO C) and X3J11 agreed to begin work on revising the ANSI/ISO C Standard. This project is informally known as C9X. At the December, 1994, meeting of WG14 and X3J11, X3J11

voted to propose seven of the eight components of this TR as part of the US contribution to C9X. The Data Parallel component was *not* proposed because it was felt that this field was rapidly evolving and that, at this time, a standard for these features was premature. *It is important to note that even though seven components of this TR will be considered for inclusion in C9X, their adoption in their existing or, indeed, any form, is by no means guaranteed.* The schedule for C9X calls for the technical specification to be completed by the end of 1996. At that time, we should know which parts of this TR have been accepted.

ANSI's rules do not require that a TR be subjected to the same rigorous processing as a standard. Specifically, there need be no public review. However, X3J11 made every effort to distribute the TR components to affected and interested parties, including official and de-facto standards groups, industry groups, and user groups. Many constructive comments and improvements were received from people outside X3J11 and incorporated into the TR.

The first standard for C was approved by ANSI in 1989 and was designated X3.159. The first ISO C Standard was adopted in 1990 and was designated ANSI/ISO 9899-1990. At that time, X3.159 was withdrawn and replaced by 9899-1990. Technically, the two documents are equivalent; however, at that time, the ISO rules for formatting and section numbering differed from those of ANSI. Many people who participated in the production of this report did not have a copy of 9899-1990 and used X3.159 instead. So, as a courtesy to people not having access to the ISO standard, many standard references in the TR cite both 9899 and X3.159.

Early in the process, it was determined that each subgroup should have control over its own TR component. As a result, the components were produced on a variety of text processing systems. As each component pretty much stands alone, the larger ones have their own table of contents and appendices; the largest one even has a detailed cross-reference index. In this report, each component is covered in a single chapter and the order of the chapters was determined arbitrarily, so the reader may read them in any order.

To obtain general information about this TR, contact me as indicated below. For information about specific components in this TR, contact the person identified at the beginning of the corresponding chapter.

Rex Jaeschke
Founder of NCEG and X3J11.1
Chair of X3J11
2051 Swans Neck Way
Reston, VA 22091, USA
+1 (703) 860-0091
rex@aussie.com

Table of Contents

1. Restricted Pointers
2. Arrays of Variable Length
3. Designated Initializers
4. Compound Literals
5. Floating-Point C Extensions
6. Complex C Extensions
7. Data Parallel C Extensions
8. Extended Integers for C

Chapter 1

Restricted Pointers

Contents

1 **Restricted Pointers** 1
- 1.1 Introduction . . . 1
- 1.2 Rationale . . . 1
  - 1.2.1 Aliasing . . . 1
  - 1.2.2 Library examples . . . 1
  - 1.2.3 Restricted pointers . . . 2
  - 1.2.4 Design choices . . . 2
- 1.3 Formal definition of restrict . . . 3
- 1.4 Examples . . . 4
  - 1.4.1 File scope restricted pointers . . . 4
  - 1.4.2 Function parameters . . . 4
  - 1.4.3 Block scope . . . 5
  - 1.4.4 Members of structures . . . 5
  - 1.4.5 Type definitions . . . 5
  - 1.4.6 Expressions based on restricted pointers . . . 6
  - 1.4.7 Assignments between restricted pointers . . . 7
  - 1.4.8 Assignments to unrestricted pointers . . . 7
  - 1.4.9 Ineffective uses of type qualifiers . . . 8
  - 1.4.10 Constraint violations . . . 8
- 1.5 Appendix: Overlapping objects . . . 9
- 1.6 Appendix: Discussion of design choices . . . 10
  - 1.6.1 Contiguity . . . 10
  - 1.6.2 Aliasing of unmodified objects . . . 10
  - 1.6.3 Linked lists . . . 11
- 1.7 Appendix: Array syntax and qualified parameters . . . 13
- 1.8 Appendix: Comparison with noalias . . . 15

Foreword

Aliasing through pointers was identified as the greatest single deterrent to using C as a numerical programming language. The possibility that two pointers may point to the same object can inhibit compiler optimizations that, on some systems, would improve performance by a factor of ten or more. The problem is fundamental, because the Standard specifies that array parameters to functions are adjusted to be pointers.

Although a number of approaches were considered, restricted pointers have several important advantages. They require only a very small extension to the language, so that it was easy to experiment with an implementation. They can be used effectively in a variety contexts, including at file scope, as function parameters, and as members of structures. They are very portable, because eliminating all occurrences of the `restrict` keyword from a program does not change its meaning. (This can be done without editing the text of the program by simply defining `restrict` to be a preprocessor macro with an empty replacement-list.)

Restricted pointers have now been implemented in C compilers from Cray Research, Inc., from Convex Corporation, and in a C/C++ compiler from the Edison Design Group.

Bill Homer
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121
homer@cray.com

People who made contributions to this document, and their affiliations at the time of their contributions, include:

Convex Corporation, Austin Curley
Convex Corporation, Bill Torkelson
Cray Research, Inc., Dave Becker
Cray Research, Inc., Steve Collins
Cray Research, Inc., John Dawson
Cray Research, Inc., Doug Jackson
Cray Research, Inc., Tom MacDonald
Cray Research, Inc., Mark Pagel
DEC Professional, Rex Jaeschke
Digital Equipment Corporation, Randy Meyers
Farance, Inc., Frank Farance
Hewlett Packard, John Kwan
IBM, Fred Tydeman
Keaton Consulting, David Keaton
Kuck & Associates Inc., Arch Robison
Lawrence Livermore National Laboratory, Linda Stanberry
Open Software Foundation, Mike Meissner
Plum Hall, Inc., Thomas Plum
SunPro, Bob Jervis
Thinking Machines Corporation, James L. Frankel
University of California at Irvine, Joe Hummel
Unix Systems Laboratories, David Prosser

Chapter 1

Restricted Pointers

1.1 Introduction

This chapter describes a proposal for an additional type qualifier, spelled `restrict`. Like the `volatile` type qualifier, `restrict` is intended to affect compiler optimizations (though in the opposite direction, by enabling rather inhibiting them). Like the `const` type qualifier, `restrict` has semantic rules to which a program must conform to avoid undefined behavior. The proposed changes to the text of the C Standard, which are relatively small and confined to two sections, are presented after the rationale.

1.2 Rationale

1.2.1 Aliasing

For many compiler optimizations, ranging from simply holding a value in a register to the parallel execution of a loop, it is necessary to determine whether two distinct lvalues designate distinct objects. If the objects are not distinct, the lvalues are said to be *aliases*. It is aliasing through pointers that presents the greatest difficulty, because there is often not enough information available within a single function, or even within a single compilation unit, to determine whether two pointers can point to the same object. Even when enough information is available, this analysis can require substantial time and space. For example, it could require an analysis of a whole program to determine the possible values of a pointer that is a function parameter.

1.2.2 Library examples

Consider how potential aliasing enters into implementations in C of two Standard C library functions. There are no restrictions on the use of `memmove`, and the implementation shown in Figure 1.1 follows the model described in the Standard by copying through a temporary array. (Other approaches are possible, but this one is straightforward and strictly conforming.) Since `memcpy` cannot be used for copying between overlapping arrays, its implementation can be a direct copy, as in Figure 1.2. (See the appendix on "Overlapping objects.")

Note that the restriction on `memcpy` is expressed only in its *Description* in the Standard, and cannot be expressed directly in its implementation in C. While this does allow the source-level optimization of eliminating the temporary used in `memmove`, it does not provide for compiler optimization of the resulting single loop. In many architectures, it is faster to copy bytes in blocks, rather than one at a time. The implementation of `memmove` uses `malloc` to obtain the temporary array, and this guarantees that the temporary is disjoint from the source and target arrays. From this a compiler can deduce that block copies may safely be used for both loops. The implementation of `memcpy`, on the other hand, provides the compiler no basis for ruling out the possibility that, for example, `s1` and `s2` point to

(CONTINUED W/ 81) 67

Foreword

Aliasing through pointers was identified as the greatest single deterrent to using C as a numerical programming language. The possibility that two pointers may point to the same object can inhibit compiler optimizations that, on some systems, would improve performance by a factor of ten or more. The problem is fundamental, because the Standard specifies that array parameters to functions are adjusted to be pointers.

Although a number of approaches were considered, restricted pointers have several important advantages. They require only a very small extension to the language, so that it was easy to experiment with an implementation. They can be used effectively in a variety contexts, including at file scope, as function parameters, and as members of structures. They are very portable, because eliminating all occurrences of the `restrict` keyword from a program does not change its meaning. (This can be done without editing the text of the program by simply defining `restrict` to be a preprocessor macro with an empty replacement-list.)

Restricted pointers have now been implemented in C compilers from Cray Research, Inc., from Convex Corporation, and in a C/C++ compiler from the Edison Design Group.

Bill Homer
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121
homer@cray.com

People who made contributions to this document, and their affiliations at the time of their contributions, include:

Convex Corporation, Austin Curley
Convex Corporation, Bill Torkelson
Cray Research, Inc., Dave Becker
Cray Research, Inc., Steve Collins
Cray Research, Inc., John Dawson
Cray Research, Inc., Doug Jackson
Cray Research, Inc., Tom MacDonald
Cray Research, Inc., Mark Pagel
DEC Professional, Rex Jaeschke
Digital Equipment Corporation, Randy Meyers
Farance, Inc., Frank Farance
Hewlett Packard, John Kwan
IBM, Fred Tydeman
Keaton Consulting, David Keaton
Kuck & Associates Inc., Arch Robison
Lawrence Livermore National Laboratory, Linda Stanberry
Open Software Foundation, Mike Meissner
Plum Hall, Inc., Thomas Plum
SunPro, Bob Jervis
Thinking Machines Corporation, James L. Frankel
University of California at Irvine, Joe Hummel
Unix Systems Laboratories, David Prosser

The definition given in section 1.3 favors the second view. It allows restricted pointers to be modifiable, to be members of structures and elements of arrays, and to be "strongly scoped," in the sense that a restricted pointer declared in a nested block makes a non-aliasing assertion only within that block.

1.3 Formal definition of restrict

What follows are proposed additions to the document *International Standard Programming Languages — C (ISO/IEC 9899:1990(E))*.

Keywords [ISO §6.1.1 (X3.159 §3.1.1)]

Add a keyword: **restrict**

Type Qualifiers [ISO §6.5.3 (X3.159 §3.5.3)]

Syntax

Add a type-qualifier: **restrict**

Constraints

Add a constraint:

Types other than pointer types derived from object or incomplete types shall not be restrict-qualified.

Semantics

Add the following text after line 22.

Let `D` be a declaration of an ordinary identifier that provides a means of designating an object `P` as a restrict-qualified pointer.

If `D` appears inside a block and does not have storage-class `extern`, let `B` denote the block. If `D` appears in the list of parameter declarations of a function definition, let `B` denote the associated block. Otherwise, let `B` denote the block of `main` (or the block of whatever function is called at program startup, in a freestanding environment).

In what follows, a pointer expression `E` is said to be *based* on object `P` if (at some sequence point in the execution of `B` prior to the evaluation of `E`) modifying `P` to point to a copy of the array object into which it formerly pointed would change the value of `E`. (In other words, `E` depends on the value of `P` itself rather than on the value of an object referenced indirectly through `P`. For example, if identifier `p` has type `(int **restrict)`, then the pointer expressions `p` and `p+1` are based on the restricted pointer object designated by `p`, but the pointer expressions `*p` and `p[1]` are not.)

During each execution of `B`, let `O` be the array object that is determined dynamically by all references through pointer expressions based on `P`. Then *all* references to values of `O` shall be through pointer expressions based on `P`. Furthermore, if `P` is assigned the value of a pointer expression `E` that is based on another restricted pointer object `P2`, associated with block `B2`, then either the execution of `B2` shall begin before the execution of `B`, or the execution of `B2` shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

Here an execution of `B` means that portion of the execution of the program during which storage is guaranteed to be reserved for an instance of an object that is associated with `B` and has automatic storage duration. A reference to a value means either an access to or a modification of the value. During an execution of `B`, attention is confined to those references that are actually evaluated (this excludes references that appear in unevaluated expressions, and also excludes references that are "available," in the sense of employing visible identifiers, but do not actually appear in the text of `B`).

A translator is free to ignore any or all aliasing implications of uses of `restrict`.

1.4 Examples

1.4.1 File scope restricted pointers

A file scope restricted pointer is subject to very strong restrictions. It should point into a single array object for the duration of the program. That array object may not be referenced both through the restricted pointer and through either its declared name (if it has one) or another restricted pointer. Because of these restrictions, references through the pointer can be optimized as effectively as references to a static array through its declared name. File scope restricted pointers are therefore useful for providing access to dynamically allocated global arrays. In Figure 1.4, a compiler can deduce from the restrict qualifiers that there is no potential aliasing among the names `a`, `b`, and `c`. Notice how the single block of allocated storage is "subdivided" into two unique arrays in `init`.

Figure 1.4: File scope restricted pointer.

```
float *restrict a, *restrict b;
float c[100];

int init(int n) {
    float *t =  malloc(2 * n * sizeof(float));
    a = t;      /* a refers to 1st half. */
    b = t + n;  /* b refers to 2nd half. */
}
```

1.4.2 Function parameters

Restricted pointers are also very useful as pointer parameters of a function. In the function `f3` in Figure 1.5, it is possible for a compiler to infer that there is no aliasing of modified objects, and so to optimize the loop aggressively. Upon entry to `f3`, the restricted pointer `a` must provide exclusive access to its associated array. In particular, within `f3` neither `b` nor `c` may point into the array associated with `a`, because neither is assigned a pointer value based on `a`. For `b`, this is evident from the const-qualifier in its declaration, but for `c`, an inspection of the body of `f3` is required.

Figure 1.5: Restricted pointer function parameters.

```
float x[100];
float *c;

void f3(int n, float *restrict a, float *const b) {
    int i;
    for ( i = 0; i < n; i++ )
        a[i] = b[i] + c[i];
}
void g3(void) {
    float d[100], e[100];
    c = x; f3(100,  d,    e); /* Behavior defined.   */
           f3( 50,  d, d+50); /* Behavior defined.   */
           f3( 99, d+1,  d); /* Behavior undefined. */
    c = d; f3( 99, d+1,  e); /* Behavior undefined. */
           f3( 99,  e, d+1); /* Behavior defined.   */
}
```

Two of the calls shown in g3 result in aliasing that is inconsistent with the `restrict` qualifier, and their behavior is undefined. Note that it is permitted for `c` to point into the array associated with `b`.

Note also that, for these purposes, the "array" associated with a particular pointer means only that portion of an array object which is actually referenced through that pointer.

For a similar example in which c is another parameter instead of a file scope array, see "Aliasing of unmodified objects" in the appendix on "Discussion of design choices."

1.4.3 Block scope

A block scope restricted pointer makes an aliasing assertion that is limited to its block. This seems more natural than allowing the assertion to have function scope. It allows local assertions that apply only to key loops, for example. It also allows equivalent assertions to be made when inlining a function by converting it into a macro. In Figure 1.6, the original restricted pointer parameter is represented by a block scope restricted pointer.

Figure 1.6: Macro version of f3.

```
float x[100];
float *c;

#define f3(N, A, B)                                    \
{   int n = (N);                                       \
    float *restrict a = (A);                           \
    float *const    b = (B);                           \
    int i;                                             \
    for ( i = 0; i < n; i++ )                          \
        a[i] = b[i] + c[i];                            \
}
```

1.4.4 Members of structures

A restricted pointer member of a structure makes an aliasing assertion, and the scope of that assertion is the scope of the ordinary identifier used to access the structure. Thus although the structure type is declared at file scope in Figure 1.7, the assertions made by the declarations of the parameters of `f4` have block (of the function) scope.

Figure 1.7: Restricted pointers as members of a structure.

```
struct t {      /* Restricted pointers assert that    */
    int n;      /* members point to disjoint storage. */
    float *restrict p;
    float *restrict q;
};

void f4(struct t r, struct t s) {
    /* r.p, r.q, s.p, s.q should all point to    */
    /* disjoint storage during each execution of f4. */
    /* ... */
}
```

1.4.5 Type definitions

A `restrict` qualifier in a type definition makes an aliasing assertion when the typedef name is used in the declaration of an ordinary identifier that provides access to an object. As with members of

structures. it is the scope of the latter identifier, not the scope of the typedef name, that determines the scope of the aliasing assertion.

1.4.6 Expressions based on restricted pointers

In Figure 1.8, the restricted pointer parameter p is potentially adjusted to point into a copy of its original array of two structures. By definition, a subsequent pointer expression is said to be based on p if and only if its value is changed by this adjustment. In the comment, the values of the pointer expressions in the first column are changed by this adjustment, and so those expressions are based on p. The values of the pointer expressions in the second column are not changed by the adjustment, and so those expressions are not based on p. This can be verified by adding appropriate print statements for the expressions, and comparing the values produced by the two calls of `f5` in `main`.

It is important to note that the definition of "based on" applies to expressions that rely on implementation-defined behavior. This is illustrated in this example, which assumes that the casts `(int)` followed by `(struct t *)` give the original value.

Figure 1.8: Pointer expressions based on p.

```
#include <stdlib.h>
#include <string.h>

struct t { int *q; int i; } a[2] = { /* ... */ };

void f5(struct t *restrict p, int c)
{
    struct t *q;
    int n;
    if(c) {
        struct t *r;
        r = malloc(2*sizeof(*p));
        memcpy(r, p, 2*sizeof(*p));
        p = r;
    }
    q = p;
    n = (int)p;
    /* - - - - - - - - - - - - - - - - - - - - - - - - -
      Pointer expressions     Pointer expressions
      based on p              not based on p

      p                       p -> q
      p + 1                   p[1] . q
      &p[1]                   &p
      &p[1] . i
      &p->q
      q                       q -> q
      ++q
      (char *)p               (char *)(p -> i)
      (struct t *)n           ((struct t *)n) -> q
    - - - - - - - - - - - - - - - - - - - - - - - - - */
}

main() {
    f5(a, 0);
    f5(a, 1);
}
```

1.4.7 Assignments between restricted pointers

Let us say that one restricted pointer is "newer" than another if the block with which the first is associated begins execution after the block associated with the second. Then the formal definition allows a newer restricted pointer to be assigned a value based on an older restricted pointer. This allows, for example, a function with a restricted pointer parameter to be called with an argument that is a restricted pointer.

Conversely, an older restricted pointer may be assigned a value based on a newer restricted pointer only after execution of the block associated with the newer restricted pointer has ended. This allows, for example, a function to return the value of a restricted pointer that is local to the function, and the return value then to be assigned to another restricted pointer.

The behavior of a program is undefined if it contains an assignment between two restricted pointers that does not fall into one of these two categories. Some examples are given in Figure 1.9.

Figure 1.9: Assignments between restricted pointers.

```
int *restrict p1, *restrict p2;

void f6(int *restrict q1, int *restrict q2)
{
    p1 = p2;      /* Behavior undefined. */
    p1 = q1;      /* Behavior undefined. */
    q1 = q2;      /* Behavior undefined. */
    {
        int *restrict r1, *restrict r2;
        ...
        r1 = r2; /* Behavior undefined. */
        q1 = r1; /* Behavior undefined. */
        p1 = r1; /* Behavior undefined. */
        ...
    }
}
```

1.4.8 Assignments to unrestricted pointers

This proposal permits assignment of the value of a restricted pointer to an unrestricted pointer, as in Figure 1.10.

Figure 1.10: Assigning restricted to unrestricted pointers.

```
void f7(int n, float *restrict r, float *restrict s) {
    float *p = r, *q = s;
    while(n-- > 0)
        *p++ = *q++;
}
```

If a compiler tracks pointer values, it should be able to optimize the loop as effectively as if the restricted pointers `r` and `s` were used directly.

More complicated ways of combining restricted and unrestricted pointers are less likely to be effective because they are more difficult for a compiler to analyze. As always, a programmer concerned about performance must adapt his style to the capabilities of available compilers. A conservative approach would be to avoid using both restricted and unrestricted pointers in the same function.

1.4.9 Ineffective uses of type qualifiers

Except where specifically noted in the formal definition, the `restrict` qualifier behaves in the same way as `const` and `volatile`. In particular, note that it is not a constraint violation for a function return type or the *type-name* in a cast to be qualified, but the qualifier has no effect because function call and cast expressions are not lvalues. Thus the presence of the `restrict` qualifier in the declaration of `f8` in Figure 1.11 makes no assertion about aliasing in functions that call `f8`. Similarly, the two casts make no assertion about aliasing of the references through the pointers `p` and `r`.

Figure 1.11: Qualified function return type and casts.

```
float *restrict f8(void)  /* No assertion about aliasing. */
{
    extern int i, *p, *q, *r;

    r = (int *restrict)q; /* No assertion about aliasing. */

    for(i = 0; i < 100; i++)
        *(int *restrict)p++ =  r[i];   /* No assertion    */
                                       /* about aliasing. */
    return p;
}
```

1.4.10 Constraint violations

It is a violation of the constraint in ISO §6.5.3 (X3.159 §3.5.3) (of the Standard as modified above) to restrict-qualify an object type which is not a pointer type, or to restrict-qualify a pointer to a function.

Figure 1.12: Restrict cannot qualify non-pointer object types.

```
int restrict x;   /* Constraint violation. */
int restrict *p;  /* Constraint violation. */
```

Figure 1.13: Restrict cannot qualify pointers to functions.

```
float (*restrict f9)(void); /* Constraint violation. */
```

1.5 Appendix: Overlapping objects

The restriction in the *Description* of `memcpy` prohibits copying between "overlapping objects." An interpretation from X3J11 and WG14 in response to DR #042, X3J11 92-001, clarified what is meant by "objects" in this context. The following quotation from that defect report provides a background for the notion of an "object that is determined dynamically" that appears in the formal definition of `restrict`.

> From subclause 3.14, an object is "a region of data storage ... Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined ..."

Figure 1.14: Memcpy between rows of a matrix.

```
void f1(void) {
    extern char a[2][N];
    memcpy(a[1], a[0], N);
}
```

Therefore, the "objects" referred to by subclause 7.11.2.1 are exactly the regions of data storage pointed to by the pointers and dynamically determined to be of `N` bytes in length (i.e. treated as an array of `N` elements of character type).

a) So, no, the objects are not "the largest objects into which the arguments can be construed as pointing".

b) In Figure 1.14, the call to `memcpy` has defined behavior.

c) The behavior is defined because the pointers point into different (non-overlapping) objects.

Figure 1.15: Memcpy between halves of an array.

```
void f2(void) {
    extern char b[2*N];
    memcpy(b+N, b, N);
}
```

a) Yes, for `memcpy`, a contiguous sequence of elements within an array can be regarded as an object in its own right.

b) The objects are not the smallest contiguous sequence of bytes that can be construed; they are exactly the regions of data storage starting at the pointer and of `N` bytes in length.

c) Yes, the non-overlapping halves of array `b` can be regarded as objects in their own rights.

d) The behavior (in Figure 1.15) is defined.

...

Length is determined by "various methods." For strings in which all elements are accessed, length is inferred by null byte termination. For `mbstowcs`, `wcstombs`, `strftime`, `vsprintf`, `sscanf`, `sprintf`, and all other similar functions, it was the intent of the standard that the rules in subclause 7.11.1 be applicable by extension (i.e., the objects and lengths are similarly dynamically determined).

1.6 Appendix: Discussion of design choices

1.6.1 Contiguity

A restricted pointer points to an associated object that is determined dynamically. When all references, both direct and indirect, through a restricted pointer comprise a contiguous sequence of bytes, then the associated object is exactly that sequence. Otherwise, the object consists of both the referenced bytes and those unreferenced bytes that fall between any two referenced bytes.

This convention is consistent with the specification noted above, that "except for bit-fields, objects are composed of contiguous sequences of one or more bytes." It is also consistent with the notions of objects and argument association in Fortran 77. Thus, in environments that support mixed-language programming, restricted pointer parameters can be used in C prototypes for functions defined in Fortran, and also in definitions of C functions that are called from Fortran.

Note that an implementation is free to support restricted pointers to non-contiguous sets of bytes as an extension. Such support is not mandated, however, because the need for it does not seem compelling, and it could inhibit optimizations for architectures that cannot load and store single bytes or that have software-managed caches. Note that such an extension might be useful in implementation-defined calling conventions between C and Fortran 90, because Fortran 90 permits arguments that are non-contiguous sets of array elements.

1.6.2 Aliasing of unmodified objects

Comparison with Fortran raises an issue concerning aliasing of unmodified objects, as illustrated in Figure 1.16. Knowing that parameter `a` provides exclusive access to the array into which it points is sufficient to allow the loop in `f10` to be vectorized or executed in parallel. Knowing that `b` and `c` are not used to access overlapping portions of a single array allows no additional optimizations, and so, in principle, there is no need to restrict-qualify `b` and `c`.

Figure 1.16: Aliasing matters only for modified objects.

```
void f10(int n, float *restrict a, float *b, float *c) {
    int i;
    for ( i = 0; i < n; i++ )
        a[i] = b[i] + c[i];
}
```

But, as a practical matter, it is easier for a compiler to analyze functions in which all pointer parameters are restrict-qualified. This suggests that the semantics of the qualifier should be defined so that one can restrict-qualify all three pointer parameters of `f10` and still make calls of the form `f10(x,y,y)`. This would be analogous to the aliasing semantics of Fortran dummy arguments.

The simplest way of doing this (proposed in Draft 1 of this report) is to allow aliasing through two restrict-qualified pointers provided the referenced objects are not modified. Unfortunately, if those objects are themselves pointers (i.e., there are two levels of indirection), this aliasing can inhibit optimization, even if the secondary pointers are also restrict-qualified and used to modify the objects to which they, in turn, refer. See X3J11.1/93-040 for examples.

In the final analysis, it did not seem possible to permit aliasing of unmodified objects, have effective assertions for multiple levels of indirection, and still keep the semantics relatively simple.

It was decided to drop the special treatment of unmodified objects, largely because the practical effect is quite small. First, as noted, the extra qualifiers are not needed in principle, though they may be useful for specific compilers. Second, in these cases, it is also useful, and might be sufficient, to const-qualify `b` and `c`, because this gives a compiler enough aliasing information in the parameter declarations alone to optimize `f10` (see the discussion of Figure 1.5). Finally, even if a specific compiler requires all three restrict qualifiers to give the desired optimization, it is most unlikely that their use

would give unexpected results for a call such as `f10(x,y,y)`, even though its behavior is technically undefined. Such use may therefore be justified in particular cases, as are other instances of undefined or implementation-defined behavior, such as assigning to one member of a union and then referring to another.

1.6.3 Linked lists

Operations on data contained in the items of a linked list are difficult to optimize, particularly for vector or parallel execution. Because potential aliasing is not the only difficulty, it is unlikely that the use of restricted pointers in this context will make much difference with currently available compilers. Nevertheless, as the use of parallel architectures becomes more widespread, their compilers may well be more ambitious in this area. With this in mind, the examples in Figure 1.17 illustrate how to use the `restrict` and `const` qualifiers to make assertions that could, in principle, enable effective optimizations.

First note that restricted pointers cannot, in general, be used for link pointers in portions of the program that modify the lists. Thus in Figure 1.17, the assignments to `next` and `head` would give undefined behavior if they were restrict qualified.

Figure 1.17: Operations on the structure of a linked list.

```
struct t { struct t *next; float x; } *head, *item;

void modify_links()
    /* Add a list item at the beginning. */
    item -> next = head; head = item;

    /* Add a list item as the second item. */
    item -> next = head -> next; head -> next = item;

    /* Delete second list item. */
    head -> next = head -> next -> next;

    /* Delete first list item. */
    head = head -> next;
}
```

In contrast, in functions in which the structure of the lists is constant, this can be asserted by using a combination of the `restrict` and `const` qualifiers. Note that, as in Figure 1.18, this requires a second structure type, and a cast from the original to the new type. The comments describe an optimization that allows parallel execution of the second loop. Alternatively, the second loop could be fused with the first, so that both operations are performed during a single walk. The latter optimization is likely to be beneficial on any architecture.

Unfortunately, there is a subtle problem with this approach. If there is a function call between the two loops that has `rhead` as a parameter, then there should be a guarantee that the cached links are not changed during the execution of the called function. The problem is that there is nothing to prevent the function from modifying a link via

```
*(struct rt *restrict *)&(rhead -> next) = ...
```

(assuming for the sake of argument that the structures on the list were dynamically allocated as an array of structures). To make the aliasing assertions effective would require forbidding this sort of modification. This might be done be extending the full semantics of the `const` qualfier to include objects designated as const-qualified by a restricted pointer, as well as objects that are const-qualified by their definitions. (This extension is not part of the current proposal.)

Figure 1.18: Operations on the values in a linked lists.

```
struct t { struct t *next; float x; };
struct rt { struct rt *const restrict next; float x; };

void modify_values(struct t *head) {
    /* Assert that links will not change. */
    struct rt *const restrict rhead = (struct rt *)head;

    /* Need modifiable unrestricted pointer to walk list. */
    struct rt *this;

    /* Can build a cache of link addresses. */
    for(this = rhead; this; this = this -> next) {
        this -> x += 1;
    }

    /* Can use cached addresses to execute in parallel. */
    for(this = rhead; this; this = this -> next) {
        this -> x *= 2;
    }
}
```

1.7 Appendix: Array syntax and qualified parameters

By ISO §6.7.1 (X3.159 §3.7.1), a declaration of a function parameter as "array of *type*" is adjusted to "pointer to *type*." Thus the two prototypes in Figure 1.19 for `f11` are compatible.

Figure 1.19: Equivalent parameter declarations.

```
void f11(int n, float a[][100], float b[][100]);
void f11(int n, float (*a)[100], float (*b)[100]);
```

In either case, `a` and `b` are pointers used to access their respective two-dimensional arrays, but the first form more clearly conveys the rank of the arrays. It is therefore unfortunate that only the second form allows the pointers to be qualified, e.g., by `restrict`.

This suggests an extension of the syntax to allow one or more type qualifiers to appear within the "array of" type derivation for a function parameter declared to have array type. The declaration is then adjusted to the analogously qualified pointer type.

Figure 1.20: Restricted version of f11.

```
void f11(int n, float a[restrict][100], float b[restrict][100])
{
    int i, j;
    for(i = 0; i < n; i++)
        for(j = 0; j < 100; j++)
           a[i][j] = b[i][j];
}

void g11(int m, int k, float p[][100]) {
    f11(m, p, p+k); /* Defined if and only if m <= k. */
}
```

This extension would be even more convenient in the presence of a variable length array extension, that would allow another parameter to be used in an array size expression. In Figure 1.21, the compiler, or code reviewer, can use the information in the prototype alone to deduce that the first call of `f12` in `g12` is defined, but the second call is potentially undefined (depending upon which elements of the arrays `f12` actually references).

For consistency, the extension would apply to all type qualifiers. The following changes to the Standard would be required for this extension.

Declarators [ISO §6.5.4 (X3.159 §3.5.4)]

Syntax

Allow an optional type-qualifier-list *in the third form of direct-declarator:*
direct-declarator [*type-qualifier-list* $_{opt}$ *constant-expression* $_{opt}$]

Array Declarators [ISO §6.5.4.2 (X3.159 §3.5.4.2)]

Constraints

Add a constraint:

Type qualifiers shall appear preceding or in place of a size expression only in a declaration of a function parameter of array type, and then only in the outermost array type derivation.

Figure 1.21: Restrict and variable length arrays.

```
void f12(int m, int n, float a[restrict m][n],
                       float b[restrict m][n]);

void g12(int n, float p[][n]) {
        f12(10, n, p, p+10); /* Defined behavior. */
        f12(20, n, p, p+10); /* Potential undefined behavior. */
}
```

Function Definition [ISO §6.7.1 (X3.159 §3.7.1)]

Semantics

Modify lines 23-24 to read:

...A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*" (where the type qualifiers are those specified within the square brackets of the array type derivation).

...

1.8 Appendix: Comparison with noalias

The following is an overview of the differences between `restrict` and the previously proposed `noalias`.

The X3J11 committee attempted to solve the aliasing problem in C by introducing a new type qualifier `noalias`. That effort failed because of technical problems with the proposed semantics of `noalias`. This restricted pointer proposal is different in many ways.

- Only pointers can be declared to be restricted. In the `noalias` proposal, all objects were permitted to be declared noalias-qualified.
- It is the declaration of the restricted pointer that makes the aliasing assertions, while it was the noalias-qualified lvalue that made the aliasing assertions.
- A restricted pointer can be an alias with an unrestricted pointer, whereas a pointer to a noalias-qualified type was guaranteed to be completely alias free.
- The proposed semantics of `noalias` defined an alternate execution path in which virtual objects were created and later synchronized with the original object. No alternate execution path is defined for restricted pointers. The proposed semantics for restricted pointers merely permit the optimizer to statically analyze restricted pointers.
- A block scope restricted pointer only makes assertions on the containing scope. In the `noalias` proposal, a block scope noalias-qualified object made assertions that affected the entire containing function.
- A pointer to a noalias-qualified type made no aliasing assertion if it was a member of a structure (because identifiers designating members of structures were not "handles"). A restricted pointer member of a structure does make an aliasing assertion.

Chapter 2

Arrays of Variable Length

Chapter 2

Arrays of Variable Length

Table of Contents

Introduction 2-1
References 2-1
ISO § 6.1.2.4 (X3.159 § 3.1.2.4) Storage Durations of Objects 2-1
ISO § 6.3 (X3.159 § 3.3) Expressions 2-2
ISO § 6.3.3.4 (X3.159 § 3.3.3.4) The **`sizeof`** Operator 2-2
ISO § 6.3.6 (X3.159 § 3.3.6) Additive Operators 2-2
ISO § 6.4 (X3.159 § 3.4) Constant Expressions 2-3
ISO § 6.5 (X3.159 § 3.5) Declarations 2-3
ISO § 6.5.2 (X3.159 § 3.5.2) Type Specifiers 2-3
ISO § 6.5.2.1 (X3.159 § 3.5.2.1) Structure and Union Specifiers 2-4
ISO § 6.5.4 (X3.159 § 3.5.4) Declarators 2-6
ISO § 6.5.4.2 (X3.159 § 3.5.4.2) Array Declarators 2-6
ISO § 6.5.4.3 (X3.159 § 3.5.4.3) Function Declarators (Including Prototypes) 2-7
ISO § 6.5.6 (X3.159 § 3.5.6) Type Definitions 2-10
ISO § 6.5.7 (X3.159 § 3.5.7) Initialization 2-11
ISO § 6.6 (X3.159 § 3.6) Statements 2-12
ISO § 6.6.4.2 (X3.159 § 3.6.4.2) The **`switch`** Statement 2-13
ISO § 6.6.6.1 (X3.159 § 3.6.6.1) The **`goto`** Statement 2-14
ISO § 6.7.1 (X3.159 § 3.7.1) Function Definitions 2-15
ISO § 7.6.2.1 (X3.159 § 4.6.2.1) The **`longjmp`** Function 2-15
Appendix A Language Syntax Summary 2-16
Appendix B Relaxed Lexical Ordering Semantics 2-17
Appendix C Existing Implementations 2-22

Foreword

The inability to declare arrays whose size is known only at execution time is often cited as a primary deterrent to using C as a numerical computing language. Arrays of this nature are implemented in the current GNU-C and Cray Research C compilers. The adoption of arrays whose size is known only at execution time was proposed to committee X3J11 but was dismissed as having too many far-reaching implications. Eventual adoption of some standard notion of execution time arrays is considered crucial for C's acceptance as a major player in the numerical computing world. This paper is based on an implementation of variable length arrays chosen by Cray Research for its Standard C Compiler, but the specification contains several revisions made by the committee.

Tom MacDonald
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121
tam@cray.com

People who have made valuable contributions to this document are:

Analog Devices, Marc Hoffman
Analog Devices, Kevin Leary
AT&T Bell Laboratories, Dennis Ritchie
Control Data Corporation, Azar Hashemi
Convex Corporation, Austin Curley
Convex Corporation, Bill Torkelson
Cray Research, Incorporated, Dave Becker
Cray Research, Incorporated, Steve Collins
Cray Research, Incorporated, John Dawson
Cray Research, Incorporated, Mike Holly
Cray Research, Incorporated, Bill Homer
Cray Research, Incorporated, David Knaak
Cray Research, Incorporated, Mark Pagel
Cray Research, Incorporated, Kent Zoya
DEC Professional, Rex Jaeschke
Digital Equipment Corporation, Randy Meyers
Farance Inc, Frank Farance
Free Software Foundation, Richard Stallman
Hewlett Packard, John Kwan
IBM, Fred Tydeman
Keaton Consulting, David Keaton
Kendall Square Research, Tim Peters
Lawrence Livermore National Laboratory, Linda Stanberry
Open Software Foundation, Mike Meissner
SunPro, David Hough
SunPro, Bob Jervis
Thinking Machines Corporation, James L. Frankel
University of California at Davis, Harry H. Cheng
Unix System Laboratories, David Prosser

Chapter 2
Arrays of Variable Length

Introduction

This chapter describes a proposal for Variable Length Array (VLA) types. The size of a VLA cannot necessarily be determined at translation time. Rather, its size is usually determined at execution time. Insertion of this new array type into the C Standard requires an examination of each section of the C Standard with necessary elaboration for those sections that are affected. Accordingly, the numbering, titles and general format of each of the following several paragraphs match those found in the current C Standard.

Paragraphs address definitive issues relating to VLAs. If new words are added to a paragraph already in the C Standard, a different font is used to highlight them. Overviews of the issues are given after the sections, followed by a brief rationale that explains why certain decisions were reached. Therefore, this proposal describes each feature as edits to the current C Standard with rationale interspersed appropriately.

References

1. *International Standard Programming Languages—C,* (ISO/IEC 9899:1990(E)).

2. *American National Standard for Information Systems*—Programming Language C (X3.159-1989).

3. "Variable-Size Arrays," Dennis M. Ritchie (NCEG/90-009—January 24, 1990).

4. "Variable Length Arrays and Function Definitions, Revised," Richard Stallman (X3J11.1/92-001—January 2, 1992).

5. "Extending C with Arrays of Variable Length," Harry Cheng (X3J11/94-033—June 6, 1994).

6. "Fortran 90 VLA/Data Parallel Report," Frank Farance (X3J11/94-038—June 8, 1994).

ISO § 6.1.2.4 (X3.159 § 3.1.2.4) Storage Durations of Objects

An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration.* Storage is guaranteed to be reserved for a new instance of such an object on each normal entry into the block with which it is associated. ***If the block with which the object is associated is entered by a jump from outside the block to a labeled statement in the block or in an enclosed block, then storage is guaranteed to be reserved provided the object does not have a variable length array type. If the object is variably modified and the block is entered by a jump to a labeled statement, then the behavior is undefined.*** If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement. Storage for the object is no longer guaranteed to be reserved, when execution of the block ends in any way. (Entering an enclosed block suspends but does not end execution of the enclosing block. Calling a function suspends but does not end execution of the block containing the call.) The value of a pointer that referred to an object with automatic storage duration that is no longer guaranteed to be reserved is indeterminate.

Forward references: *variably modified* (6.5), (3.5), *variable length array* (6.5.4.2), (3.5.4.2), the **`goto`** statement (6.6.6.1), (3.6.6.1).

(continues w/ '98) 87

ISO § 6.3 (X3.159 § 3.3) Expressions

ISO § 6.3.3.4 (X3.159 § 3.3.3.4) The `sizeof` Operator

Semantics

When applied to an operand that has array type, the result is the total number of bytes in the array. For *variable length array* types the result is not a constant expression and is computed at program execution time.

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

Example 1

```
int i = 2, j = 3, k, koo;
int *p = &i;

int func(int n) {
   char b[*p==n ? k = j++ : n+j]; /* side effects OK */
   return sizeof(b);              /* execution time expression */
}

main() {
   koo = func(10);                /* execution time sizeof; koo == 13 */
}
```

ISSUE

Overview:

The **`sizeof`** operator could previously be used in any constant expression (except those associated with preprocessor directives such as **`#if`**). With this extension to the language, this is no longer the case since execution time code must be generated to calculate the size of a *variable length array* type.

Rationale:

The notion of "size" is an important part of such operations as pointer increment, subscripting, and pointer difference. Although the **`sizeof`** operator can now produce a value computed at execution time, there still exists a consistency when applied to the previously mentioned operations. Furthermore, it can still be used in an argument to the **`malloc`** function and to compute the number of elements in an array.

ISO § 6.3.6 (X3.159 § 3.3.6) Additive Operators

Rationale:

The description of pointer arithmetic does not require modification. Pointer arithmetic is still well defined with pointers to *variable length array* types as shown by the following example.

Example 2

```
{
    int n = 4, m = 3;
    int a[n][m];
    int (*p)[m] = a;    /* p == &a[0]     */
    p += 1;             /* p == &a[1]     */
    (*p)[2] = 99;       /* a[1][2] == 99  */
    n = p - a;          /* n == 1         */
}
```

If array **a** in the above example is declared to be a fixed length array, and pointer **p** is declared to be a pointer to a fixed length array that points to **a**, the computed results are still the same.

ISO § 6.4 (X3.159 § 3.4) Constant Expressions

Semantics

An *integral constant expression* shall have integral type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions whose operand does not have *variable length array* type or a parenthesized name of such a type, and floating constants that are the immediate operands of casts. An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and **sizeof** expressions whose operand does not have *variable length array* type or a parenthesized name of such a type.

ISO § 6.5 (X3.159 § 3.5) Declarations

Semantics

If the sequence of specifiers in a declarator contains a *variable length array* type, the type specified by the declarator is said to be *variably modified.*

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

ISO § 6.5.2 (X3.159 § 3.5.2) Type Specifiers

Constraints

Only identifiers with block scope or function prototype scope can have a *variably modified* type. Only ordinary identifiers (as defined in **6.1.2.3 (3.1.2.3)**) without linkage may be declared with a *variable length array* type. If an identifier is declared to be an object with static storage duration, it shall not have a *variable length array* type.

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

ISSUES

Overview:

This section discusses three issues about declarations containing *variable length array* specifiers. It is important to note that there is a distinction made between *variable length array* types and *variably modified* types (e.g., a pointer to a *variable length array*). First, all declarations of *variably modified* types must be declared at either block scope or function prototype scope. This means that file scope identifiers cannot be declared with a *variably modified* type of any fashion. Second, array objects declared with

either the **static** or **extern** storage class specifiers cannot be declared with a *variable length array* type. However, block scope pointers declared with the **static** storage class specifier can be declared as pointers to *variable length array* types. Finally, if the identifier that is being declared has a *variable length array* type (as opposed to a pointer to a *variable length array),* then it must be an ordinary identifier. This eliminates structure and union members.

Example 3

```
extern int n;
typedef int T[n];                    /* Error - file scope identifier */
struct t { int (*z)[n]; };           /* Error - file scope identifier */
int A[n];                            /* Error - file scope identifier */
static int (*p1)[n];                 /* Error - file scope identifier */
extern int (*p2)[n];                 /* Error - file scope identifier */
int B[100];                          /* OK - file scope but not a VLA */

void func(int m, int C[m][m]) {      /* OK - prototype VLA */
   typedef int VLA[m][m];            /* OK - block scope typedef VLA */
   struct tag {
      int (*y)[n];                   /* OK - block scope member pointer to VLA */
      int z[n];                      /* Error - z is not an ordinary identifier */
   };
   int D[m];                         /* OK - auto VLA */
   static int E[m];                  /* Error - static block scope VLA */
   extern int F[m];                  /* Error - F has linkage and is a VLA */
   int (*s)[m];                      /* OK - auto pointer to VLA */
   extern int (*r)[m];               /* Error - r has linkage and is a pointer to VLA */
   static int (*q)[m] = &B;          /* OK - static block scope pointer to VLA */
}
```

Rationale:

Restricting *variable length array* declarators to identifiers with automatic storage duration is natural since ''variableness'' at file scope requires some notion of parameterized typing.

ISO § 6.5.2.1 (X3.159 § 3.5.2.1) Structure and Union Specifiers

Constraints

A structure or union shall not contain a member with a *variable length array* type.

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

ISSUES

Overview:

This issue involves the declaration of members of structures or unions with *variably modified* types. A structure member can be declared to be a pointer to a VLA, but not a VLA. Allowing structure members to have a VLA type introduces a host of problems, such as the treatment when passing such objects (or even pointers to such objects) as parameters. In addition, the semantics of the **offsetof** macro would need to be extended for examples similar to the following.

Example 4

```
int n = 8;
main() {
   struct tag {
      int m1;
      int m2[n];                    /* not allowed by this proposal */
      int m3;
   };
   int i;
   i = offsetof(struct tag, m1); /* OK */
   i = offsetof(struct tag, m2); /* OK */
   i = offsetof(struct tag, m3); /* undefined behavior? */
}
```

Because the expansion of **offsetof** is implementation specific, it seems impractical to guarantee any behavior for members which follow a *variable length array* type member. Finally, any structure or union containing a *variable length array* type must not be declared at file scope due to its variable nature. This forces every user to re-specify the structure type inside each function, and raises type compatibility questions. Allowing formal arguments to be structures with *variable length array* type members (as well as pointers to same) is problematic and, since functions could not return them either, the utility of the whole exercise centered around stack allocated structures with variable length members. Objects of this sort are narrowly focused and thus the decision was made to disallow the *variable length array* type within structures and unions.

If all *ordinary identifiers* found in the arbitrary size expressions of a given member were compelled to resolve to other members of the same structure or union, then some utility could be found in allowing structures and unions to have *variably modified* members. Such members would become self-contained "fat pointers" of sorts. The following example demonstrates this concept:

Example 5

```
struct tag {        /* not allowed by this proposal */
   int n;           /* initialized to 5 */
   int m2[n];       /* uses member n to complete size */
} x = { 5 };
```

The possibility of resolving the size of *variable length array* members amongst other members could be explored. There were two votes taken at NCEG meeting #5 in Norwood, MA, which asked the following questions:

Q: Should there be some way to declare a VLA member
Yes: 11 No: 1 Undecided: 4

Q: In favor of type shown in Example 4? 3
Q: In favor of type shown in Example 5? 6
Q: Undecided? 7

There is considerable sentiment for adding VLA members but it appears the implications of each approach needs to be explored before the committee can reach consensus. This proposal has elected to pursue VLAs outside of structures and unions. A future proposal may attempt to resolve the VLA issue for members.

ISO § 6.5.4 (X3.159 § 3.5.4) Declarators

Semantics

Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

In the following subsections, consider a declaration

```
T D1
```

where `T` contains the declaration specifiers that specify a type *T* (such as `int`) and `D1` is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

If, in the declaration "`T D1`," `D1` has the form

identifier

then the type specified for *ident* is *T*.

If, in the declaration "`T D1`," `D1` has the form

```
( D )
```

then *ident* has the type specified by the declaration "`T D`." Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complex declarators may be altered by parentheses.

ISO § 6.5.4.2 (X3.159 § 3.5.4.2) Array Declarators

Constraints

The `[` and `]` may delimit an expression or `*`. If `[` and `]` delimit an expression (which specifies the size of an array), it shall have an integral type. If the expression is a constant expression then it shall have a value greater than zero.

Semantics

If, in the declaration "`T D1`," `D1` has the form

`D[`*assignment-expression*$_{opt}$`]`

or

```
D[*]
```

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T,*" then the type specified for *ident* in "`T D`" is "*derived-declarator-type-list* array of *T.*" If the size expression is not present the array type is an incomplete type. ***If * is used instead of a size expression, the array type is a variable length array type of unspecified size, which can only be used in declarations with function prototype scope. If the size expression is an integer constant expression and the element type has a fixed size, the array type is a fixed length array type. Otherwise, the array type is a variable length array type. If the size expression is not a constant expression, it is evaluated at program execution time, it may contain side effects, and shall evaluate to a value greater than zero. The size of a variable length array type does not change until the execution of the block containing the declaration has ended.***

For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present ***and integer constant expressions, then both size specifiers shall have the same constant value. If either size specifier is variable, the two sizes must evaluate, at program execution time, to***

equal values. If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values at execution time.

Example 6

```
extern int n;
extern int m;

func() {
   int a[n][6][m];
   int (*p)[4][n+1];
   int c[n][n][6][m];
   int (*r)[n][n][n+1];
   p = a;      /* Error - not compatible because 4 != 6 */
   r = c;      /* compatible, but undefined behavior unless n == 6 and m == n+1 */
}
```

Example 7

```
int dim4 = 4;
int dim6 = 6;

void main() {
   int (*q)[dim4][8][dim6];
   int (*r)[dim6][8][1];
   r = q;      /* compatible, but undefined behavior at execution time */
}
```

ISSUES

Overview:

An issue raised by this section concerns type compatibility and the fact that *variably modified* types are not fully specified until execution time. Is that when compatibility checks should occur? Should the checks be mandatory? This kind of execution time checking is considered to be a quality of implementation issue. Finally, the presence of possible side effects in type declarations is new. Since side effects may include accessing volatile objects there seems to be no compelling reason to forbid side effects in *variable length array* size specifiers.

Rationale:

The language chosen by this proposal reflects a "path of least resistance" approach. The "undefined behavior" paradigm is invoked if things do not match up at execution time as they normally would for *fixed length arrays.* Prototype side-effects are handled with a syntactic option (see use of `[*]` in section **6.5.4.3 (3.5.4.3)** below) and by inhibiting expression evaluation until function definition time.

ISO § 6.5.4.3 (X3.159 § 3.5.4.3) Function Declarators (Including Prototypes)

Semantics

For each parameter declared with *variable length array* type, the type used for compatibility comparisons is the one that results from conversion to a pointer type, as in **6.7.1 (3.7.1)**. In a function prototype declarator that is not part of a function definition, the syntax `[*]` may be used to specify a *variable length array* type.

Example 8

```
/* The following prototype has a variably modified parameter */

void addscalar(int n, int m, double a[n][n*m+300], double x);

double A[4][308];

main() {
   addscalar(4, 2, A, 3.14);
}

void addscalar(int n, int m, double a[n][n*m+300], double x) {

   int i, j, k=n*m+300;

   for (i = 0; i < n; i++)
      for (j = 0; j < k; j++)
         a[i][j] += x;     /* a is a pointer to a VLA of size: n*m+300 */
}
```

The following are all compatible function prototype declarators.

Example 9

```
void matrix_mult(int n, int m, double a[n][m], double b[m][n], double c[n][n]);

void matrix_mult(int n, int m, double a[*][*], double b[*][*], double c[*][*]);

void matrix_mult(int n, int m, double a[ ][*], double b[ ][*], double c[ ][*]);

void matrix_mult(int n, int m, double a[ ][m], double b[ ][n], double c[ ][n]);
```

The size expression can involve identifiers declared at file scope. The following example shows how this can be done.

Example 10

```
extern int n;

void f(double a[][n], int n) { ... }  /* file scope n used */

extern int m;

void g(double b[][n*m]) { ... }
```

The following example is also acceptable. The parameter **n** is used to complete the VLA size expression.

Example 11

```
extern int n;

void g(int n, int a[][n]) {  /* OK */
   ...
}
```

ISSUES

Overview:

Currently, programmers do not need to concern themselves with the order in which formal parameters are specified, and one common programming style is to declare the most important parameters first. Consider the following prototype declaration and function definition:

Example 12

```
/* prototype declaration for old-style definition */

void f(double a[*][*], int n);

void f(a, n)
   int n;
   double a[n][n];
{
   ...
}
```

The order in which the names are specified in the parameter list does not depend on the order of the parameter declarations themselves. However, because of the lexical ordering rules in Standard C, the accompanying prototype declaration is not allowed. All identifiers used in VLA size expressions must be declared prior to use (as is the case with the usage of all identifiers).

Example 13

```
void f(double a[n][n], int n) {  /* Error - n declared after use */
   ...
}
```

With Standard C's lexical ordering rules the declaration of **a** would force **n** to be undefined or captured by an outside declaration. The possibility of allowing the scope of parameter **n** to extend to the beginning of the parameter-type-list was explored (relaxed lexical ordering). This allows the size of parameter **a** to be defined in terms of parameter **n**. However, such a change to the lexical ordering rules is not considered to be in the "Spirit of C." This is an unforeseen side-effect of Standard C prototype syntax. The full proposal for relaxed lexical ordering semantics is in **Appendix B**.

The following example shows how to declare parameters in any order and avoid lexical ordering issues.

Example 14

```
void g(void *ap, int n) {
   double (*a)[n] = ap;

   ... a[1][2] ...
}
```

In this case, the parameter **ap** is assigned to a local pointer that is declared to be a pointer to a VLA.

Another issue concerns function prototype declarators whose parameters have *variable length array* types. Since the rules for parameters inside function prototype declarators with *variable length array* types that do not use the **[*]** syntax are the same whether or not the function declarator is part of a definition, any identifier used to specify the size of these *variable length array* types needs to be visible.

```
/* prototype declarator with an incorrect parameter type */
/* Error - x and y are never declared */

void f1(double a[x][y]);
```

This is the primary reason behind providing the **[*]** syntax.

```
void f1(double a[*][*]);  /* OK */
```

Rationale:

This issue concerns prototype declarators that contain parameters with *variable length array* types that are never completed. A previous version of this document allowed this behavior. The motivation was to not force a diagnostic because the actual array sizes were never needed. A vote that was taken at NCEG meeting #5 in Norwood, MA, asked the following question:

Q: Which notation is preferred to denote a VLA parameter (in a prototype)?

Arbitrary **[x]** (where **x** can include undeclared names) : 0
[*] : 11
Undecided : 3

Since the **[*]** syntax seems to be the most widely accepted this proposal has been changed to reflect this sentiment. Also, the presence of these diagnostics will help uncover more programmer errors.

ISO § 6.5.6 (X3.159 § 3.5.6) Type Definitions

Constraints

Typedef declarations which specify a *variably modified* type shall have block scope. The array size specified by the *variable length array* type shall be evaluated at the time the type definition is declared and not at the time it is used as a type specifier in an actual declarator.

Example 15

```
main() {
   extern void func();
   func(20);
}

void func(int n) {
   typedef int A[n];     /* OK - because declared in block scope */
   A a;
   A *p;
   p = &a;
}
```

Example 16

```
int n;
typedef int A[n];        /* Error - because declared at file scope */
```

Example 17

```
void func(int n) {
   typedef int A[n]; /* A is n ints with n evaluated now */
   n += 1;
   {
      A a;           /* a is n ints - n without += 1 */
      int b[n];      /* a and b are different sizes */
      for (i = 1; i < n; i++)
         a[i-1] = b[i];
   }
}
```

ISSUE

Overview:

The question arises whether the non-constant expression which determines the size of the *variable length array* type should be evaluated at the time the type definition itself was declared or each time the type definition is invoked for some object declaration.

Rationale:

If the evaluation were to take place each time the typedef name is used, then a single type definition could yield *variable length array* types involving many different dimension sizes. This possibility seemed to violate the spirit of type definitions and a decision was made to force evaluation of the expression at the time the type definition itself is declared.

ISO § 6.5.7 (X3.159 § 3.5.7) Initialization

Constraints

The type of the entity to be initialized shall be an object type or an array of unknown size, but not a *variable length array* type.

Example 18

```
extern int n;
main() {
   int a[n] = {1, 2};      /* Error */
   int (*p)[n] = &a;       /* OK - pointer is scalar of fixed size */
}
```

ISO § 6.6 (X3.159 § 3.6) Statements

Semantics

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression: ***a variably modified declarator;*** an initializer; the expression in an expression statement; the controlling expression of a selection statement (**if** or **switch**); the controlling expression of a **while** or **do** statement; each of the three (optional) expressions of a **for** statement; the (optional) expression in a **return** statement. The end of a full expression is a sequence point.

6.6.2 (3.6.2) Compound Statement, or Block

Semantics

A *compound statement* (also called a *block*) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations (as discussed in **6.1.2.4** (**3.1.2.4**)). The initializers of objects that have automatic storage duration, ***and declarators that have variably modified type*** are evaluated and the values are stored in the objects in the order their declarators appear in the translation unit.

Example 19

```
{
   int n = 3, m = 4;
   int a[n++][n++];        /* order of evaluation is undefined */
   int b[m++], c[m++];     /* order of evaluation is defined */
}
```

Rationale:

Since sequence points cause the side effects of previous evaluations to be complete, it seems reasonable to add a sequence point after every *variably modified* declarator. This specifies the behavior of declarations such as:

```
int x[n++];
int y[n++];
```

in that the side effects must be evaluated at the end of the declarator. However, the behavior of a declaration such as:

```
int z[n++][n++];
```

is not specified because there is no guaranteed order of evaluation until the declarator is complete. This seems consistent with an expression such as:

```
z[n++][n++]
```

in which there is no guaranteed order of evaluation either.

ISO § 6.6.4.2 (X3.159 § 3.6.4.2) The `switch` Statement

Constraints

The controlling expression shall not cause a block to be entered by a jump from outside the block to a statement that follows a **`case`** or **`default`** label in the block (or an enclosed block) if that block contains the declaration of a *variably modified* object or *variably modified* typedef name.

Example 20

```
int i = 0;
main() {
   int n = 10;
   switch (n) {      /* Error - bypasses declaration of a[n] */
      int a[n];
      case 10:
         a[0] = 1;
         break;
      case 20:
         a[0] = 2;
         break;
   }
   switch (i) {      /* OK - declaration of b[n] is not bypassed */
      case 0:
         {
            int b[n];
            b[2] = 4;
         }
         break;
      case 1:
         break;
   }
}
```

ISSUE

Overview:

The concern here is that a **`switch`** will bypass evaluation of the arbitrary integer expressions associated with a *variably modified* type. In the case of a simple pointer that is *variably modified* this could cause incorrect pointer arithmetic. In the case of a *variable length array* type this is even more serious in that the *variably modified* object itself is not even allocated.

Rationale:

Disallowing a jump passed the declaration of a *variably modified* identifier in this fashion seems reasonable and a compile time diagnostic is required.

ISO § 6.6.6.1 (X3.159 § 3.6.6.1) The `goto` Statement

Constraints

The identifier in a **`goto`** statement shall name a label located somewhere in the enclosing function. A **`goto`** statement shall not cause a block to be entered by a jump from outside the block to a labeled statement in the block (or an enclosed block) if that block contains the declaration of a *variably modified* object or *variably modified* typedef name.

Example 21

```
void func(int n) {
     int j = 4;
     goto lab3;        /* Error - going INTO scope of variable length array */
     {
        double a[n];
        a[j] = 4.4;
lab3:
        a[j] = 3.3;
        goto lab4;     /* OK - going WITHIN scope of variable length array */
        a[j] = 5.5;
lab4:
        a[j] = 6.6;
     }
     goto lab4;        /* Error - going INTO scope of variable length array */
     return;
}
```

ISSUE

Overview:

The concern here is that a **`goto`** will bypass the evaluation of the arbitrary integer expressions associated with a *variably modified* type. In the case of a simple pointer that is *variably modified* this causes incorrect pointer arithmetic. In the case of a *variable length array* object, the object itself would not even be allocated.

Rationale:

Disallowing a jump passed the declaration of a *variably modified* identifier in this fashion seems reasonable and a compile time diagnostic is required.

ISO § 6.7.1 (X3.159 § 3.7.1) Function Definitions

On entry to the function all size expressions of *variably modified* parameters are evaluated.

ISO § 7.6.2.1 (X3.159 § 4.6.2.1) The `longjmp` Function

Description

If a **`longjmp`** function invocation causes the termination of a function or block in which *variable length array* objects are still allocated, then the behavior is undefined.

ISSUE

Overview:

The **`longjmp`** function that returns control back to the point of the **`setjmp`** invocation might cause memory associated with a *variable length array* object to be squandered. Consider the following example:

Example 22

```
#include <setjmp.h>
void g(int n);
void h(int n);
int n = 6;

void f() {
   int x[n];          /* OK - f is not terminated */
   setjmp(buf);
   g(n);
}

void g(int n) {
   int a[n];          /* undefined - a is still allocated */
   h(n);
}

void h(int n) {
   int b[n];          /* undefined - b is still allocated */
   longjmp(buf,2);    /* causes undefined behavior */
}
```

In this example the function **`h`** might cause storage to be lost for the *variable length array* object **`a`**, declared in function **`g`** (whose existence is unknown to **`h`**). Additional storage could be lost for *variable length array* object **`b`** declared in function **`h`**. In this case an implementation knows that a **`longjmp`** is being performed in the presence of a *variable length array* object whose storage needs to be managed accordingly. However, since **`longjmp`** is a function, it can be invoked through a "pointer to function" and thus **`h`** would have no knowledge of the **`longjmp`** invocation occurring.

Rationale:

These semantics prevent storage from being lost by the execution environment in a conforming program.

A Language Syntax Summary

A.1.2.2 Declarations

(ISO § 6.5.4), (X3.159 § 3.5.4) direct-declarator:

identifier

(*declarator*)

direct-declarator [*assignment-expression*$_{opt}$]

direct-declarator [*$_{opt}$]

direct-declarator (*parameter-type-list*)

direct-declarator (*identifier-list*$_{opt}$)

(ISO § 6.5.5), (X3.159 § 3.5.5) direct-abstract-declarator:

(*abstract-declarator*)

direct-abstract-declarator [*assignment-expression*$_{opt}$]

direct-abstract-declarator [*$_{opt}$]

direct-abstract-declarator (*parameter-type-list*$_{opt}$)

B Relaxed Lexical Ordering Semantics

The following is a description of changes necessary to implement the relaxed lexical ordering rules for *variable length array* types declared in function prototypes. The relaxed lexical ordering rules allow the size expressions of these *variable length array* declarations to contain variables declared lexically after the arrays (but still inside the same prototype). The relaxed lexical ordering rules permit parameters to be declared in any order. This change to the lexical ordering rules was debated at great length and finally rejected because it's not in the "Spirit of C." It is included in this appendix to help others wishing to do research in this area.

ISO § 6.5.4.3 (X3.159 § 3.5.4.3) Function Declarators (Including Prototypes)

Semantics

For each parameter declared with *variable length array* type, the type used for compatibility comparisons is the one that results from conversion to a pointer type, as in **6.7.1** (**3.7.1**). In a function prototype declarator that is not part of a function definition, the syntax `[*]` may be used to specify a *variable length array* type. In a parameter-type-list, if an identifier is both declared as a parameter and appears in an array size expression, then the scope of that identifier is extended to the beginning of the parameter-type-list (rather than beginning just after the completion of its declarator). If the size of a *variable length array* depends upon the size of itself, or another variable length array declared in the same parameter-type-list, then the behavior is undefined.

Example 23

```
/* The following prototype has a variably modified parameter */
void addscalar(int n, int m, double a[n][n*m + 300], double x);
double A[4][308];

main() {
   addscalar(4, 2, A, 3.14);
}

void addscalar(int n, int m, double a[n][n*m + 300], double x) {

   int i,j,k=n*m+300;

   for (i = 0; i < n; i++)
      for (j = 0; j < k; j++)
         a[i][j] += x;     /* a is pointer to a VLA of size: n*m+300 */
}
```

Example 24

```
/* The following are compatible function prototype declarators: */
void matrix_mult(int n, int m, double a[n][m], double b[m][n], double c[n][n]);

void matrix_mult(int n, int m, double a[*][*], double b[*][*], double c[*][*]);

void matrix_mult(int n, int m, double a[ ][*], double b[ ][*], double c[ ][*]);

void matrix_mult(int n, int m, double a[ ][m], double b[ ][n], double c[ ][n]);
```

Since the order in which parameters are specified is unimportant, the following compatible function prototypes are functionally equivalent to the above four prototypes in that the values of parameters **n** and **m** are used to complete the *variable length array* modifiers.

```
void matrix_mult(double a[n][m], double b[m][n], double c[n][n], int n, int m);

void matrix_mult(double a[*][*], double b[*][*], double c[*][*], int n, int m);

void matrix_mult(double a[ ][*], double b[ ][*], double c[ ][*], int n, int m);

void matrix_mult(double a[ ][m], double b[ ][n], double c[ ][n], int n, int m);
```

The following atypical example shows that the meaning of some currently conforming functions change with the relaxed lexical ordering rules. The parameter **n** is used in the array size expression instead of the enumeration constant **n**. The relaxed lexical ordering rules eliminate this kind of surprising behavior when, for example, an enumeration constant **n** (that might be present in a header file), inadvertently changes the meaning of an intended VLA declaration. The desired behavior is to use parameter **n** as the array size specifier, instead of the enumeration constant **n**.

Example 25

```
enum { n = 5 };

/* The parameter n is used instead of enum constant n */
void f1(int a[][n], int n) { ... }
```

Because the scope of parameter **n** extends to the beginning of the parameter-type-list, the size used by parameter **a** is the intended parameter **n**.

The following atypical program is not currently standard conforming because the parameter **x** declared in function **func** is a VLA. Again the scope of parameter **t** is extended to the beginning of the parameter list.

Example 26

```
enum {r=3, s=4, t=5};
float a[6][6];

void func(int, float [][*], int);

main() {
   func(4, a, 2);   /* need ptr to 6 element array */
}

/* The parameter t is used instead of enum constant t */

void func(int n, float x[][n+t], int t) {
   printf("%s\n",
      sizeof(*x)==sizeof(float [6]) ? "OK" : "Compiler error");
}
```

The following function declarator contains an error because the array sizes of parameters **p** and **q** depend upon the size of a VLA declared within the parameter-type-list.

Example 27

```
/* Error - p and q use VLA sizes of parameters */

int func(int (*p)[sizeof(*q)], int (*q)[sizeof(*p)]) { ... }
```

In the following function prototypes, the parameter **n** is used to complete the variable length array types. The file scope variable **n** is never referenced.

Example 28

```
extern int n;          /* file scope variable is never referenced!! */

void f(double a[][n], int n) { ... }

extern int m;          /* file scope variable */

void g(double b[][n*m], int n) { ... }
```

The following example is also acceptable. The parameter **n** is used to complete the VLA size expression.

Example 29

```
extern int n;

void g(int n, int a[][n]) {  /* OK */
   ...
}
```

ISSUES

Overview:

By far the most controversial issue involves the "lexical ordering problem" that is presented when prototypes with *variably modified* parameters are used and the size expression involves an identifier that is not visible. Currently, programmers do not need to concern themselves with the order in which formal parameters are specified, and one common programming style is to declare the most important parameters first. Consider the following prototype definition:

Example 30

```
/* prototype declaration for old-style definition */

void f(double a[*][*], int n);

void f(a, n)
   int n;
   double a[n][n];
{
   ...
}
```

The order in which the names are specified in the parameter list does not depend on the order of the parameter declarations themselves. The accompanying prototype declaration is compatible with this definition and thus it seems appropriate to allow the following prototype definition in lieu of the previous old-style definition.

Example 31

```
void f(double a[n][n], int n) {
   ...
}
```

With current lexical ordering rules the declaration of **a** would force **n** to be undefined or captured by an outside declaration; but in no case would the parameter **n** be used as the programmer clearly intends. This is an unforeseen side-effect of Standard C prototype syntax. The following vote that was taken at NCEG meeting #5 in Norwood, MA, about the importance of preserving lexical ordering.

In favor of preservation of lexical ordering? 4
In favor of more relaxed rules? 4
Undecided? 7

At X3J11.1 meeting #1 in Cupertino, CA, a proposal was made to require a permutation such that all identifiers in size expressions must associate with a visible definition, and that no two permutations result in different associations. This same vote was taken again:

In favor of preservation of lexical ordering? 7
In favor of more relaxed ordering? 7
Undecided? 2

Rationale:

By assigning all identifiers in VLA size expressions a *universal type* that is compatible with any type, the parsing may proceed to the matching **]** token. A possible solution to the "lexical ordering problem" then involves tokenizing, syntactically analyzing, and marking where the *variably modified*

expressions are found within dimension specifiers, until the) that terminates the prototype is found. A second scan of the these *variably modified* expressions associates identifiers with universal types to visible declarations thereby completing the VLA types. If during the second scan it is determined that an identifier used in a VLA size expression is still not visible, a diagnostic message is produced.

Another issue concerns prototype declarators that contain parameters with *variable length array* types that are never completed. A previous version of this document allowed this behavior. The motivation was to not force the implementation to make two passes over the prototype just to issue diagnostics. A vote that was taken at NCEG meeting #5 in Norwood, MA, asked the following question:

> Q: Which notation is preferred to denote a VLA parameter (in a prototype)?
>
> Arbitrary `[x]` (where `x` can include undeclared names) : 0
> `[*]` : 11
> Undecided : 3

Since the `[*]` syntax seems to be the most widely accepted this proposal has been changed to reflect this sentiment. Also, the presence of these diagnostics will help uncover more programmer errors.

C Existing Implementations

- Cray Research, Inc. — Standard C Compiler (SCC)
- Free Software Foundation, Inc. — GNU CC

Chapter 3

Designated Initializers

Chapter 3

Designated Initializers

Designated Initializers

David Prosser (dfp@summit.novell.com)
David Keaton (dmk@dmk.com)

1. Introduction

1.1 Purpose

This document specifies the form and interpretation of a pure extension to the language portion of the C standard to provide important additional flexibility to initializers.

1.2 Scope

This document, although extending the C standard, still falls within the scope of that standard, and thus follows all rules and guidelines of that standard except where explicitly noted herein.

1.3 References

1. ANSI X3.159-1989, *American National Standard for Information Systems — Programming Language — C*.
2. ISO/IEC 9899:1990, *Programming Languages — C*.

All references to clauses of the ISO standard and to sections of the former X3.159 standard will be presented in pairs. For example, ISO §6.4 (X3.159 §3.4) references constant expressions.

2. Language

2.1 Designated Initializers

The syntax for initializers in ISO §6.5.7 (X3.159 §3.5.7) is changed to the following, and the constraints and semantics are augmented by the following:[1]

Syntax

initializer:
    assignment-expression
    { *initializer-list* **}**
    { *initializer-list* **,** **}**

initializer-list:
    $designation_{opt}$ *initializer*
    initializer-list **,** $designation_{opt}$ *initializer*

designation:
    designator-list **=**

designator-list:
    designator
    designator-list designator

designator:
    [*constant-expression* **]**
    . *identifier*

1. The = punctuator that ends the designation is unnecessary, strictly speaking. The redundancy can help an implementation recover from syntactic errors, and also simplifies any future extensions in this area (such as some form of repetition).

Constraints

No initializer shall attempt to provide a value for an object not contained within the entity being initialized.[2]

If a designator has the form

[*constant-expression*]

then the current object (defined below) shall have array type and the expression shall be an integral constant expression that shall evaluate to a valid index for that array. If the array is of unknown size, any nonnegative index value is valid.

If a designator has the form

. *identifier*

then the current object (defined below) shall have structure or union type and the identifier shall be a member of that type.

Semantics

Each brace-enclosed initializer list has an associated *current object*. When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first member of a union.[3] In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. Initialization continues forward, as normal, after the designated initializer.[4] (The designation can be thought of as a "goto" within the current object.)

Each designator list begins its description with the current object associated with the closest-surrounding brace pair. Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.[5] The current object that results at the end of the designator list is the subobject to be initialized by the following initializer.

The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject; all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration.

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer.[6]

Examples

Arrays can be initialized to correspond to the elements of an enumeration by using designators:

---

2. This replaces the current first constraint of "There shall be no more initializers in an initializer list than there are objects to be initialized."

3. If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists.

4. As usual, after a union member is initialized, the next object is not the next member of the union; instead, it is the next subobject of an object containing the union.

5. Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. Note, too, that each separate designator list is independent.

6. This encompasses the current "size determined by the number of initializers" rule.

```
enum { Mem_One, Mem_Two, /*...*/ };
const char *nm[] = {
        [Mem_Two] = "Mem Two",
        [Mem_One] = "Mem One",
        /*...*/
};
```

Structure members can be initialized to nonzero values without depending on their order:

```
div_t answer = { .quot = 2, .rem = -1 };
```

Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood:

```
struct { int a[3], b; } w[] = { [0].a = {1}, [1].a[0] = 2 };
```

Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] = {
        1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

In the above, if **MAX** is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

Finally, any member of a union can be initialized:

```
union { /*...*/ } u = { .any_member = 42 };
```

Chapter 4

Compound Literals

Chapter 4

Compound Literals

Compound Literals

David Prosser (dfp@summit.novell.com)
David Keaton (dmk@dmk.com)

1. Introduction

1.1 Purpose

This document specifies the form and interpretation of a pure extension to the language portion of the C standard to provide important additional flexibility to literals in expressions.

1.2 Scope

This document, although extending the C standard, still falls within the scope of that standard, and thus follows all rules and guidelines of that standard except where explicitly noted herein.

1.3 References

1. ANSI X3.159-1989, *American National Standard for Information Systems — Programming Language — C*.
2. ISO/IEC 9899:1990, *Programming Languages — C*.
3. "Designated Initializers," Chapter 3, *X3J11 Technical Report*, 1995.

All references to clauses of the ISO standard and to sections of the former X3.159 standard will be presented in pairs. For example, ISO §6.4 (X3.159 §3.4) references constant expressions.

2. Language

2.1 Compound Literals

The syntax, constraints, and semantics for *postfix-expression* in ISO §6.3.2 (X3.159 §3.3.2) are augmented by the following:

Syntax

postfix-expression:
(*type-name*) { *initializer-list* }
(*type-name*) { *initializer-list* , }

Constraints

The type name shall specify an object type or an array of unknown size.

No initializer shall attempt to provide a value for an object not contained within the entire unnamed object specified by the compound literal.[1]

If the postfix expression occurs in any context other than within the body of a function, the initializer list shall consist of constant expressions.

Semantics

A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is known as a *compound literal*. It provides an unnamed object with value given by the initializer list.[2]

---

1. This is the "There shall be no more initializers..." constraint modified to take into account designated initializers (see chapter 3).

2. Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types only, and the result of a cast expression is not an lvalue.

)17

If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in ISO §6.5.7 (X3.159 §3.5.7), and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

The value of the compound literal is that of an unnamed object initialized by the initializer list. The object has static storage duration if and only if the postfix expression occurs in a context other than within the body of a function; otherwise, it has automatic storage duration associated with the enclosing block.

Except that the initializers need not be constant expressions (when the unnamed object has automatic storage duration), all the semantic rules and constraints for initializer lists in ISO §6.5.7 (X3.159 §3.5.7) are applicable to compound literals.[3] The order in which any side effects occur within the initialization list expressions is unspecified.[4]

String literals, and compound literals with const-qualified types, need not designate distinct objects.[5]

Examples

The file scope definition

```
int *p = (int []){2, 4};
```

initializes **p** to point to the first element of an array of two **int**s, the first having the value two and the second, four. The expressions in this compound literal must be constant. The unnamed object has static storage duration.

In contrast, in

```
void f(void)
{
        int *p;
        /*...*/
        p = (int [2]){*p};
        /*...*/
```

p is assigned the address of an unnamed automatic storage duration object that is an array of two **int**s, the first having the value previously pointed to by **p** and the second, zero.

Designated initializers (see chapter 3) readily combine with compound literals. On-the-fly structure objects can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
         (struct point){.x=3, .y=4});
```

Or, if **drawline** instead expected pointers to **struct point**:

```
drawline(&(struct point){.x=1, .y=1},
         &(struct point){.x=3, .y=4});
```

A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

The following three expressions have different meanings:

---

3. For example, subobjects without explicit initializers are initialized to zero.

4. In particular, the evaluation order need not be the same as the order of subobject initialization. The extensions to initializers described in chapter 3 prescribe an ordering for the implicit assignments to the subobjects that comprise the unnamed object.

5. This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char[]){"/tmp/fileXXXXXX"}
```

The first always has static storage duration and has type array of **char**, but need not be modifiable: the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

Like string literals, const-qualified compound literals can be be placed into read-only memory and can even be shared. For example,

```
(const char[]){"abc"} == "abc"
```

might yield 1 if the literals' storage is shared.

Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object **endless_zeros** below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

Chapter 5

Floating-Point C Extensions

Table Of Contents

Foreword 5-v

Introduction 5-1
- Purpose 5-1
- Scope 5-1
- References 5-1
- Organization of Document 5-3
- Definition of Terms 5-4
- Compliance 5-6

Environment 5-7
- Translation 5-9
- Execution 5-9
  - Startup 5-9
  - Changing the environment 5-10
- Environmental Limits 5-10
  - Characteristics of floating types 5-10

Language 5-12
- Lexical Elements 5-12
  - Types 5-12
    - NaNs 5-13
  - Floating constants 5-14
    - Hexadecimal floating constants 5-14
- Conversions 5-16
  - Floating and integral 5-16
  - Floating types 5-17
  - Expression evaluation 5-17
    - Methods 5-18
    - Contraction operators 5-20
  - Wide representation 5-21
- Expressions 5-23
  - The arithmetic operators 5-23
  - Relational and equality operators 5-23
- Constant Expressions 5-23
- Initialization 5-24
- Predefined Macro Names 5-25

Library 5-26
- Introduction 5-26
- Standard C Library Extensions 5-26
  - General utilities <stdlib.h> 5-26
    - The atof function 5-26
    - The strtof, strtod, and strtold functions 5-26
  - Input / output <stdio.h> 5-28
    - The fprintf function 5-28
    - The fscanf function 5-30

Floating-Point Extensions <fp.h>....5-30
Overloaded functions....5-37
Trigonometric functions....5-40
The acos function....5-40
The asin function....5-40
The atan function....5-40
The atan2 function....5-40
The cos function....5-41
The sin function....5-41
The tan function....5-41
Hyperbolic functions....5-41
The acosh function....5-41
The asinh function....5-41
The atanh function....5-42
The cosh function....5-42
The sinh function....5-42
The tanh function....5-42
Exponential and logarithmic functions....5-42
The exp function....5-43
The exp2 function....5-43
The expm1 function....5-43
The frexp function....5-43
The ldexp function....5-43
The log function....5-44
The log10 function....5-44
The log1p function....5-44
The log2 function....5-44
The logb function....5-44
The modf functions....5-45
The scalb function....5-45
Power and absolute value functions....5-46
The fabs function....5-46
The hypot function....5-46
The pow function....5-46
The sqrt function....5-46
Error and gamma functions....5-46
The erf function....5-47
The erfc function....5-47
The gamma function....5-47
The lgamma function....5-47
Nearest integer functions....5-48
The ceil function....5-48
The floor function....5-48
The nearbyint function....5-48
The rint function....5-49
The rinttol function....5-49
The round function....5-49
The roundtol function....5-50
The trunc function....5-50
Remainder functions....5-50
The fmod function....5-50
The remainder function....5-51
The remquo function....5-51

12"

Manipulation functions 5-51
The copysign function 5-52
The nan functions 5-52
The nextafter functions 5-53
Maximum, minimum, and positive difference functions 5-53
The fdim function 5-54
The fmax function 5-54
The fmin function 5-54
Floating-Point Environment <fenv.h> 5-55
Exceptions 5-56
The feclearexcept function 5-57
The fegetexcept function 5-57
The feraiseexcept function 5-57
The fesetexcept function 5-58
The fetestexcept function 5-58
Rounding 5-58
The fegetround function 5-59
The fesetround function 5-59
Environment 5-60
The fegetenv function 5-60
The feholdexcept function 5-60
The fesetenv function 5-61
The feupdateenv function 5-61

Appendixes 5-62
Language Syntax Extensions 5-62
Constants 5-62
Preprocessor Numbers 5-62
Pragmas 5-62
Optimization 5-63
Expression evaluation 5-63
Expression transformations 5-63
Relational operators 5-64
Constant arithmetic 5-65
Wide representation 5-65
Library Summary 5-65
General utilities <stdlib.h> 5-66
Input/output <stdio.h> 5-66
Floating-point <fp.h> 5-66
Floating-point environment <fenv.h> 5-67
Implementation Limits 5-68
Operator / Function Documentation Guide 5-68
<fp.h> for IEEE Implementations 5-70
Trigonometric functions 5-71
The acos function 5-71
The asin function 5-71
The atan function 5-71
The atan2 function 5-71
The cos function 5-72
The sin function 5-72
The tan function 5-72

125

Hyperbolic functions....5-72
The acosh function....5-72
The asinh function....5-72
The atanh function....5-73
The cosh function....5-73
The sinh function....5-73
The tanh function....5-73
Exponential and logarithmic functions....5-73
The exp function....5-73
The exp2 function....5-73
The expm1 function....5-73
The frexp function....5-74
The ldexp function....5-74
The log function....5-74
The log10 function....5-74
The log1p function....5-74
The log2 function....5-75
The logb function....5-75
The modf functions....5-75
The scalb function....5-75
Power and absolute value functions....5-76
The fabs function....5-76
The hypot function....5-76
The pow function....5-76
The sqrt functions....5-77
Error and gamma functions....5-77
The erf function....5-77
The erfc function....5-77
The gamma function....5-77
The lgamma function....5-77
Nearest integer functions....5-77
The ceil function....5-77
The floor function....5-78
The nearbyint function....5-78
The rint function....5-78
The rinttol function....5-78
The round function....5-78
The roundtol function....5-79
The trunc function....5-79
Remainder functions....5-79
The fmod function....5-79
The remainder function....5-80
The remquo function....5-80
Manipulation functions....5-80
The copysign function....5-80
The nan functions....5-80
The nextafter functions....5-80
Maximum, minimum, and positive difference functions....5-81
The fdim function....5-81
The fmax function....5-81
The fmin function....5-81

Foreword

Following is a specification for C extensions designed to address two long-standing needs of programmers: (1) more predictable floating-point arithmetic for all implementations, and (2) a standard language binding for features of IEEE arithmetic.

Vagaries of floating-point arithmetic have plagued programmers and users since its inception. And they still do, even though hardware floating-point is now largely standardized. When IEEE binary floating-point standard 754 became an official standard in July 1985, 26 months before the radix-independent standard 854, several IEEE implementations were already shipping. Now virtually all new floating-point implementations conform to the IEEE standards—at least in format, if not to the last detail. Although these standards have been enormously successful in influencing hardware implementation, many of their features, including predictability, remain impractical or unavailable for use by programmers. The IEEE standards do not include language bindings—a cost of delivering the basic standard in a timely fashion. The ANSI C committee attempted to remove conflicts with IEEE arithmetic, but did not specify IEEE support. Expediencies of programming language implementation and optimization can deny the features offered by modern hardware. In the meantime, particular companies have defined their own IEEE language extensions and libraries [4, 8, 10]; not surprisingly, lack of portability has impeded programming for these interfaces.

The Numerical C Extensions Group, NCEG, at its initial meeting in May 1989, identified support for IEEE floating-point arithmetic as one of its focus areas and organized a subgroup to produce a technical report. The subgroup benefited from the considerable C language and IEEE floating-point expertise associated with NCEG. It included individuals with substantial experience with language extensions (albeit proprietary) for IEEE floating-point. And, following after the standardization of C, it had a stable, well defined base for its extensions. Thus NCEG had a unique opportunity to solve this problem.

The initial version of this document, in August 1989, was organized foremost as a specification for IEEE implementations, with notes for other implementations. However, from the beginning, substantial portions of the specification were not specific to IEEE floating-point. For broader utility, the goal was extended to address predictability for all implementations, and the document was reorganized as a general floating-point specification with additional specification for IEEE implementations.

Major milestones for "Floating-Point C Extensions" include:

Doc. Date	Draft No.	Milestone
May 92	X3J11.1/92-040	Distributed for preliminary review to various professional organizations, including X3J11, WG14, X3J16/WG21, X3T2, and X/Open
Jan 93	X3J11.1/93-001	Distributed for public comment
Jun 93	X3J11.1/93-028, WG14/N291, X3J11/93-037	Forwarded by NCEG/X3J11.1 to X3J11 as its FP/IEEE technical report
Jan 94	WG14/N319, X3J11/94-003	Draft 2 approved by NCEG/X3J11.1

Jun 94	WG14/N350, X3J11/94-035	Draft 3 accepted by X3J11 as part of its technical report on numerical C extensions
Dec 94	this document	Draft 4 accepted by X3J11 as an update to Draft 3

NCEG/X3J11.1 and X3J11 mailings have included thirteen drafts of this document, which have been reviewed to varying degrees by NCEG's FP/IEEE subgroup, NCEG, X3J11, and numerous other interested parties. Proprietary extensions for IEEE support have provided prior art for many features. The specification has been implemented in both developmental and commercial compilers and libraries [13, 28, 29].

People who have made especially substantial contributions to this document include, in alphabetical order: Jerome Coonen, Bill Gibbons, David Hough, Rex Jaeschke, W. Kahan, Clayton Lewis, Stuart McDonald, Colin McMaster, Rick Meyers, David Prosser, and Fred Tydeman.

Others who have provided invaluable contributions, reviews, or administrative help include, in alphabetical order: Joel Boney, Norris Boyd, Larry Breed, Walter Bright, W. J. Cody, Elizabeth Crockett, Karen Deach, James Demmel, Fred Dunlap, Clive Feather, Yinsun Feng, Samuel Figueroa, Paul Finlayson, James Frankel, Scott Fraser, David Gay, Eric Grosse, Ron Guilmette, Doug Gwyn, Bill Homer, Kenton Hanson, Paul Hilfinger, Martha Jaffe, Bob Jervis, David Keaton, Earl Killian, David Knaak, John Kwan, Roger Lawrence, Tom MacDonald, Michael Meissner, Randy Meyer, Randy Meyers, Antonine Mione, Stephen Moshier, Jon Okada, Conor O'Neill, Tom Pennello, Tim Peters, P.J. Plauger, Thomas Plum, Sanjay Poonen, Pat Ricci, Ali Sazegari, Roger Schlafly, Steve Sommars, Richard Stallman, Linda Stanberry, Gordon Sterling, Bill Torkelson, and Terrence Yee.

This document benefited from the author's experience with numerics and languages groups at Apple Computer, Inc., developing the Standard Apple Numerics Environment (SANE) and its various language bindings, and at Taligent, Inc., implementing this specification.

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
jim_thomas@taligent.com

March 29, 1995

Chapter 5
Floating-Point C Extensions

1. INTRODUCTION

1.1 Purpose

This document specifies:

1. a set of Standard C extensions, suitable for most implementations, and designed to facilitate a wide range of numerical programming;

2. a further set of extensions and definitions, suitable for implementations that support IEEE floating-point standards 754 and 854, and designed to provide full access to the features of those standards—allowing access also to similar features in non-IEEE implementations.

This document has been approved by the ANSI C committee (X3J11) for incorporation into its Technical Report on Numerical C Extensions.

1.2 Scope

The specifications of this document, while extending Standard C, still lie within the scope of that standard ([17] §4.2; [1] §1.2).

This document does not:

1. describe Standard C, except where it relates to extensions or subtleties of implementation, but instead refers implicitly to the Standard C documents [17], [1];

2. describe IEEE floating-point per se, but instead refers implicitly to the 754 and 854 standards documents;

3. address other NCEG areas, such as complex arithmetic, which are expected to accommodate IEEE standard arithmetic.

1.3 References

1. *American National Standard for Information Systems—Programming Language C* (X3.159-1989).

2. *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985). Also, *Binary floating-point arithmetic for microprocessor systems* (IEC 559:1989).

(CONTINUED w/ 209)

129

3. *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE Std 854-1987).

4. *Apple Numerics Manual*. Second Edition. Addison-Wesley (1988).

5. "Functions to Support the IEEE Standard for Binary Floating-Point Arithmetic", by W. J. Cody and Jerome T. Coonen. *Transactions on Mathematical Software*. Vol. 19, No. 4. Pages 443-451 (December 1993).

6. "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic", by Jerome Coonen. PhD thesis. University of California. Berkeley (1984).

7. "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit", by W. Kahan. Proceedings of the joint IMA/SIAM conference on The State of the Art in Numerical Analysis, 14-18 April 1986. Clarendon Press (1987).

8. *Numerical Computation Guide*. Sun Microsystems. Inc. (March 16. 1990).

9. "Enhanced Arithmetic for Fortran", by R. P. Corbett. *ACM Signum Newsletter*. Vol. 18. No. 1 (January 1983) and *ACM Sigplan Newsletter*. Vol. 17. No. 12 (December 1982).

10. *UNIX System V/386 Release 4 Programmer's Reference Manual*. Prentice Hall (1990).

11. "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic", by W. J. Cody et al. *IEEE Micro*, Vol. 4, No. 4 (August 1984).

12. "Correctly Rounded Binary-Decimal and Decimal-Binary Conversion", by David M. Gay, Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories (November 30. 1990)—NCEG 91-019.

13. *Numerics Programming Guide*. Zortech C++ Compiler 3.0. Zortech Limited (May 1991). A pioneer implementation of substantial portions of this specification.

14. "Overloading Floating-Point Functions in C", Bill Gibbons (November 1991)—NCEG 91-047.

15. "Generic Functions", David Hough (November 1991)—NCEG 91-046.

16. *American National Standard Programming Language FORTRAN* (ANSI X3.9-1978).

17. *International Standard Programming Languages—C*, (ISO/IEC 9899:1990 (E)).

18. *Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++* (X3J16/94-0098, WG21/N0485).

19. "Contracted Multiply-Adds", W. Kahan (September 20, 1991)—NCEG 91-042.

20. "An Argument of Apple's Support of the Extended Floating-Point Type", Jim Thomas and Jerome Coonen (September 14, 1989)—NCEG 89-037.

21. *C* Language Reference Manual*. James L. Frankel. Draft Technical Report. Thinking Machines Corporation (May 16. 1991)—NCEG 91-023.

22. "Augmenting a Programming Language with Complex Arithmetic". W. Kahan and J. W. Thomas (November 15. 1991)—NCEG 91-039.

23. "Algorithm XXX: Specfun—A Portable Package of Special Functions and Test Drivers". W. J. Cody (work in progress).

24. "Floating-Point C Extensions". Jim Thomas (August 10. 1991)—NCEG 91-028.

25. "Amendment #1 to ISO C standard" (ISO/IEC 9899:1990/Amendment 1:1994(E)).

26. "A Rationale for Floating-Point C Extensions". Jim Thomas (April 23. 1994)—WG14/N338. X3J11/94-023.

27. "An Introduction to Floating-Point C Extensions", Jim Thomas and Jerome T. Coonen. *C/C++ Users Journal*. Vol. 13. No. 1. Pages 49-57 (January 1995).

28. *Inside Macintosh: PowerPC Numerics*. Addison-Wesley (March 1994).

29. *CommonPoint Application System*. Foundation Services. Part 3. Numerical computation. Taligent. Inc. (1995).

1.4 Organization of Document

The major subsections are:

1. this introduction;
2. characteristics of the translation and execution environments;
3. language syntax, constraints, and semantics;
4. library facilities;

A. summary of language syntax extensions;

B. optimization notes;

C. summary of library facilities;

D. implementation limits;

E. documentation guide for operators and functions;

F. specification of library facilities for IEEE implementations.

Sections that can be regarded as extending or modifying a particular section of the Standard C documents are marked with the section numbers from the ISO C [17] and X3.159 [1] documents, for example

3.1.2 Floating constants (ISO §6.1.3.1; X3.159 §3.1.3.1)

Rationale, in smaller type size, accompanies sections that are controversial or where further explanation seems needed. Subsections entitled "For IEEE Implementations" contain specification that applies specifically to implementations that support features of the IEEE floating-point standards.

As in the IEEE floating-point standards, use of the word *shall* signifies that which is obligatory for conformance; use of the word *should* signifies that which is strongly recommended, though impractical for certain implementations.

1.5 Definition of Terms

- Double—the IEEE floating-point standards' double data format. Written in computer-voice style, `double` refers to the Standard C data type. Where noted, double refers locally to both IEEE and also non-IEEE double precision formats.
- Double-based architecture—a floating-point architecture designed to produce primarily double format results.
- Evaluation format—the format used to represent the result of an expression. It may be wider than the semantic type of the expression.
- Exception flag—a flag signifying that a floating-point exception has occurred. IEEE implementations support overflow, underflow, invalid, divide-by-zero, and inexact exception flags.
- Expression evaluation method—a method for determining the evaluation formats for expressions.
- Extended—the IEEE floating-point standards' *double-extended* data format. Extended refers to both the common 80-bit and so-called *quadruple* 128-bit IEEE formats. Where noted, *extended* refers locally to both IEEE and also non-IEEE formats that are wider than double.
- Extended-based architecture—a floating-point architecture designed to produce primarily extended format results.
- Flag—see *exception flag*.
- Floating-point environment—collectively all floating-point status flags and control modes.
- Floating-point standards—specifically *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985) and *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE Std 854-1987).
- IEEE implementation—an implementation that conforms to one or both of the IEEE floating-point standards. See *floating-point standards*.

- Implementation-defined behavior—behavior that depends on the characteristics of the implementation and that each implementation shall document. (As in Standard C.)
- Mode—short for floating-point control mode.
- NaN—in a floating-point format, an encoding that signifies *Not-a-Number*. In this document the term NaN generally denotes quiet NaNs.
- Normal number—a nonzero, finite number that is not subnormal. (As in IEEE standard 854.)
- Quiet NaN—a NaN that propagates through almost every arithmetic operation without raising an exception.
- Quiet operation or function—an operation or function that does not raise an exception.
- Rounding direction mode—a dynamic mode that controls the style of rounding used by floating-point operations. For IEEE implementations the rounding directions are upward, downward, toward zero, and to nearest.
- Rounding precision mode—a dynamic or static mode that controls the precision to which floating-point operations round their results.
- Signaling NaN—a NaN that generally raises an exception when occurring as an arithmetic operand. This specification does not cover signaling NaNs.
- Single—the IEEE floating-point standards' single data format. Where noted, *single* refers locally to both IEEE and also non-IEEE single precision formats.
- Single/double architecture—a floating-point architecture designed to produce with like efficiency either single or double format results, with any extended arithmetic being substantially slower.
- Single/double/extended architecture—a floating-point architecture designed to produce with like efficiency either single, double, or extended results.
- Standard C—the C language standard, *International Standard ISO/IEC 9899* and *American National Standard Programming Language C* .
- Subnormal number—a nonzero floating-point number whose exponent is the format's minimum and whose leading significand digit is zero. (As in IEEE standard 854.)
- Undefined behavior—behavior, for an erroneous program or erroneous data, for which this specification imposes no requirement. (As in Standard C.)
- Unspecified behavior—behavior, for a correct program and correct data, for which this specification imposes no requirement. (As in Standard C.)
- Warning—a translation-time message alerting the user when reasonable or common expectations for a formally correct program may not be met.

1.6 Compliance

In order to comply as an implementation supporting IEEE standard floating-point, an implementation must conform to one, or both, of the IEEE standards 754 and 854—which ones is implementation-defined.

2. ENVIRONMENT

Any floating-point status flags and control modes supported by the implementation are here referred to collectively as the *floating-point environment*. Programs that test flags or run under non-default modes must do so under the effect of an enabling `fenv_access` pragma.

```
#pragma fenv_access on-off-switch
```

on-off-switch: one of:

```
on      off      default
```

can occur outside external declarations and takes effect from its occurrence until another `fenv_access` pragma is encountered, or until the end of the translation unit. The effect of this pragma appearing inside an external declaration is undefined. If part of a program tests flags or runs under non-default mode settings, but is not under the effect of an enabling `fenv_access` pragma, then the behavior of that program is undefined. (Part of a program is under the effect of an enabling `fenv_access` pragma if that part is translated while the state of the pragma is *on*.) The default state (*on* or *off*) for the pragma is implementation-defined. (§4.4 specifies facilities for accessing the floating-point environment.)

The floating-point environment as defined here includes only execution-time modes, not the myriad of possible translation-time options that may affect a program's results. Examples of such translation-time options include: chopped or rounded multiplication on CRAY Y-MP systems, D or G format for VAX, and fast or correctly-rounded divide on the Intel 860. Each option's implementation-defined or deviant properties, relative to this specification, should be well documented.

The purpose of the `fenv_access` pragma is to allow certain optimizations, namely *global common subexpression elimination*, *code motion*, and *constant folding*, that could subvert the testing of flags and changing of modes. For example, in

```
{
   double x;
   void f(double);
   void g(double);
   ...
   f(x + 1);
   g(x + 1);
   ...
}
```

the function `f` might depend on status flags set as a side effect of the first `x + 1`. The second `x + 1` might depend on control modes set as a side effect of the function call `f`. The imposed temporal ordering would require two evaluations of `x + 1`. This specification says that if in fact the program tests flags or changes modes through the call to `f`, it must so declare with an enabling `fenv_access` pragma. In general, if the state of `fenv_access` is *off* then the translator can assume that default modes are in effect and the flags are not tested.

The user model is supported by certain programming conventions:

1. A function call must not alter its caller's modes, clear its caller's flags, nor depend on the state of its caller's flags unless the function is so documented. (To do otherwise would be extremely dangerous, irrespective of this specification.)

2. A function call is assumed to require default modes, unless its documentation promises otherwise, or unless the function is known not to use floating-point.

3. A function call is assumed to have the potential of raising floating-point exceptions, unless its documentation promises otherwise, or unless the function is known not to use floating-point.

With these conventions a programmer can assume default modes, or ignore them altogether, safely and without code or intellectual overhead. Responsibilities, and ordinarily modest overhead, associated with accessing the floating-point environment fall entirely on the programmer or program that does so.

No standard library function, except those in `<fenv.h>`, tests or clears its caller's flags or changes its caller's modes.

Libraries are encouraged to document their use, or non-use, of floating-point and their raising of floating-point exceptions.

The performance of code under the effect of an enabling `fenv_access` pragma may well be important: in fact, an algorithm may access the floating-point environment specifically for performance. The implementation should optimize as aggressively as the `fenv_access` pragma allows. (See §B.1.)

An implementation could simply honor the floating-point environment in all cases and ignore the pragma.

Dynamic vs static modes

Dynamic modes are potentially problematic because

1. the programmer may have to defend against undesirable mode settings—which imposes intellectual, as well as time and space, overhead.

2. the translator may not know which mode settings will be in effect or which functions change them at execution time—which inhibits optimization.

This proposal attempts to address these problems without changing the dynamic nature of the modes.

An alternate approach would have been to present a model of *static modes*, with explicit utterances to the translator about what mode settings would be in effect. This would have avoided any uncertainty due to the global nature of dynamic modes or the dependency on unenforced conventions. However, some essentially dynamic mechanism still would have been needed in order to allow functions to inherit (honor) their caller's modes. The IEEE standards require dynamic rounding direction modes. For the many architectures that maintain these modes in control registers, implementation of the static model would be more costly. Also, Standard C has no facility, other than pragmas, for supporting static modes.

An implementation on an architecture that provides only static control of modes, for example through opword encodings, still could support the dynamic model, by generating multiple code streams with tests of a private global variable containing the mode setting. Only modules under an enabling `fenv_access` pragma would need such special treatment. To further limit the problem, the implementation might employ additional pragmas specifically to indicate where non-default modes would be admissible.

For IEEE Implementations

The IEEE floating-point standards require that an implementation support certain status flags and control modes.

The *exception flags*—invalid, overflow, underflow, divide-by-zero, and inexact—can be queried at execution time to determine whether a floating-point exception has occurred since the beginning of execution or since its flag was explicitly cleared. (The flags are *sticky*.)

The *rounding direction modes*—to-nearest, toward-zero, upward (toward $+\infty$), and downward (toward $-\infty$)—can be altered at execution time to control the rounding direction for floating-point operations.

The *rounding precision modes* cause a system whose results are always double or extended to shorten the precision of its results, in order to mimic systems that deliver results to single or double precision. Some systems need not implement rounding precision modes—see §4.4.2.

The traps recommended by the IEEE floating-point standards require modes for enabling and disabling. Trap-enable modes lie outside the scope of this document.

2.1 Translation

An implementation should provide a warning for each translation-time floating-point exception, other than inexact. The implementation shall document all mode settings that affect the behavior of translation-time arithmetic.

For IEEE Implementations

During translation the IEEE standard default modes are in effect:

rounding direction:	to-nearest
rounding precision:	results not shortened
trapping:	all traps disabled

An implementation is not required to provide a facility for altering the modes for translation-time arithmetic, or for making exception flags from the translation available to the executing program. The language and library provide facilities to cause floating-point operations to be done at execution time, when they can be subjected to varying dynamic modes and their exceptions detected. The need does not seem sufficient to require similar facilities for translation.

2.2 Execution

2.2.1 Startup

At program startup any floating-point modes are initialized as for translation-time arithmetic. The implementation shall document all startup mode settings that affect the behavior of arithmetic.

This promotes consistency between translation- and execution-time computations.

For IEEE Implementations

At program startup the floating-point environment is initialized as prescribed by the IEEE floating-point standards:

exception flags:	all clear
rounding direction:	to-nearest
rounding precision:	results not shortened
trapping:	all traps disabled

2.2.2 Changing the environment

Operations defined in ISO §6.3 (X3.159 §3.3) and functions and macros defined for standard libraries change flags and modes just as indicated by their specification (which may include conformance to standards); they do not change flags or modes (so as to be detectable by the user) in any other cases.

2.3 Environmental Limits

2.3.1 Characteristics of floating types (ISO §5.2.4.2; X3.159 §2.2.4.2)

The value of the Standard C macro

```
FLT_ROUNDS
```

is dynamically determined to represent the effective rounding direction.

> Thus IEEE implementations, and any others that allow changing the rounding direction at execution time, must implement `FLT_ROUNDS` as an execution-time inquiry, not as a constant (when the state of `fenv_access` is *on*).

This specification augments `<float.h>` with two macros that characterize the evaluation method (§3.2.3.1) for evaluating floating-point expressions. The minimum evaluation format is characterized by the value of `_MIN_EVAL_FORMAT`:

```
0  float
1  double
2  long double
```

Whether widest-need evaluation is performed is characterized by `_WIDEST_NEED_EVAL`:

```
0  no
1  yes
```

> If the minimum evaluation format is `long double` then the value of `_WIDEST_NEED_EVAL` is irrelevant.

The macros `_MIN_EVAL_FORMAT` and `_WIDEST_NEED_EVAL` evaluate to constant expressions suitable for use in `#if` preprocessing directives.

An implementation that provides alternate expression evaluation methods must assure that translation-time evaluation of `_MIN_EVAL_FORMAT` and `_WIDEST_NEED_EVAL` reflects the current one.

3. LANGUAGE

3.1 Lexical Elements (ISO §6.1; X3.159 §3.1)

3.1.1 Types (ISO §6.1.2.5; X3.159 §3.1.2.5)

The `long double` type should have strictly more precision than `double` which should have at least twice the number of digits of precision as `float`. If not, the implementation should emit a warning when processing a translation unit that uses distinct floating types with the same precision.

This facilitates porting code that, intentionally or not, depends on differences in type widths. Many results are exact or correctly rounded when computed with twice the number of digits of precision as the data. For example, the calculation

```
float d, x, y, z, w;
d = (double) x * y - (double) z * w;
```

yields a correctly rounded determinant if `double` has twice the precision of `float` and the individual operations are correctly rounded. (The casts to `double` are unnecessary if the minimum evaluation format is `double` or `long double`—see §3.2.3.1.)

For IEEE Implementations

The C floating types match the IEEE standard floating-point formats as follows:

C type	IEEE format
`float`	single
`double`	double
`long double`	extended, else a non-IEEE extended format, else double

Any non-IEEE extended format, used as the `long double` format for an IEEE implementation, has more precision than IEEE double and at least the range.

Minimal conformance to the IEEE floating-point standards does not require a format wider than single. The narrowest C `double` type allowed by Standard C (ISO §5.2.4.2; X3.159 §2.2.4.2) is wider than IEEE single, and wider than the minimum IEEE *single-extended* format. (IEEE single-extended is an optional format intended only for those implementations that don't support double; it has at least 32 bits of precision.) Both Standard C and the IEEE standards would be satisfied if `float` were IEEE single and `double` were an IEEE single-extended format with at least 35 bits of precision. However, this specification goes slightly further by requiring `double` to be IEEE double rather than just a wide IEEE single-extended.

The primary objective of the IEEE part of this specification is to facilitate writing portable code that exploits the IEEE floating-point standards, including their standardized single and double data formats. Bringing the C data types and the IEEE standard formats into line advances this objective.

This specification accommodates what are expected to be the most important IEEE floating-point architectures for general C implementations—see §3.2.3.

Because of Standard C's bias toward `double`, extended-based architectures might appear to be better served by associating the C `double` type with IEEE extended. However, such an approach would not allow standard C types for both IEEE double and single and would go against current industry naming, in addition to undermining this specification's portability goal. Other features in this document, for example the type definitions `float_t` and `double_t` (defined in §4.3), are intended to allow effective use of architectures with more efficient, wider formats.

The `long double` type is not required to be IEEE extended because

1. some of the major IEEE floating-point architectures for C implementations do not support extended,
2. double precision is adequate for a broad assortment of numerical applications, and
3. extended is less standard than single or double in that only bounds for its range and precision are specified in IEEE standard 754.

For implementations without extended in hardware, non-IEEE extended arithmetic written in software, exploiting double in hardware, provides some of the advantages of IEEE extended but with significantly better performance than true IEEE extended in software. [20] explains advantages of extended precision.

Specification for a variable-length extended type—one whose width could be changed by the user—was deemed premature. However, not unduly encumbering experimentation and future extensions, for example for variable length extended, is a goal of this specification.

Narrow-format Implementations

Some C implementations, namely ones for digital signal processing, provide only the IEEE floating-point standards' single format, possibly augmented by single-extended, which may be narrower than IEEE double or Standard C `double`, and possibly further augmented by double in software. These non-conforming implementations might generally adopt this specification, though not matching its requirements for types.

One approach would be: match Standard C `float` with single; match Standard C `double` with single-extended, else single; and match Standard C `long double` with double, else single-extended, else single. Then most of this specification could be applied straightforwardly. Users should be clearly warned that the types may not meet expectations.

Another approach would be to refer to a single-extended format as `long float` and then not recognize any C types not truly supported. This would provide ample warning for programs requiring double. The translation part of porting programs could be accomplished easily with the help of type definitions. In the absence of a double type, most of this specification for `double` could be adopted for the `long float` type. Having distinct types for `long float` and `double`, previously synonyms, requires more imagination.

3.1.1.1 NaNs

Floating types may support *NaN* (Not-a-Number) values, which do not represent numbers. A NaN that generally raises an exception when encountered as an operand of arithmetic operations is called a *signaling NaN*, and the operation is said to *trigger* the signaling NaN; this document does not define the behavior of signaling NaNs. A NaN that behaves predictably and does not raise exceptions in arithmetic operations is called a *quiet NaN*; this document uses the term *NaN* to denote quiet NaNs.

The IEEE floating-point standards specify quiet and signaling NaNs, but these terms can be applied for some non-IEEE implementations as well—for example, the VAX *reserved operand* and the CDC and CRAY *indefinite* qualify as signaling NaNs. In IEEE standard arithmetic, operations that trigger a signaling NaN argument generally return a quiet NaN result provided no trap is taken; neither traps nor any other facility for signaling NaNs is required. True support for signaling NaNs implies restartable traps, such as the optional traps specified in the IEEE floating-point standards.

The primary utility of quiet NaNs—"to handle otherwise intractable situations, such as providing a default value for 0.0/0.0" [11]—can be supported through straightforward library extensions to C. See §4.2.1.2, 4.2.2.1-2, 4.3, 4.3.9.2.

Other applications of NaNs may prove useful. Available parts of NaNs have been used to encode auxiliary information, for example about the NaN's origin [4]. Signaling NaNs are good candidates for filling uninitialized storage; and their available parts could distinguish uninitialized floating objects. IEEE signaling NaNs and trap handlers potentially provide hooks for maintaining diagnostic information or for implementing special arithmetics.

However, C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways. The IEEE floating-point standards require that NaNs propagate, but not all implementations faithfully propagate the entire contents. And even the IEEE standards fail to specify the contents of NaNs through format conversion, which is pervasive in some C implementation mechanisms. For these reasons this document does not define the behavior of signaling NaNs nor specify the interpretation of NaN significands.

[24], a previous version of this document, contains specification for signaling NaNs. It could serve as a guide for implementation extensions in support of signaling NaNs.

3.1.2 Floating constants (ISO §6.1.3.1; X3.159 §3.1.3.1)

Floating constants are converted to their internal representation at translation time. Each constant is represented in a format (perhaps wider than required by the constant's type) determined by the effective expression evaluation method (§3.2.3.1). The resulting values from translation-time conversion of (valid) floating constants match those from execution-time conversion with default rounding modes by library functions, like `strtod`. See §4.2.1.2.

Note that since floating constants are converted to appropriate internal representations at translation time, default rounding direction and precision will be in effect and execution-time exceptions will not be raised, even under the effect of an enabling `fenv_access` pragma. Library functions, for example `strtod`, provide execution-time conversion of decimal strings.

3.1.2.1 Hexadecimal floating constants (ISO §6.1.3.1, 6.1.8; X3.159 §3.1.3.1, 3.1.8)

The Standard C *floating-constant* syntax is augmented to include hexadecimal floating constants.

Syntax

floating-constant:
...
hexadecimal-floating-constant

hexadecimal-floating-constant:
`0x` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
`0X` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
`0x` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$
`0X` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{opt}$. *hexadecimal-digit-sequence*
hexadecimal-digit-sequence .

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

binary-exponent-part:
`p` *sign*$_{opt}$ *digit-sequence*
`P` *sign*$_{opt}$ *digit-sequence*

The binary exponent gives a decimal integer representing a power of 2. If `FLT_RADIX` is a power of two, hexadecimal floating constants are correctly rounded. The implementation shall emit a warning if a hexadecimal constant cannot be represented exactly in its evaluation format (see §3.2.3.1). Further accuracy specification for the conversion of hexadecimal floating constants to internal format is implied by the specification for the `strtod` functions (§4.2.1.2).

In order to accommodate hexadecimal floating constants, the Standard C syntax for preprocessing numbers (ISO §6.1.8; X3.159 §3.1.8) is augmented to include

pp-number:
pp-number `p` *sign*
pp-number `P` *sign*

Example

`<float.h>` for an IEEE 754 implementation might contain:

```
#define  FLT_MAX      0x1.FFFFFEp127f  /* or 0xF.FFFFFp124f */
#define  FLT_EPSILON  0x1p-23f
#define  FLT_MIN      0x1p-126f
```

Example

The constant `-0x1.0000000000001p0` represents the largest IEEE 754 double precision value less than -1.

Hexadecimal more clearly expresses the significance of floating constants.

The binary-exponent part is required to avoid ambiguity from an `f` suffix (being mistaken as a hexadecimal digit).

Constants of `long double` type are not generally portable, even among IEEE implementations.

Unlike integers, floating values cannot all be represented directly by hexadecimal constant syntax. A sign can be prefixed for negative numbers and -0. Infinities might be produced by hexadecimal constants that overflow. NaNs require some other mechanism. Note that `0x1.FFFFFEp128f`, which might appear to be an IEEE single-format NaN, in fact overflows to an infinity in the single format (and causes a translation-time warning).

Infinity and NaN constants, useful for static and aggregate initialization, should be considered for future extensions.

An alternate approach might have been to represent bit patterns. For example

```
#define     FLT_MAX         0x.7F7FFFFF
```

This would have allowed representation of NaNs and infinities. However, numerical values would have been more obscure owing to bias in the exponent and the implicit significand bit. NaN representations would not have been portable—even the determination of IEEE quiet NaN vs signaling NaN is implementation-defined. NaNs and infinities are provided through macros in §4.3.

The straightforward approach of denoting octal constants by a `0` prefix would have been inconsistent with allowing a leading `0` digit—a moot point as the need for octal floating constants was deemed insufficient.

The caret ^ was ruled out as a character to introduce the exponent because doing so would have used up a potential operator.

3.2 Conversions

3.2.1 Floating and integral (ISO §6.2.1.3; X3.159 §3.2.1.3)

For conversion from floating to integer type, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type then the invalid exception, if available, is raised and the value of the result is unspecified.

For IEEE Implementations

Whether conversion of nonintegral floating values whose integral part is within the range of the integer type raises the inexact exception is unspecified.

Conversion from floating to integral rounds toward zero, consistent with Standard C. IEEE rounding is provided by the library function `rinttol`.

IEEE standard 854, though not 754, directly specifies that floating-to-integral conversion raise the inexact exception for nonintegral in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the inexact exception. See `rint`, `rinttol`, and `nearbyint` in §F.6.

Conversion from integer to floating type is an IEEE standard floating-point operation, rounding according to the current rounding direction mode.

Note that rounding indeed will be required if an integer is too wide to represent exactly in the floating-point format.

3.2.2 Floating types (ISO §6.2.1.4; X3.159 §3.2.1.4)

For IEEE Implementations

Conversions between floating types are specified by the IEEE floating-point standards.

3.2.3 Expression evaluation (ISO §6.2.1.5; X3.159 §3.2.1.5)

This specification recognizes five distinct *expression evaluation methods*. A floating-point expression has both a semantic type and an *evaluation format*. Each evaluation format is the format of one of the floating types. The result of the expression is represented in the expression's evaluation format. An expression evaluation method determines the evaluation formats for expressions. Which expression evaluation methods are provided is implementation-defined.

For most floating operations Standard C defines the semantic type of the operation to be the widest type of its operands, but gives explicit license to represent the operation's result in a format wider than its type.

The use here of the term *method* is unrelated to its use in Smalltalk.

Although specifying just one method would have facilitated porting code, any one method would have been unacceptably inefficient on some important architectures. On the other hand, still other expression evaluation methods are conceivable, for example evaluating `float` operations to `float` format, and all others to `long double`. The expression evaluation methods described in this section comprise an intentionally small set with at least one method that is efficient for any of the existing or anticipated, commercially significant, floating-point architectures:

Floating-point architectures

In the following description of floating-point architectures, the terms *extended*, *double*, and *single* have a slightly broader meaning than in the rest of this document. They still refer to floating formats, but apply to both IEEE and non-IEEE systems. Generally, extended is wider than double which is wider than single.

Extended-based. The arithmetic engine is extended. Source operands can be single, double, or extended, though generally arithmetic with single and double types is less efficient, requiring extra conversions. Examples include Intel 80x87, Cyrix 3D87, Motorola 6888x, and AT&T WE 32x06. The Motorola 88110 and Intel 960 can be used as extended-based architectures, or alternatively as single/double/extended ones (see below).

Double-based. The arithmetic engine is double. Source operands can be single or double, though generally arithmetic with the single type is less efficient, requiring extra conversions. Extended may be available, but implemented in software. Examples include IBM RISC System/6000, PDP-11 in double mode, CRAY, and CYBER 180. On CRAY and CYBER, single and double may be the same format. The CYBER provides some hardware support for extended.

Single/double. These provide orthogonal operations for single and double arithmetic. Single is typically faster than double. Extended may be available, but implemented in

software. Examples include MIPS, SPARC, HP PA-RISC, Motorola 88100, Intel 860, and systems assembled with Weitek or BIT processors. The MIPS, SPARC, and HP PA-RISC architectures specify extended, though it is not yet in hardware.

Single/double/extended. These provide orthogonal operations for single, double, and extended arithmetic. Single is faster than double, which is faster than extended. Examples include Motorola 88110, Intel 960, IBM S/370, and VAX.

3.2.3.1 Methods

The expression evaluation methods are characterized by two properties: (1) the minimum-width format (`float`, `double` or `long double`) for expression evaluation, and (2) whether determination of the evaluation format is affected by the context of the operation according to the rules for *widest-need* evaluation (explained below). The combinations of these two properties yield the five expression evaluation methods:

	min-eval-format	widest-need
1.	`float`	no
2.	`double`	no
3.	`long double`	<irrelevant>
4.	`float`	yes
5.	`double`	yes

Minimum evaluation format. The minimum evaluation format is one of `float`, `double`, or `long double`. The evaluation format for operations subject to the usual arithmetic conversions and for floating constants is at least this wide, even if the semantic type is less.

The early C implementations provided just the `float` and `double` floating-point types and evaluated all floating expressions to `double`. Intentional or not, some C programs have relied on extra precision for their computation with `float` operands. A minimum evaluation format of `double` is a natural choice for double-based architectures.

Implementations for single/double and single/double/extended architectures may find a minimum evaluation format of `float` compellingly more efficient, despite potential problems of conformity to expectations based on C's tradition of wide evaluation.

Even on a single/double or single/double/extended architecture, an implementation might provide double as a minimum evaluation format, for compatibility reasons. Common statements of the form

```
f1 = f2 op f3;      /* where f1, f2, f3 are of type float  */
```

can be done optimally by many such implementations, including all IEEE ones, where rounding the result to `double` and then to `float` is equivalent to rounding to `float` directly.

A minimum evaluation format of `long double` is common on extended-based architectures. Programs that run under one of the other expression evaluation methods generally run at least as well when all expressions are evaluated to `long double`. Most program failures due to extra precision arise from its inconsistent use (see §B.5).

Representation of constants in a format commensurate with expression evaluation, not a traditional practice, better meets certain expectations than would representation strictly according to semantic type—for example, `0.1f == 1.0f/10.0f`. Viewed as translation-

time operations that convert decimal strings to internal floating representations. literal floating constants naturally follow the method of expression evaluation.

Without widest-need evaluation. Without widest-need, the evaluation format for an operation or constant is simply the wider of its semantic type and the minimum evaluation format.

With widest-need evaluation. With widest-need, the evaluation format for an operation is the widest of the semantic types appearing in a certain enclosing expression and at least as wide as the minimum evaluation format. More precisely, the evaluation format for an operation subject to the usual arithmetic conversions, or for an assignment (including the assignment of function arguments to parameters, but not cast conversions), is propagated to its operands (or arguments): if an operand is a variable or an operation not subject to the usual arithmetic conversions it is converted to the evaluation format; if the operand is an operation subject to the usual arithmetic conversions, or a floating constant, the evaluation format is imposed recursively.

Examples

With the declarations

```
float f;
double d;
long double ld;
double dd(double);
double fd(float);
```

widest-need with a minimum evaluation format of `float` implies:

Outer-most expression	Operation	Evaluation format
`ld + (f * f)`	`*`	`long double`
	`+`	`long double`
`ld + fd(f * f)`	`*`	`float`
	`+`	`long double`
`ld + dd(f * f)`	`*`	`double`
	`+`	`long double`
`ld + (d = f * f)`	`*`	`double`
	`+`	`long double`
`f + (d = ld * ld)`	`*`	`long double`
	`+`	`double`
`ld + (double)(f * f)`	`*`	`float`
	`+`	`long double`
`ld + (d -= f * f)`	`*`	`double`
	`-`	`double`
	`+`	`long double`
`ld + (f++ * --d)`	`++`	`float`
	`--`	`double`
	`*`	`long double`
	`+`	`long double`
`ld == f * f`	`*`	`long double`

`f / (d + d, f * f)`	`+`	`double`
	`*`	`float`
	`/`	`float`
`f + (f - f ? d / d : ld)`	`-`	`float`
	`/`	`long double`
	`+`	`long double`
`ld + 0.1f`	`0.1f`	`long double`
	`+`	`long double`
`f + 0.1`	`0.1`	`double`
	`+`	`double`

The definition of widest-need is based on a *widest need* evaluation for Fortran presented in [9]. It does not affect integer expression evaluation, which is covered by Standard C.

As computer speed and memory size increase, so will the number of problems attempted and the size of data sets. Thus, the likelihood that a program will suffer serious roundoff error for some actual data will increase. Wider precision, not for the entire computation but just for expressions containing certain variables, often will fix the problem, without unduly affecting performance, and without requiring costly error analysis. With widest-need, an expression is automatically evaluated to the format of its widest component. To achieve the same effect without widest-need expression evaluation, the programmer must add or delete casts and constant suffixes throughout the program.

Widest-need expression evaluation is a particularly attractive compromise for architectures whose wider formats are significantly slower. It offers the accuracy of wide evaluation where likely needed and also the speed of narrow evaluation where clearly intended. Note that casts can be used to inhibit widest-need widening, even within a wide expression.

Previous drafts defined a `#pragma evaluate` which allowed switching expression evaluation methods between external declarations. This facility was believed to be without sufficient utility and somewhat error prone. Implementations that support multiple expression evaluation methods can supply translation options.

3.2.3.2 Contraction operators

A floating-point engine may provide an atomic operation for multiple binary operators. Such an atomic operation is referred to here as a *contraction* operator and the multiple binary operators as *contracted*, because rounding errors or side effects from their computation may be omitted. Contracted operators should incur no greater-magnitude rounding error than if they were not contracted. Whether contraction operators are employed is implementation-defined.

An implementation that is able to multiply two `double` operands and produce a `float` result in just one machine instruction might contract the multiplication and assignment in:

```
float f;
double d1, d2;
...
f = d1 * d2;
```

Other examples of potential contraction operators include:

compound assignments	`+=, -=,` etc.;
ternary add	`x + y + z;`
multiply-add	`x * y + z.`

Contractions can lead to subtle anomalies even while increasing accuracy. The value of expressions like `a * b + c * d` will depend on how the translator uses a contracted multiply-add. Knowing that the implementation contracts multiply-adds, the programmer should be able to control results (and reap the benefits of contraction) through simple coding measures, for example parenthesizing `(a * b) + c * d`. However, the Intel 860's multiply-add is slightly more problematic. Since it keeps a wide but partial product, `a * b + z` may differ from `c * d + z` even though the exact mathematical products `a * b` and `c * d` are equal; the result depends not just on the mathematical result and the format, as ordinarily expected for error analysis, but also on the particular values of the operands.

The extra accuracy of the IBM RISC System/6000's *fused* multiply-add, which produces a result with just one rounding, can be exploited for simpler and faster codes. See [19] for details.

The `fp_contract` pragma can be used to allow or disallow contraction operators.

`#pragma fp_contract` *on-off-switch*

This pragma can occur outside external declarations, and allows or disallows contraction operators from its occurrence until another `fp_contract` pragma is encountered, or until the end of the translation unit. The effect of this pragma appearing inside an external declaration is undefined. The default state (*on* or *off*) for the pragma is implementation-defined.

For IEEE Implementations

Contraction operators honor infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by the IEEE floating-point standards. Contraction operators should raise exceptions in a manner generally consistent with the basic arithmetic operations covered by the IEEE floating-point standards.

Contraction operators should deliver the same value as their uncontracted counterpart, else be correctly rounded—that is, deliver the infinitely precise result of their contracted operations, rounded (once) to the destination format according to the effective rounding method.

3.2.4 Wide representation

Three pragmas give license for the implementation to represent function return values, function parameters, and variables in a format wider than their declared type.

`#pragma fp_wide_function_returns` *on-off-switch*
`#pragma fp_wide_function_parameters` *on-off-switch*
`#pragma fp_wide_variables` *on-off-switch*

`#pragma fp_wide_function_returns on`—allows floating return values to be represented in a wider format than (and not narrowed to) the declared type of the function.

`#pragma fp_wide_function_parameters on`—allows floating parameters to be in a wider format than (and not narrowed to) the declared type of the formal parameter.

`#pragma fp_wide_variables on`—allows values of automatic scalar floating variables to be in a wider format than their declared type.

If the *on-off-switch* is `off` then widening is disallowed. The default state for these pragmas is *off*.

Standard C compatibility requires that the default state for the pragmas be *off*.

These pragmas can occur outside external declarations, and allow (if *on*) or disallow (if *off*) widening from their occurrence until another pragma instructing otherwise is encountered, or until the end of the translation unit. The effect of these pragmas appearing inside an external declaration is undefined. Widening is consistent throughout the effect of an enabling pragma: either all instances of the returns, parameters, or variables are widened or none are; the representation format for a parameter or variable does not vary. However, which, if any, of these pragmas actually cause widening is implementation-defined. The `fp_wide_function_returns` and `fp_wide_function_parameters` pragmas may affect function definitions or calls but not prototypes.

When the implementation detects an address or `sizeof` operator of a widened parameter or variable it emits a translation-time warning; execution-time behavior is then undefined.

This mechanism facilitates generating efficient code for extended-based or double-based architectures. §3.2.3 and §4.3 provide rationale. The typedefs `float_t` and `double_t` allow finer application than do the pragmas, but require more extensive changes to existing code.

The function writer who decides that narrowing arguments and returns to their semantic type is less desirable than efficiency can write the function under the effect of enabling `fp_wide_function_parameters` and `fp_wide_function_returns` pragmas. Similarly the programmer who uses the function can apply these pragmas to the call site. In either case widening (not narrowing) may or may not occur. These pragmas do not demand widening, nor even recommend it, but merely declare that widening would be acceptable. It is up to the implementation to determine whether widening would be both safe and also more efficient. For example, implementations that pass different type (floating) parameters in different formats can not widen parameters of external functions safely.

Even among implementations which respond to the pragmas, the location in the source code where the pragmas must be placed to be effective may vary. For example, an implementation might perform requisite narrowing of parameters at the call site, in which case the call would have to be under the effect of an enabling `fp_wide_function_parameters` pragma; or, it might narrow within the function, in which case the function definition would have to be under the effect of the pragma.

The pragmas do not affect function prototypes. Doing so might have benefited implementations that pass different floating type arguments and returns in different ways. However, maintaining consistency between the prototype and implementation seemed particularly error prone. A language extension to assure the consistency seemed unjustified. The `float_t` and `double_t` type definitions are available for such prototypes.

Casts are not affected by any of these pragmas, nor by wide expression evaluation, so can be used portably to force narrowing.

Another approach would have been to introduce `register` as a function qualifier that would have allowed widening of both the parameters and the return value. This would have been

inconsistent with the `register` storage class specifier, which is unrelated to widening, and would not have helped with variables.

3.3 Expressions (ISO §6.3; X3.159 §3.3)

3.3.1 The arithmetic operators (ISO §6.3.3.1, 6.3.3.3, 6.3.4-6.3.6, 6.3.16; X3.159 §3.3.3.1, 3.3.3.3, 3.3.4-3.3.6, 3.3.16)

The implementation should follow the guide referenced in Appendix E to document its arithmetic operators.

For IEEE Implementations

These operations fall under the specification of the IEEE floating-point standards:

++	--	+	-	*	/
=	casts	+=	-=	*=	/=

The IEEE floating-point standards require that, given default rounding precision, these operations be rounded to the precision of their evaluation format.

3.3.2 Relational and equality operators (ISO §6.3.8; X3.159 §3.3.8)

For IEEE Implementations

Each of the relational operators (<, <=, >, >=) yields 0 and raises the invalid exception if its operands are unordered, that is, if one or both of the operands is a NaN.

This is as required by the IEEE standards. The IEEE standards identify a need for additional comparison predicates to facilitate writing code that accounts for NaNs. The comparison macros in §4.3 supplement the Standard C equality and relational operators to address this need.

3.4 Constant Expressions (ISO §6.4; X3.159 §3.4)

A non-integral constant expression, other than one in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type, is evaluated (as if) during execution.

This is merely a clarification of Standard C. See §B.4 regarding optimization.

Example

```
#pragma fenv_access on
void f(void) {
   float w[] = { 0.0/0.0 };       /* does not raise an exception */
   static float x = 0.0/0.0;      /* does not raise an exception */
   float y = 0.0/0.0;             /* may raise an exception */
   double z = 0.0/0.0;            /* may raise an exception */
```

For the aggregate and static initializations, the division is done at translation time, raising no (execution-time) exceptions. On the other hand, for the two automatic scalar initializations the invalid division occurs at execution time.

Previous versions of this section allowed translation-time constant arithmetic, but empowered the unary + operator, when applied to an operand, to inhibit translation-time evaluation of constant expressions. The programmer can achieve the efficiency of translation-time evaluation by using static initialization, for example

```
static double one_third = 1.0/3.0;
```

or to decrease the likelihood of storage allocation

```
const static double one_third = 1.0/3.0;
```

For IEEE Implementations

This specification acknowledges the semantics of IEEE arithmetic for the *as if* principle. In particular, execution-time operations must be affected by modes and must raise exceptions as required by the IEEE floating-point standards.

Note that results of inexact expressions like `1.0/3.0` can be affected by rounding modes set at execution time, and expressions such as `0.0/0.0` and `1.0/0.0` can be used reliably to generate execution-time exceptions.

3.5 Initialization (ISO §5.1.2, 6.5.7; X3.159 §2.1.2, 3.5.7)

All computation for automatic scalar initialization is done (as if) at execution time. All computation for initialization of objects that have static storage duration or that have aggregate or union type is done (as if) at translation time.

Standard C does not specify when aggregate and union initialization is done. Otherwise, this is merely a clarification of Standard C. Note that, under the effect of an enabling `fenv_access` pragma, any exception resulting from execution-time initialization must be raised at execution time. See §B.4 regarding optimization.

Example

```
#pragma fenv_access on
void f(void) {
   float u[] = { 1.1e75 };          /* does not raise exceptions */
   static float v = 1.1e75;         /* does not raise exceptions */
   float w = 1.1e75;                /* may raise exceptions */
   double x = 1.1e75;               /* may raise exceptions */
   float y = 1.1e75f;               /* may raise exceptions */
   long double z = 1.1e75;          /* does not raise exceptions */
```

The aggregate and static initializations of `u` and `v` raise no (execution-time) exceptions because their computation is done at translation time. The automatic initialization of `w` requires an execution-time conversion to `float` of the wider value `1.1e75`, which raises exceptions on some implementations, including all IEEE ones. The automatic initializations of `x` and `y` entail execution-time assignment, but, in some expression evaluation methods, not to a narrower format, in which case no exceptions need be raised. The automatic initialization of `z` entails execution-time assignment, but not to a narrower format, so no exception is raised. Note that the conversions of the floating

constants `1.1e75` and `1.1e75f` to their internal representations occur at translation time in all cases.

Use of `float_t` and `double_t` variables increases the likelihood of translation-time computation. For example, the automatic initialization

```
double_t x = 1.1e75;
```

could be done at translation time, regardless of the expression evaluation method.

The specification of §3.4-5 does not suit C++, whose static and aggregate initializers need not be constant. Specifying all floating-point constant arithmetic and initialization to be (as if) at execution time would be consistent with C++ and, given the `fenv_access` mechanism, still would allow the bulk of constant arithmetic to be done, in actuality, at translation time.

For IEEE Implementations

This specification acknowledges the semantics of IEEE arithmetic for the *as if* principle. In particular, execution-time operations must be affected by modes and must raise exceptions as required by the IEEE floating-point standards.

3.6 Predefined Macro Names (ISO §6.8.8; X3.159 §3.8.8)

`__FPCE__`	The decimal constant 1, indicating conformance to this specification.
`__FPCE_IEEE__`	The decimal constant 1, indicating conformance to the parts of this specification for IEEE implementations.

None of these macro names shall be the subject of a `#define` or a `#undef` preprocessing directive.

4. LIBRARY

4.1 Introduction (ISO §7.1.2, 7.1.6; X3.159 §4.1.2, 4.1.6)

Section 4.2 covers numerical extensions to the Standard C libraries. Section 4.3 documents a new header `<fp.h>` which provides facilities for general floating-point programming; `<fp.h>` supersedes `<math.h>`. Section 4.4 documents a new header `<fenv.h>` which provides access to the floating-point environment. The identifiers with external linkage declared in either `<fp.h>` or `<fenv.h>` are reserved for use as identifiers with external linkage only if at least one inclusion of either `<fp.h>` or `<fenv.h>` occurs in one or more of the translation units that constitute the program. In other regards, `<fp.h>` and `<fenv.h>` follow the Standard C specification for standard headers and reserved identifiers (ISO §7.1.2, 7.1.2.1; X3.159 §4.1.2, 4.1.2.1).

This specification for new identifiers with external linkage follows that in [25].

In some cases names are not unique in the first six characters, unlike in Standard C.

4.2 Standard C Library Extensions

4.2.1 General utilities <stdlib.h>

4.2.1.1 The atof function (ISO §7.10.1.1; X3.159 §4.10.1.1)

```
#include <stdlib.h>
double atof(const char *nptr);
```

The `atof` function meets the specifications below for `strtod`.

This document does not require `float` and `long double` versions of `atof`, but instead encourages the use of `strtof` and `strtold` which have a more generally useful interface.

4.2.1.2 The strtof, strtod, and strtold functions (ISO §7.10.1.4; X3.159 §4.10.1.4)

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

The `strtof` and `strtold` functions are `float` and `long double` versions of `strtod`. Their behavior is the same as that of `strtod`. However, if the correct value is outside the range of representable values, `strtof` and `strtold` return `HUGE_VALF` and `HUGE_VALL`, respectively, with the appropriate sign. (`HUGE_VALF` and `HUGE_VALL` are introduced in §4.3.)

Translation-time conversion of valid floating constants exactly matches execution-time conversion provided the default rounding modes are in effect. See §3.1.2.

The expected form of the subject sequence accepted by `strtod` is broadened to include an optional plus or minus sign, then a `0x` or `0X`, and then followed by one of:

1. a nonempty sequence of hexadecimal digits optionally containing a "decimal-point" character, then a binary-exponent part, but no floating suffix;

2. a nonempty sequence of hexadecimal digits containing a "decimal-point" character, but no floating suffix.

For implementations whose `FLT_RADIX` is a power of 2, a hexadecimal floating source value is correctly rounded to the appropriate internal format. For implementations whose `FLT_RADIX` is not a power of 2, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error have the correct sign for the current rounding direction.

Also, the syntax accepted by `strtod` includes:

sign$_{opt}$ **INF**
sign$_{opt}$ **INFINITY**
sign$_{opt}$ **NAN**
sign$_{opt}$ **NAN(***n-char-sequence*$_{opt}$ **)**

where

n-char-sequence:
digit
nondigit
n-char-sequence digit
n-char-sequence nondigit

and any strings equivalent except for case in the `INF`, `INFINITY`, or `NAN` part.

The character sequences `INF` and `INFINITY` produce an infinity, if representable, else are treated as numeric input that overflows.

Character sequences of the form `NAN` or `NAN(`*n-char-sequence*$_{opt}$`)` are converted to quiet NaNs, if supported, else are treated as invalid input. An implementation may use the *n-char-sequence* to determine extra information to be represented in the NaN's significand; which *n-char-sequence*'s are meaningful is implementation-defined.

So much is implementation-defined because so little is portable. Attaching meaning to NaN significands is problematic, even for one implementation, even an IEEE one. For example, the IEEE floating-point standards do not specify the effect of format conversions on NaN significands—conversions, perhaps generated by the compiler, may alter NaN significands in obscure ways.

Requiring a sign for NaN or infinity input was considered as a way of minimizing the chance of mistakenly accepting nonnumeric input. The need for this was deemed insufficient, partly on the basis of prior art.

For simplicity, the infinity and NaN representations are provided through straightforward extensions, rather than through a new locale (ISO §7.4; X3.159 §4.4). Note also that Standard C locale categories do not affect the representations of infinities and NaNs.

The `strtod` function should honor the sign of zero if the arithmetic supports signed zeros.

For IEEE Implementations

Conversion from a numeric decimal string with `DECIMAL_DIG` (defined in §4.3) or fewer significant digits to the widest supported IEEE format is correctly rounded according to the effective rounding direction. For conversion of a numeric decimal string D, with more than `DECIMAL_DIG` digits, consider the two bounding, adjacent decimal strings L and U, both having `DECIMAL_DIG` significant digits, such that the values of L, D, and U satisfy $L \le D \le U$. The result of conversion is one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D has the correct sign for the current rounding direction.

> *Correct rounding* refers to the style of rounding that the IEEE floating-point standards prescribe for their arithmetic operations: the delivered value is the (destination) format's value that is closest, in the sense of the effective rounding direction, to the infinitely precise result.
>
> `DECIMAL_DIG` is sufficiently large that L and U will usually round to the same internal floating value, but if not will round to adjacent values.
>
> The IEEE floating-point standards require perfect rounding for a large though incomplete subset of decimal conversions. This specification goes beyond the IEEE floating-point standards by requiring perfect rounding for all decimal conversions, involving `DECIMAL_DIG` or fewer decimal digits and a supported IEEE format, because practical methods are becoming publicly available (see [12]). Although not requiring correct rounding for arbitrarily wide decimal numbers, this specification is sufficient in the sense that it ensures that every internal numeric value in an IEEE format can be determined as a decimal constant.

The `strtod` function honors the sign of zero. It treats the overflow, underflow, and inexact exceptions in accordance with the IEEE floating-point standards. And, it raises the invalid exception on invalid numeric input.

> This document's previous specification that `strtod` return a NaN for invalid numeric input, as recommended by IEEE standard 854, was withdrawn because of the incompatibility with Standard C, which demands that `strtod` return 0 for invalid numeric input.

4.2.2 Input / output <stdio.h>

4.2.2.1 The fprintf function (ISO §7.9.6.1; X3.159 §4.9.6.1)

The `a` and `A` conversion specifiers provide hexadecimal floating representations. The conversion specifier `a` means that the argument is converted to a hexadecimal floating representation in the style `[-]0xh.hhhhp±d`. The number of hexadecimal digits `h` after the decimal-point character is equal to the precision; if the precision is missing then for implementations whose `FLT_RADIX` is a power of 2 the precision is sufficient for an exact representation of the source value, and for other implementations the precision should be sufficient to distinguish (see rationale below) values of the source type, except that trailing zeros may be omitted. The hexadecimal digit to the left of the decimal-point character is nonzero for normal values and is otherwise chosen by the

implementation; if the precision is zero and the # flag is not specified, no decimal-point character appears. The A conversion specifier will produce a number with X and P instead of x and p. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2.

Binary implementations should choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble boundaries. For example, the next value greater than one in the common IEEE 754 80-bit extended format should be

```
0x8.000000000000001p-3
```

The next value less than one in IEEE 754 double should be

```
0x1.fffffffffffffp-1
```

Note that if the precision is missing, trailing zeros may be omitted. For example, the value positive zero might be

```
0x0.p+0
```

The more suggestive conversion specifiers for hexadecimal formatting, namely x and h, were unavailable. Since h was taken H was ruled out in favor of a lower/upper case option. Possibilities other than a included: b j k m q r t v w y z. The optional h to indicate hexadecimal floating, as in %he, was deemed a less natural fit with the established scheme for specifiers and options.

The *decimal-point character* is defined in Standard C (ISO §7.1.1; X3.159 §4.1.1). *Radix-point character* would have been a better term.

The precision p is sufficient to *distinguish* values of the source type if

$$16^{p-1} > b^{n}$$

where b is FLT_RADIX and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character. (See [6], Ch 7.)

For implementations whose FLT_RADIX is a power of 2, the source value is correctly rounded to a hexadecimal floating number with the given precision. For implementations whose FLT_RADIX is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error have the correct sign for the current rounding direction.

The F conversion specifier has the same effect as f except for infinity and NaN formatting (specified below).

The optional precision, optional L, and 0 and # flag characters all have the same effect for a, A, and F conversions as for e, E, f, g, and G.

With e-style formatting, the exponent field contains at least two digits, and only as many more digits as necessary to represent the exponent.

For all floating conversions, an infinite value is converted in one of the styles [-]inf or [-]infinity—which style is implementation-defined.

For all floating conversions, a NaN value is converted in one of the styles `[-]nan` or `[-]nan(`*n-char-sequence*`)`—which style, and the meaning of any *n-char-sequence*, is implementation-defined.

See §4.2.1.2 Rationale.

When applied to infinite and NaN values, the `-`, `+`, and *space* flag characters have their usual meaning; the `#` and `0` flag characters have no effect .

Use of an upper case format specifier, `A`, `E`, `F`, or `G`, results in `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`.

Use of the `A` and `F` format specifiers constitutes a minor extension to Standard C which does not reserve them.

The results of all floating conversions of a negative zero should include a minus sign.

For IEEE Implementations

If the number of significant decimal digits is at most `DECIMAL_DIG`, then the result from converting a value from an IEEE format is correctly rounded for the current rounding direction. If the number of significant decimal digits is more than `DECIMAL_DIG` but the source value is exactly representable with `DECIMAL_DIG` digits, then the result is an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having `DECIMAL_DIG` significant digits; the value of the result decimal string D satisfies $L \le D \le U$, with the extra stipulation that the error have the correct sign for the current rounding direction.

See §4.2.1.2 Rationale. For binary-to-decimal conversion, the *infinitely precise result* is just the source value, and the destination format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

The results of all floating conversions of a negative zero shall include a minus sign.

4.2.2.2 The fscanf function (ISO §7.9.6.2; X3.159 §4.9.6.2)

For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, all valid syntax—including hexadecimal-floating-constant, infinity, NaN, and signed zero—is treated in the same manner as `strtod`.

4.3 Floating-Point Extensions <fp.h>

The header `<fp.h>` declares several types, macros and functions to support general floating-point programming. It provides essentially all of Standard C `<math.h>`, as well as facilities required or recommended by the IEEE floating-point standards, and a few other facilities that have proved broadly useful. [17] and [1] contain basic descriptions, not repeated in this document, of the Standard C functions. Appendix F contains specification that is particularly suited for IEEE implementations. The implementation should follow the guide referenced in Appendix E to document its `<fp.h>` functions.

The typedefs

```
float_t
double_t
```

are defined to be the implementation's most efficient floating types at least as wide as `float` and `double`, respectively. (Every value of type `float_t` is also a value of type `double_t`.)

Facility to use wider types is needed for writing portable efficient code. Currently Standard C gives no way of asking for the most efficient floating type with at least a given width. Thus different prototypes are more efficient on different floating-point architectures.

architecture (see §3.2.3)	most efficient prototype
extended-based	`long double f(long double)`
double-based	`double f(double)`
single/double	`float f(float)`
single/double/extended	`float f(float)`

Differences may involve whether values can be kept in registers, hence are substantial. Implementations for the various floating-point architectures might use these type definitions:

architecture	`float_t`	`double_t`
extended-based	`long double`	`long double`
double-based	`double`	`double`
single/double	`float`	`double`
single/double/extended	`float`	`double`

An alternate approach of modifying the semantics of the `register` storage-class specifier, when applied to a floating type, to mean that the associated value may be wider than the type, was rejected as inconsistent with existing use of `register` in Standard C.

The macro

```
HUGE_VAL
```

is as defined in ISO C §7.5.

The macros

```
HUGE_VALF
HUGE_VALL
```

are `float` and `long double` analogs of `HUGE_VAL`. They expand to positive `float` and `long double` expressions, respectively. Like `HUGE_VAL`, each can be a positive infinity in an implementation that supports infinities.

The macro

```
INFINITY
```

expands to a floating expression representing an implementation-defined positive or unsigned infinity, if available, else to a positive floating constant that overflows at translation time.

The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs. It expands to a floating-point expression representing an implementation-defined quiet NaN.

Ideally the `INFINITY` and `NAN` macros would be suitable for static and aggregate initialization, though currently such is not required by this specification.

No type is specified for the `INFINITY` and `NAN` macros because conversions between floating-point types convert infinities to infinities and NaNs to NaNs, without raising exceptions.

The macros

```
FP_NAN
FP_INFINITE
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

are for number classification. They represent the mutually exclusive kinds of floating-point values. They expand to `int` constant expressions with distinct values and suitable for use in `#if` preprocessing directives.

Some prior art uses a finer classification: `FP_POS_INFINITE`, `FP_NEG_INFINITE`, etc. The current consensus is that those specified, in conjunction with the `signbit` macro, are generally preferable.

Each macro function specified below expands to code that evaluates its arguments exactly once, fully protected by parentheses where necessary. Its result is undefined if its arguments are not of floating type. The notation *`floating-expr`* in the macro function forms indicates an expression of floating type.

Requiring the arguments to be of floating type allows efficient implementation.

The macro

```
fpclassify(floating-expr)
```

evaluates to the value of the number classification macro appropriate to the value of its argument. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

It is important to know the type that classification is based on. For example, a value might be zero when converted to `float`, subnormal when converted to `double`, and normal when converted to `long double`.

`fpclassify` might be implemented as

```
#define fpclassify(x) ((sizeof(x) == sizeof(float)) ? __fpclassifyf(x) \
    : (sizeof(x) == sizeof(double)) ? __fpclassifyd(x) \
    : __fpclassifyl(x))
```

The macro

`signbit(`*floating-expr*`)`

evaluates to an `int` expression that is nonzero if and only if the sign of the argument (infinities, zeros, and NaNs included) is negative.

The IEEE floating-point standards do not specify the effect of arithmetic operations on a NaN's sign bit. However, `signbit` of a NaN is valid in the sense that, even if `x` is a NaN, `signbit(x)` is invariant until `x` changes.

The macro

`isfinite(`*floating-expr*`)`

evaluates to an `int` expression that is nonzero if and only if its argument has a finite value (zero, subnormal, or normal). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

The macro

`isnormal(`*floating-expr*`)`

evaluates to an `int` expression that is nonzero if and only if its argument value is *normal*: neither zero, subnormal, infinite, nor NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

The macro

`isnan(`*floating-expr*`)`

evaluates to an `int` expression that is nonzero if and only if its argument value is a NaN.

The Standard C relational and equality operators support the usual mathematical relationships between numeric values. Comparison macros supplement these operators to support relationships involving nonnumeric values, NaNs. For any ordered pair of numeric values exactly one of the relationships—*less*, *greater*, and *equal*—is true. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.

The IEEE standards require that the relational operators `<`, `<=`, `>`, and `>=` raise the invalid exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs. The relational macros specified below, which

are just *quiet* versions of the relational operators, facilitate writing efficient code that accounts for NaNs without suffering the invalid exception.

The macro

```
isgreater(floating-expr, floating-expr)
```

evaluates to an `int` expression that is nonzero if and only if its first argument is greater than its second argument. The value of `isgreater(x,y)` is always equal to `(x) > (y)`; however, unlike `(x) > (y)`, `isgreater(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

The macro

```
isgreaterequal(floating-expr, floating-expr)
```

evaluates to an `int` expression that is nonzero if and only if its first argument is greater than or equal to its second argument. The value of `isgreaterequal(x,y)` is always equal to `(x) >= (y)`; however, unlike `(x) >= (y)`, `isgreaterequal(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

The macro

```
isless(floating-expr, floating-expr)
```

evaluates to an `int` expression that is nonzero if and only if its first argument is less than its second argument. The value of `isless(x,y)` is always equal to `(x) < (y)`; however, unlike `(x) < (y)`, `isless(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

The macro

```
islessequal(floating-expr, floating-expr)
```

evaluates to an `int` expression that is nonzero if and only if its first argument is less than or equal to its second argument. The value of `islessequal(x,y)` is always equal to `(x) <= (y)`; however, unlike `(x) <= (y)`, `islessequal(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

The macro

```
islessgreater(floating-expr, floating-expr)
```

evaluates to an `int` expression that is nonzero if and only if its first argument is less than or greater than its second argument. The macro `islessgreater(x,y)` is similar in spirit to `(x) < (y) || (x) > (y)`; however, `islessgreater(x,y)` does not raise the invalid exception when `x` and `y` are unordered (nor does it evaluate `x` and `y` twice).

The macro

```
isunordered(floating-expr, floating-expr)
```

evaluates to an `int` expression that is nonzero if and only if its arguments are unordered.

The comparison macros (`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, `isunordered`) should be implemented to be as efficient as if they were built-in operators.

For implementations with NaNs, the translator should recognize the macros in order to provide efficient implementation. Typical hardware offers efficient quiet comparisons. The semantically correct implementation

```
int isless(x, y) { return ! (isnan(x) || isnan(y) || x >= y); }
```

is unsuitable for efficiency reasons.

Programs written for (or ported to) systems with NaNs will be expected to handle invalid and NaN input in reasonable ways. The comparison macros support such programs. The arithmetic operators (+, *, ...) propagate NaNs quietly; the comparison macros facilitate directing NaNs through branches quietly. Thus NaNs can flow through many computations without the need for inefficient or unduly obfuscating code, and without raising inappropriate exceptions.

For implementations without NaNs, the macros can be defined trivially:

```
#define isgreater(x,y) ((x)>(y))
...
#define isunordered(x,y) 0
```

Several previous versions of this document proposed extending the relational operators:

Symbol	Relation
<	less
>	greater
<=	less or equal
>=	greater or equal
==	equal
!=	unordered, less, or greater
!<>=	unordered
<>	less or greater
<>=	less, equal, or greater
!<=	unordered or greater
!<	unordered, greater, or equal
!>=	unordered or less
!>	unordered, less, or equal
!<>	unordered or equal

The additional operators were to be analogous to, and have the same precedence as, the Standard C relational operators. The ! symbol was to indicate awareness of NaNs, so operators including the ! symbol would not raise the invalid exception for unordered operands. Where the operands have types and values suitable for relational operators, the semantics detailed in ISO §6.3.8 (X3.159 §3.3.8) were to apply. The Standard C *operator* syntax (ISO §6.1.5; X3.159 §3.1.5) was to be augmented to include the additional operators. This approach would have had the advantages of brevity and clearer promise of efficiency, at no greater implementation cost for systems that support NaNs (the large

majority of systems do support NaNs). However, it was rejected because of reluctance to extend the language definition for functionality which could be provided with a library interface, and because continuing contentiousness might discourage implementation. Also, in some cases, the macros provide a more straightforward articulation, e.g. `isless(x,y)` instead of `! (x !< y)`.

The IEEE floating-point standards enumerate 26 functionally distinct comparison predicates, from combinations of the four comparison results and whether invalid is raised. The following table shows how the previous and current proposals cover all important cases:

greater	less	equal	unordered	raises exception	previous proposal	current proposal
		√			`x == y`	`x == y`
√	√		√		`x != y`	`x != y`
√				√	`x > y`	`x > y`
√		√		√	`x >= y`	`x >= y`
	√			√	`x < y`	`x < y`
	√	√		√	`x <= y`	`x <= y`
			√		`x !<>= y`	`isunordered(x,y)`
√	√			√	`x <> y`	N/A
√	√	√		√	`x <>= y`	N/A
√			√		`x !<= y`	`! islessequal(x,y)`
√		√	√		`x !< y`	`! isless(x,y)`
	√		√		`x !>= y`	`! isgreaterequal(x,y)`
	√	√	√		`x !> y`	`! isgreater(x,y)`
		√	√		`x !<> y`	`! islessgreater(x,y)`
	√	√	√	√	`! (x > y)`	`! (x > y)`
	√		√	√	`! (x >= y)`	`! (x >= y)`
√		√	√	√	`! (x < y)`	`! (x < y)`
√			√	√	`! (x <= y)`	`! (x <= y)`
√	√	√			`! (x !<>= y)`	`! isunordered(x,y)`
		√	√	√	`! (x <> y)`	N/A
			√	√	`! (x <>= y)`	N/A
	√	√			`! (x !<= y)`	`islessequal(x,y)`
	√				`! (x !< y)`	`isless(x,y)`
√		√			`! (x !>= y)`	`isgreaterequal(x,y)`
√					`! (x !> y)`	`isgreater(x,y)`
√	√				`! (x !<> y)`	`islessgreater(x,y)`

The previous proposal would have naturally covered the four N/A cases not covered by the current proposal. (The current proposal covers them, except for the invalid exception.) However, covering these cases per se is unimportant, because the facility would provide no additional capability except more ways to write NaN-unaware code.

In the interest of efficiency, note that each quiet combination of less, greater, equal, and unordered can be tested with a single comparison macro or Standard C equality or relational operator.

The proposal for `!` operators supplanted an earlier proposal that would have augmented the set of relations by using the `?` symbol to denote unordered, for example `a ?>= b` instead of `a !< b`. Use of the `?` relationals would have had the advantage that the unordered case would have been dealt with explicitly. However, the `!` relationals seemed a more natural language extension, particularly from the point of view of programmers for (non-IEEE) implementations not detecting unordered. Also, using `??` as proposed for the unordered operator would have conflicted with Standard C trigraphs.

Other macro approaches, such as

```
isrelation(x, FP_UNORDERED | FP_LESS | FP_EQUAL, y)
```

seemed more cumbersome than the current proposal.

Without any language or library support `isgreater(a,b)` might be implemented by the programmer as

```
! (a != a || b != b || a <= b)
```

However, even more awkward code would be required if `a` or `b` had side effects. The programmer would have to remember to put the NaN tests first, and trust the compiler not to replace `a != a || b != b` by *false*. Also, special optimization would be necessary to generate efficient code. Use of `isnan` helps only a little.

The macro

```
DECIMAL_DIG
```

expands to an `int` constant expression suitable for use in `#if` preprocessing directives. Its value is an implementation-defined number of decimal digits which is supported by conversion between decimal and all internal floating-point formats. Conversion from (at least) `double` to decimal with `DECIMAL_DIG` digits and back should be the identity function. And, `DECIMAL_DIG` should give an appropriate number of digits to carry in canonical decimal representations.

In order that correctly rounded conversion from an internal floating-point format with precision m to decimal with `DECIMAL_DIG` digits and back be the identity function, `DECIMAL_DIG` should be a positive integer n satisfying

$$\begin{array}{ll} n \geq m, & \text{if FLT\_RADIX is 10} \\ 10^{n-1} > \texttt{FLT\_RADIX}^{m}, & \text{otherwise} \end{array}$$

Refer to [6], Chapter 7, for details.

`DECIMAL_DIG` is distinct from Standard C's `DBL_DIG`, which is defined in terms of conversion from decimal to `double` and back.

`DECIMAL_DIG` was deemed more useful than `FP_CONV_DIG`, which previous versions of this document defined as the number of decimal digits for which the implementation guaranteed correctly rounded conversion.

For IEEE Implementations

Conversion from the widest supported IEEE standard format to decimal with `DECIMAL_DIG` digits and back is the identity function. See §4.2.1.2 and 4.2.2.1 regarding conversion requirements.

Examples

If the minimum-width IEEE 754 extended format (64 bits of precision) is supported, `DECIMAL_DIG` must be at least 21. If IEEE 754 double (53 bits of precision) is the widest IEEE format supported, then `DECIMAL_DIG` must be at least 17. By contrast, `LDBL_DIG` and `DBL_DIG` are 19 and 15, respectively, for these formats.

4.3.1 Overloaded functions

The specification of the header `<fp.h>` uses the notion of *overloaded* functions. The function designators are *overloaded* with multiple prototypes which become declared

when `<fp.h>` is included. They are used much like ordinary functions, but they behave more like typical built-in arithmetic operators: their semantic type is affected by their arguments; their result is at least as wide as the minimum evaluation format: and, with widest-need expression evaluation, the evaluation format propagates from the call to its arguments.

For each overloaded function defined in this section, the implementation provides a set of functions—one for the minimum evaluation format and one for each wider floating format. Their prototypes are identical except for their floating-point return type and one or more overloading parameters whose type is identical to the return type. Prototypes have the form

```
floating-type function-designator ( argument-list ) ;
```

Subsequent sections give a form in this style for each overloaded function. The form determines the required prototypes.

Example

The square root function has the prototype form

```
floating-type sqrt(floating-type x);
```

So, if the minimum evaluation format were `float` then the implementation would have

```
float sqrt(float);
double sqrt(double);
long double sqrt(long double);
```

Example

The `remquo` function has the form

```
floating-type remquo(floating-type x, floating-type y, int *quo);
```

So, if the minimum evaluation format were `double` then the implementation would have

```
double remquo(double, double, int *);
long double remquo(long double, long double, int *);
```

A function call that is compatible with one of the prototypes for an overloaded function is compatible with all of them. For such a function call, occurring after inclusion of `<fp.h>` in the translation unit, the implementation determines the evaluation format for the call and chooses the matching prototype. The evaluation format for the call is determined to be the widest of

1. the floating arguments for the overloading parameters,
2. the minimum evaluation format,

and in the case of widest-need (§3.2.3.1)

3. any evaluation format propagated down to the call.

With widest-need expression evaluation, the evaluation format for the call, which matches the type of the invoked function, is propagated through the call to the arguments corresponding to the overloading parameters.

Taking the address of an overloaded function or passing it as an argument to a function results in undefined behavior.

Overloaded functions cooperate with expression evaluation methods much as arithmetic operators do. For example, with

```
double d; float f;
d = sqrt(f + f);
```

the type of `sqrt` and evaluation format of `+` actually used are:

min-eval-format	widest-need	sqrt()	f + f
float	no	float	float
double	no	double	double
long double	either	long double	long double
float	yes	double	double
double	yes	double	double

Modeled after C++, the rudimentary overloading required by this specification can reasonably be expected to be compatible with future overloading extensions to C. General purpose overloading for C is beyond the scope of this document.

This specification does not prescribe any particular implementation mechanism. It could be implemented simply with built-in macros. A function such as

```
floating-type sqrt(floating-type);
```

could be implemented with

```
#undef sqrt
#define sqrt(x) __BUILTIN_OVERLOAD_sqrt(x)
```

Note that implementations whose minimum evaluation format is `long double` require no overloading mechanism since only one prototype, the `long double` one, is needed for each function.

This specification borrows heavily from [14] and [15]. [15] is functionally similar, but is presented in the style of Fortran generic intrinsic functions. Like this specification, [14] is consistent with C++ overloading. Adding a suitable subset of C++ overloading to C, as prescribed by [14], would support the overloading required here, and as well provide a useful user-extensible overloading mechanism. (With the current C++ overloading rules, additional prototypes would be required to handle integer arguments. [14] suggests how C++ overloading might be extended to accommodate widest-need.)

The C* language, an extended C, includes a function overloading scheme that supports its parallel types. The C* definition in [21] differs from this specification and current C++ in that its overloading resolution depends on the order of parameters.

The great majority of existing C programs are expected to run correctly straightaway when `<fp.h>` is included instead of `<math.h>`. Overloaded functions differ from corresponding Standard C library functions essentially only in the semantic types of return values.

Use of Standard C's reserved `f` and `l` suffixed identifiers could be supported with macros such as

```
#define sinf(x) sin((float)(x))
```

(Taking the address of such a function or passing it as an argument would be disallowed.)

The ability to overload on integer as well as floating types would have been useful for some functions, for example `copysign`. Overloading with different numbers of arguments would have allowed reusing names, for example `remainder` for `remquo`. However, these facilities would have complicated the specification somewhat and their natural consistent use, such as for a floating `abs` or a two argument `atan`, would have introduced further inconsistencies with Standard C for insufficient benefit.

This specification in no way limits the implementation's options for efficiency, including inlining library functions (ISO §7.1.6; X3.159 §4.1.6).

4.3.2 Trigonometric functions (ISO §7.5.2; X3.159 §4.5.2)

The header `<fp.h>` declares overloaded versions of all the Standard C trigonometric functions.

4.3.2.1 The acos function (ISO §7.5.2.1; X3.159 §4.5.2.1)

Synopsis

```
#include <fp.h>
floating-type acos(floating-type x);
```

4.3.2.2 The asin function (ISO §7.5.2.2; X3.159 §4.5.2.2)

Synopsis

```
#include <fp.h>
floating-type asin(floating-type x);
```

4.3.2.3 The atan function (ISO §7.5.2.3; X3.159 §4.5.2.3)

Synopsis

```
#include <fp.h>
floating-type atan(floating-type x);
```

4.3.2.4 The atan2 function (ISO §7.5.2.4; X3.159 §4.5.2.4)

Synopsis

```
#include <fp.h>
floating-type atan2(floating-type y, floating-type x);
```

4.3.2.5 The cos function (ISO §7.5.2.5; X3.159 §4.5.2.5)

Synopsis

```
#include <fp.h>
floating-type cos(floating-type x);
```

4.3.2.6 The sin function (ISO §7.5.2.6; X3.159 §4.5.2.6)

Synopsis

```
#include <fp.h>
floating-type sin(floating-type x);
```

4.3.2.7 The tan function (ISO §7.5.2.7; X3.159 §4.5.2.7)

Synopsis

```
#include <fp.h>
floating-type tan(floating-type x);
```

4.3.3 Hyperbolic functions (ISO §7.5.3; X3.159 §4.5.3)

The header `<fp.h>` declares overloaded versions of the Standard C hyperbolic functions and their *arc* counterparts.

4.3.3.1 The acosh function

Synopsis

```
#include <fp.h>
floating-type acosh(floating-type x);
```

Description

The `acosh` function computes the (nonnegative) arc hyperbolic cosine of `x`.

Returns

The `acosh` function returns the arc hyperbolic cosine.

4.3.3.2 The asinh function

Synopsis

```
#include <fp.h>
floating-type asinh(floating-type x);
```

Description

The `asinh` function computes the arc hyperbolic sine of `x`.

Returns

The `asinh` function returns the arc hyperbolic sine.

4.3.3.3 The atanh function

Synopsis

```
#include <fp.h>
floating-type atanh(floating-type x);
```

Description

The `atanh` function computes the arc hyperbolic tangent of `x`.

Returns

The `atanh` function returns the arc hyperbolic tangent .

4.3.3.4 The cosh function (ISO §7.5.3.1; X3.159 §4.5.3.1)

Synopsis

```
#include <fp.h>
floating-type cosh(floating-type x);
```

4.3.3.5 The sinh function (ISO §7.5.3.2; X3.159 §4.5.3.2)

Synopsis

```
#include <fp.h>
floating-type sinh(floating-type x);
```

4.3.3.6 The tanh function (ISO §7.5.3.3; X3.159 §4.5.3.3)

Synopsis

```
#include <fp.h>
floating-type tanh(floating-type x);
```

4.3.4 Exponential and logarithmic functions (ISO §7.5.4; X3.159 §4.5.4)

The header `<fp.h>` declares overloaded versions of the Standard C exponential and logarithmic functions—except for `modf` which is declared but not overloaded—and several related functions.

4.3.4.1 The exp function (ISO §7.5.4.1; X3.159 §4.5.4.1)

Synopsis

```
#include <fp.h>
floating-type exp(floating-type x);
```

4.3.4.2 The exp2 function

Synopsis

```
#include <fp.h>
floating-type exp2(floating-type x);
```

Description

The `exp2` function computes the base-2 exponential of `x`: 2^x.

Returns

The `exp2` function returns the base-2 exponential.

4.3.4.3 The expm1 function

Synopsis

```
#include <fp.h>
floating-type expm1(floating-type x);
```

Description

The `expm1` function computes the base-e exponential of the argument, minus `1`: $e^x - 1$. For small magnitude `x`, `expm1(x)` is expected to be more accurate than `exp(x) - 1`.

Returns

The `expm1` function returns $e^x - 1$.

4.3.4.4 The frexp function (ISO §7.5.4.2; X3.159 §4.5.4.2)

Synopsis

```
#include <fp.h>
floating-type frexp(floating-type value, int *exp);
```

4.3.4.5 The ldexp function (ISO §7.5.4.3; X3.159 §4.5.4.3)

Synopsis

```
#include <fp.h>
floating-type ldexp(floating-type x, int exp);
```

4.3.4.6 The log function (ISO §7.5.4.4; X3.159 §4.5.4.4)

Synopsis

```
#include <fp.h>
floating-type log(floating-type x);
```

4.3.4.7 The log10 function (ISO §7.5.4.5; X3.159 §4.5.4.5)

Synopsis

```
#include <fp.h>
floating-type log10(floating-type x);
```

4.3.4.8 The log1p function

Synopsis

```
#include <fp.h>
floating-type log1p(floating-type x);
```

Description

The `log1p` function computes the base-e logarithm of 1 plus the argument. For small magnitude `x`, `log1p(x)` is expected to be more accurate than `log(1 + x)`.

Returns

The `log1p` function returns the base-e logarithm of 1 plus the argument.

4.3.4.9 The log2 function

Synopsis

```
#include <fp.h>
floating-type log2(floating-type x);
```

Description

The `log2` function computes the base-2 logarithm of `x`.

Returns

The `log2` function returns the base-2 logarithm.

4.3.4.10 The logb function

Synopsis

```
#include <fp.h>
floating-type logb(floating-type x);
```

Description

The `logb` function extracts the exponent of `x`, as a signed integral value in the format of `x`. If `x` is subnormal it is treated as though it were normalized; thus for positive finite `x`,

$$1 \leq x * \texttt{FLT\_RADIX}^{-\texttt{logb}(x)} < \texttt{FLT\_RADIX}$$

The treatment of subnormal `x` follows the recommendation in IEEE standard 854, which differs from IEEE standard 754 on this point. Even 754 implementations should follow this definition rather than the one recommended (not required) by 754.

Particularly on machines whose radix is not 2, `logb` can be expected to obtain the exponent more accurately and quickly than `frexp`.

Returns

The `logb` function returns the signed exponent of its argument.

4.3.4.11 The modf functions (ISO §7.5.4.7; X3.159 §4.5.4.6)

Synopsis

```
#include <fp.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

4.3.4.12 The scalb function

Synopsis

```
#include <fp.h>
floating-type scalb(floating-type x, long int n);
```

Description

The `scalb` function computes `x * FLT_RADIX`^{n} efficiently, not normally by computing `FLT_RADIX`^{n} explicitly.

Returns

The `scalb` function returns `x * FLT_RADIX`^{n} .

On machines whose radix is not 2, `scalb`, compared with `ldexp`, can be expected to have better accuracy, speed, and overflow and underflow behavior.

The second parameter has type `long int`, unlike the corresponding `int` parameter for `ldexp`, because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width `int` format allowed by Standard C.

4.3.5 Power and absolute value functions (ISO §7.5.5, 7.5.6; X3.159 §4.5.5, 4.5.6)

The header <fp.h> declares overloaded versions of the Standard C power and absolute value functions and a hypotenuse function.

4.3.5.1 The fabs function (ISO §7.5.6.2; X3.159 §4.5.6.2)

Synopsis

```
#include <fp.h>
floating-type fabs(floating-type x);
```

4.3.5.2 The hypot function

Synopsis

```
#include <fp.h>
floating-type hypot(floating-type x, floating-type y);
```

Description

The `hypot` function computes the square root of the sum of the squares of `x` and `y`, without undue overflow or underflow.

Returns

The `hypot` function returns the square root of the sum of the squares of `x` and `y`.

4.3.5.3 The pow function (ISO §7.5.5.1; X3.159 §4.5.5.1)

Synopsis

```
#include <fp.h>
floating-type pow(floating-type x, floating-type y);
```

4.3.5.4 The sqrt function (ISO §7.5.5.2; X3.159 §4.5.5.2)

Synopsis

```
#include <fp.h>
floating-type sqrt(floating-type x);
```

4.3.6 Error and gamma functions

See [23] regarding implementation.

4.3.6.1 The erf function

Synopsis

```
#include <fp.h>
floating-type erf(floating-type x);
```

Description

The `erf` function computes the error function of `x`.

Returns

The `erf` function returns the error function of `x`.

4.3.6.2 The erfc function

Synopsis

```
#include <fp.h>
floating-type erfc(floating-type x);
```

Description

The `erfc` function computes the complementary error function of `x`.

Returns

The `erfc` function returns the complementary error function of `x`.

4.3.6.3 The gamma function

Synopsis

```
#include <fp.h>
floating-type gamma(floating-type x);
```

Description

The `gamma` function computes the gamma function of `x`: $\Gamma(x)$.

Returns

The `gamma` function returns $\Gamma(x)$.

In UNIX System V [10], both the `gamma` and `lgamma` functions compute $\log_e(|\Gamma(x)|)$.

4.3.6.4 The lgamma function

Synopsis

```
#include <fp.h>
floating-type lgamma(floating-type x);
```

Description

The `lgamma` function computes the logarithm of the absolute value of gamma of `x`: $\log_e(|\Gamma(x)|)$.

In UNIX System V [10], a call to `lgamma` sets an external variable `signgam` to the sign of `gamma(x)`, which is -1 if

```
x < 0 && remainder(floor(x), 2) != 0
```

Note that this specification does not remove the external identifier `signgam` from the user's name space. An implementation that supports, as an extension, `lgamma`'s setting of `signgam` must still protect the external identifier `signgam` if defined by the user.

Returns

The `lgamma` function returns $\log_e(|\Gamma(x)|)$.

4.3.7 Nearest integer functions (ISO §7.5.6; X3.159 §4.5.6)

The header `<fp.h>` declares overloaded versions of the Standard C nearest integer functions, nearest integer functions specified by the IEEE standards, and functions similar to common Fortran nearest integer functions.

4.3.7.1 The ceil function (ISO §7.5.6.1; X3.159 §4.5.6.1)

Synopsis

```
#include <fp.h>
floating-type ceil(floating-type x);
```

4.3.7.2 The floor function (ISO §7.5.5.1; X3.159 §4.5.5.1)

Synopsis

```
#include <fp.h>
floating-type floor(floating-type x);
```

4.3.7.3 The nearbyint function

Synopsis

```
#include <fp.h>
floating-type nearbyint(floating-type x);
```

Description

The `nearbyint` function differs from the `rint` function (§4.3.7.4) only in that the `nearbyint` function does not raise the inexact exception. (See §F.6.3-4.)

For implementations that do not support the inexact exception, `nearbyint` and `rint` are equivalent.

Returns

The `nearbyint` function returns the rounded integral value.

4.3.7.4 The rint function

Synopsis

```
#include <fp.h>
floating-type rint(floating-type x);
```

Description

The `rint` function rounds its argument to an integral value in floating-point format, using the current rounding direction.

Returns

The `rint` function returns the rounded integral value.

4.3.7.5 The rinttol function

Synopsis

```
#include <fp.h>
long int rinttol(long double x);
```

Description

The `rinttol` function rounds its argument to the nearest `long int`, rounding according to the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified.

Returns

The `rinttol` function returns the rounded `long int` value, using the current rounding direction.

4.3.7.6 The round function

Synopsis

```
#include <fp.h>
floating-type round(floating-type x);
```

Description

The `round` function rounds its argument to the nearest integral value in floating-point format, using *add half to the magnitude and chop* rounding a la the Fortran `anint` function, regardless of the current rounding direction.

Returns

The `round` function returns the rounded integral value.

4.3.7.7 The roundtol function

Synopsis

```
#include <fp.h>
long int roundtol(long double x);
```

Description

The `roundtol` function returns the rounded `long int` value, using *add half to the magnitude and chop* rounding a la the Fortran `nint` function and the Pascal `round` function, regardless of the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified.

Returns

The `roundtol` function returns the rounded `long int` value.

4.3.7.8 The trunc function

Synopsis

```
#include <fp.h>
floating-type trunc(floating-type x);
```

Description

The `trunc` function rounds its argument to the integral value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

The `trunc` function returns the truncated integral value.

4.3.8 Remainder functions (ISO §7.5.6; X3.159 §4.5.6)

The header `<fp.h>` declares an overloaded Standard C `fmod` function and two versions of the IEEE floating-point standards' remainder function.

4.3.8.1 The fmod function (ISO §7.5.6.4; X3.159 §4.5.6.4)

Synopsis

```
#include <fp.h>
floating-type fmod(floating-type x, floating-type y);
```

4.3.8.2 The remainder function

Synopsis

```
#include <fp.h>
floating-type remainder(floating-type x,
                        floating-type y);
```

Description

The `remainder` function computes the remainder `x` REM `y` required by the IEEE floating-point standards.

> "When $y \neq 0$, the remainder $r = x$ *REM* y is defined regardless of the rounding mode by the mathematical relation $r = x - y * n$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x." [3] §5.1.

(This definition can be implemented for non-IEEE as well as IEEE machines.)

Returns

The `remainder` function returns `x` REM `y`.

4.3.8.3 The remquo function

Synopsis

```
#include <fp.h>
floating-type remquo(floating-type x, floating-type y,
                     int *quo);
```

Description

The `remquo` function computes the same remainder as the `remainder` function. In the object pointed to by `quo` it stores a value whose sign is the sign of `x/y` and whose magnitude is congruent mod 2^n to the magnitude of the integral quotient of `x/y`, where `n` is an implementation-defined integer at least 3.

Returns

The `remquo` function returns `x REM y`.

> The `remquo` function is intended for implementing argument reductions, which can exploit a few low-order bits of the quotient. Note that `x` may be so large in magnitude relative to `y` that an exact representation of the quotient is not practical.

4.3.9 Manipulation functions

These extension functions manipulate representations of floating-point numbers.

4.3.9.1 The copysign function

Synopsis

```
#include <fp.h>
floating-type copysign(floating-type x, floating-type y);
```

Description

The `copysign` function produces a value with the magnitude of `x` and the sign of `y`. It produces a NaN (with the sign of `y`) if `x` is a NaN. On implementations that represent a *signed* zero but do not treat negative zero consistently in arithmetic operations, the `copysign` function regards the sign of zero as positive.

The requirement that `copysign` regard a negative sign of zero as positive if the arithmetic treats negative zero like positive zero is justified in order to preserve more identities. For example, to preserve the identity, *the square root of the product is the product of the square roots*, the algorithm in [22] for the complex square root depends on consistency of `copysign` with the rest of the arithmetic: if -0 behaves like +0 then the square root of the product would yield

$$\sqrt{3 * (-1 - 0i)} = \sqrt{-3 + 0i} \rightarrow 0 + \sqrt{3}i$$

but if `copysign` were to treat the sign of -0 as negative then the product of the square roots would yield

$$\sqrt{3} * \sqrt{-1 - 0i} \rightarrow \sqrt{3} * (0 - i) = 0 - \sqrt{3}i$$

Returns

The `copysign` function returns a value with the magnitude of `x` and the sign of `y`.

4.3.9.2 The nan functions

Synopsis

```
#include <fp.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Description

If the implementation supports quiet NaNs in the type of the function, then the call `nan("`*n-char-sequence*`")` is equivalent to `strtod("NAN(`*n-char-sequence*`)", (char**) NULL)`; the call `nan("")` is equivalent to `strtod("NAN()", (char**) NULL)`. Similarly `nanf` and `nanl` are defined in terms of `strtof` and `strtold`. If `tagp` does not point to an *n-char-sequence* string then the result NaN's content is unspecified. A call to a `nan` function of a type for which the implementation does not support quiet NaNs is unspecified.

Returns

The `nan` functions return a quiet NaN, if available, with content indicated through `tagp`.

4.3.9.3 The nextafter functions

Synopsis

```
#include <fp.h>
floating-type nextafter(floating-type x, long double y);
float nextafterf(float x, float y);
double nextafterd(double x, double y);
long double nextafterl(long double x, long double y);
```

Description

The `nextafter` functions determine the next representable value, in the type of the function, after `x` in the direction of `y`. The `nextafter` functions return `y` if `x == y`.

Returns

The `nextafter` functions return the next representable value after `x` in the direction of `y`.

It's sometimes desirable to find the next representation after a value in the direction of a previously computed value—maybe smaller, maybe larger. The `nextafter` functions have a second floating argument so that the program will not have to include floating-point tests for determining the direction in such situations. And, on some machines these tests may fail due to overflow, underflow, or roundoff.

The overloaded `nextafter` function depends substantially on the expression evaluation method—which is appropriate for certain uses but not for others. The explicitly typed functions can be employed to obtain next values in a particular format. For example,

```
nextafterf(x, y)
```

will return the next `float` value after `(float) x` in the direction of `(float) y` regardless of the evaluation method.

The second parameter of the overloaded `nextafter` function has type `long double` primarily to keep the overloading scheme simple. Promotion of the second argument to `long double` is harmless but unnecessary.

For the case `x == y`, the IEEE floating-point standards recommend that `x` be returned. This specification differs in order that `nextafter(-0.0, +0.0)` return `+0.0` and `nextafter(+0.0, -0.0)` return `-0.0`.

4.3.10 Maximum, minimum, and positive difference functions

These extension functions correspond to standard Fortran functions, `dim`, `max`, and `min` [16].

Their names have `f` prefixes to allow for integer versions—following the example of `fabs` and `abs`.

4.3.10.1 The fdim function

Synopsis

```
#include <fp.h>
floating-type fdim(floating-type x, floating-type y);
```

Description

The `fdim` function determines the *positive difference* between its arguments:

```
x - y ,  if x > y
 +0   ,  if x ≤ y
```

Returns

The `fdim` function returns the positive difference between `x` and `y`.

4.3.10.2 The fmax function

Synopsis

```
#include <fp.h>
floating-type fmax(floating-type x, floating-type y);
```

Description

The `fmax` function determines the maximum numeric value of its arguments.

NaN arguments should be treated as missing data. If one argument is a NaN and the other numeric, then `fmax` should chose the numeric value. (See §F.9.2.)

Returns

The `fmax` function returns the maximum numeric value of its arguments.

4.3.10.3 The fmin function

Synopsis

```
#include <fp.h>
floating-type fmin(floating-type x, floating-type y);
```

Description

The `fmin` function determines the minimum numeric value of its arguments.

NaN arguments should be treated as missing data. If one argument is a NaN and the other numeric, then `fmin` should chose the numeric value. (See §F.9.3.)

Returns

The `fmin` function returns the minimum numeric value of its arguments.

4.4 Floating-Point Environment <fenv.h>

The header `<fenv.h>` declares two types and several macros and functions to provide access to the floating-point environment.

Names of macros and functions in this section consistently include an `FE_` or `fe` prefix and employ certain abbreviations. The prefix calls attention to environmental access functions, which require an enabling `fenv_access` pragma (see §2). The abbreviations `env` and `except` are used in Standard C and UNIX System V, respectively.

The interface described here is not intended to support trap handlers, which are outside the scope of this document.

The typedef

```
fenv_t
```

is a type for representing the entire floating-point environment.

The typedef

```
fexcept_t
```

is a type for representing the floating-point exception flags collectively, including any status the implementation associates with the flags.

See rationale in §4.4.1.

Each macro

```
FE_INEXACT
FE_DIVBYZERO
FE_UNDERFLOW
FE_OVERFLOW
FE_INVALID
```

is defined if and only if the implementation supports the exception by means of the functions in §4.4.1. The defined macros expand to `int` constant expressions whose values are distinct powers of 2, all suitable for use in `#if` preprocessing directives.

The macro

```
FE_ALL_EXCEPT
```

is simply the bitwise OR of all exception macros defined by the implementation.

Each macro

```
FE_TONEAREST
FE_UPWARD
FE_DOWNWARD
FE_TOWARDZERO
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the `fegetround` and `fesetround` functions. The defined macros expand to `int` constant expressions whose values are distinct nonnegative values, all suitable for use in `#if` preprocessing directives.

The rounding direction macros might expand to constants corresponding to the values of `FLT_ROUNDS`, the Standard C inquiry for the rounding direction of addition, but need not.

The macro

```
FE_DFL_ENV
```

represents the default floating-point environment—the one installed at program startup—and has type *pointer to* `fenv_t`. It can be used as an argument to `<fenv.h>` functions that manage the floating-point environment.

Unsupported macros are not defined in order to assure that their use results in a translation error. A program might explicitly define such macros, to allow translation of code (perhaps never executed) containing the macros. An unsupported exception macro should be defined to be 0, for example

```
#ifndef FE_INEXACT
#define FE_INEXACT 0
#endif
```

so that a bitwise OR of macros has a reasonable effect.

4.4.1 Exceptions

The following functions provide access to the exception flags. The `int` input argument for the functions represents a subset of floating-point exceptions, and can be constructed by bitwise ORs of the exception macros, for example `FE_OVERFLOW | FE_INEXACT`. For other argument values the behavior of these functions is undefined.

The IEEE standards require that exception flags represent, at least, whether the flag is set or clear; the functions `fetestexcept`, `feraiseexcept`, and `feclearexcept` support this basic abstraction. However, an implementation may endow exception flags with more information—for example, the address of the code which raised the exception; the functions `fegetexcept` and `fesetexcept` deal with the full content of flags.

In previous drafts of this specification, several of the exception functions returned an `int` indicating whether the `excepts` argument represented supported exceptions. This facility was deemed unnecessary because `excepts & ~FE_ALL_EXCEPT` can be used to test invalidity of the `excepts` argument.

4.4.1.1 The feclearexcept function

Synopsis

```
#include <fenv.h>
void feclearexcept(int excepts);
```

Description

The `feclearexcept` function clears the supported exceptions represented by its argument. The argument `excepts` represents exceptions as a bitwise OR of exception macros.

4.4.1.2 The fegetexcept function

Synopsis

```
#include <fenv.h>
void fegetexcept(fexcept_t *flagp, int excepts);
```

Description

The `fegetexcept` function stores an implementation-defined representation of the exception flags indicated by the argument `excepts` through the pointer argument `flagp`.

4.4.1.3 The feraiseexcept function

Synopsis

```
#include <fenv.h>
void feraiseexcept(int excepts);
```

Description

The `feraiseexcept` function raises the supported exceptions represented by its argument. The argument `excepts` represents exceptions as a bitwise OR of exception macros. The order in which these exceptions are raised is unspecified.

It is intended that enabled traps for exceptions raised by the `feraiseexcept` function are taken.

For IEEE Implementations

If the argument `excepts` represents coincident exceptions that are valid for atomic operations—namely overflow and inexact, or underflow and inexact—then overflow or underflow is raised before inexact.

The function is not restricted to accept only valid coincident expressions for atomic operations, so that the function can be used to raise exceptions accrued over several operations.

4.4.1.4 The fesetexcept function

Synopsis

```
#include <fenv.h>
void fesetexcept(const fexcept_t *flagp, int excepts);
```

Description

The `fesetexcept` function sets the complete status for those exception flags indicated by the argument `excepts`, according to the representation in the object pointed to by `flagp`. The value of `*flagp` must have been set by a previous call to `fegetexcept`; if not, the effect on the indicated exception flags is undefined. This function does not *raise* exceptions, but only sets the state of the flags.

4.4.1.5 The fetestexcept function

Synopsis

```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

The `fetestexcept` function determines which of a specified subset of the exception flags are currently set. The `excepts` argument specifies—as a bitwise OR of the exception macros—the exception flags to be queried.

Returns

The `fetestexcept` function returns the bitwise OR of the exception macros corresponding to the currently set exceptions included in `excepts`.

Example

Call `f` if invalid is set, `g` if overflow is set:

```
#pragma fenv_access on
int set_excepts;
set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
if (set_excepts & FE_INVALID) f();
if (set_excepts & FE_OVERFLOW) g();
```

This mechanism allows testing several exceptions with just one function call. The argument is a *mask* because querying all flags may be more expensive on some architectures.

4.4.2 Rounding

The `fegetround` and `fesetround` functions provide control of rounding direction modes.

The IEEE floating-point standards prescribe *rounding precision* modes (in addition to the rounding direction modes covered by the functions in this section) as a means for systems

whose results are always double or extended to mimic systems that deliver results to narrower formats. An implementation of C can meet this goal in any of the following ways:

1. By supporting the `float` minimum evaluation format (see §3.2.3).

2. By providing pragmas or compile options to shorten results by rounding to IEEE single or double precision.

3. By providing functions to set and get, dynamically, rounding precision modes which shorten results by rounding to IEEE single or double precision. Recommended are functions `fesetprec` and `fegetprec` and macros `FE_FLTPREC`, `FE_DBLPREC`, and `FE_LDBLPREC`, analogous to the functions and macros for the rounding direction modes.

This specification does not include a portable interface for precision control because the IEEE floating-point standards are ambivalent on whether they intend for precision control to be dynamic (like the rounding direction modes) or static. Indeed, some floating-point architectures provide control modes, suitable for a dynamic mechanism, and others rely on instructions to deliver single- and double-format results, suitable only for a static mechanism.

4.4.2.1 The fegetround function

Synopsis

```
#include <fenv.h>
int fegetround(void);
```

Description

The `fegetround` function gets the current rounding direction.

Returns

The `fegetround` function returns the value of the rounding direction macro representing the current rounding direction.

4.4.2.2 The fesetround function

Synopsis

```
#include <fenv.h>
int fesetround(int round);
```

Description

The `fesetround` function establishes the rounding direction represented by its argument `round`. If the argument does not match a rounding direction macro, the rounding direction is not changed.

Returns

The `fesetround` function returns a nonzero value if and only if the argument matches a rounding direction macro (that is, if and only if the requested rounding direction can be established).

Example

Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#pragma fenv_access on
int save_round;
save_round = fegetround();
int setround_ok = fesetround(FE_UPWARD);
assert(setround_ok);
. . .
fesetround(save_round);
```

4.4.3 Environment

The functions in this section manage the floating-point environment—exception flags, dynamic rounding modes, and any dynamic modes for handling floating-point exceptions—as one entity.

4.4.3.1 The fegetenv function

Synopsis

```
#include <fenv.h>
void fegetenv(fenv_t *envp);
```

Description

The `fegetenv` function stores the current floating-point environment in the object pointed to by `envp`.

4.4.3.2 The feholdexcept function

Synopsis

```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description

The `feholdexcept` function saves the current environment in the object pointed to by `envp`, clears the exception flags, and installs the non-stop mode, if available, for all exceptions. For implementations whose floating-point environment contains modes for exception handling, including a *non-stop* (continue on exceptions) mode, the `feholdexcept` function can be used in conjunction with the `feupdateenv` function to write routines that hide spurious exceptions from their callers.

Returns

The `feholdexcept` function returns nonzero if and only if non-stop exception handling was successfully installed.

`feholdexcept` should be effective on typical IEEE implementations which have the default non-stop mode and at least one other mode for trap handling or aborting. If the

implementation provides only the non-stop mode then installing the non-stop mode is trivial.

More appropriate for the user model prescribed in §2, `feholdexcept` supersedes `feprocentry` which is equivalent to

```
fegetenv(envp);
fesetenv(FE_DFL_ENV);
```

4.4.3.3 The fesetenv function

Synopsis

```
#include <fenv.h>
void fesetenv(const fenv_t *envp);
```

Description

The `fesetenv` function establishes the floating-point environment represented by the object pointed to by `envp`. The argument `envp` must point to an object set by a call to `fegetenv`, or equal the macro `FE_DFL_ENV` or an implementation-defined value of type *pointer to* `fenv_t`. Note that `fesetenv` merely installs the state of the exception flags represented through its argument, and does not raise these exceptions.

4.4.3.4 The feupdateenv function

Synopsis

```
#include <fenv.h>
void feupdateenv(const fenv_t *envp);
```

Description

The `feupdateenv` function saves the current exceptions in its automatic storage, installs the environment represented through `envp`, and then raises (actually re-raises) the saved exceptions. For implementations whose floating-point environment contains modes for exception handling, including a *non-stop* (continue on exceptions) mode, the `feupdateenv` function can be used in conjunction with the `feholdexcept` function to write routines that hide spurious exceptions from their callers.

`feupdateenv` was called `feprocexit` in earlier drafts of this specification.

Example

Hide spurious underflow exceptions:

```
#pragma fenv_access on
fenv_t save_env;
feholdexcept(&save_env);
...
if (... underflow is spurious ...) feclearexcept(FE_UNDERFLOW);
feupdateenv(&save_env);
```

APPENDIXES

A. Language Syntax Extensions (ISO §B; X3.159 §A)

This appendix summarizes language syntax extensions to Standard C.

A.1 Constants (ISO §B.1.4; X3.159 §A.1.4)

floating-constant:
...
hexadecimal-floating-constant

hexadecimal-floating-constant:
`0x` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
`0X` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
`0x` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$
`0X` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{opt}$ `.` *hexadecimal-digit-sequence*
hexadecimal-digit-sequence `.`

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

binary-exponent-part:
`p` *sign*$_{opt}$ *digit-sequence*
`P` *sign*$_{opt}$ *digit-sequence*

A.2 Preprocessor Numbers (ISO §B.1.9; X3.159 §A.1.9)

pp-number:
...
pp-number `p` *sign*
pp-number `P` *sign*

A.3 Pragmas

`#pragma fenv_access` *on-off-switch*

on-off-switch: one of:
`on` `off` `default`

`#pragma fp_contract` *on-off-switch*

`#pragma fp_wide_function_returns` *on-off-switch*
`#pragma fp_wide_function_parameters` *on-off-switch*
`#pragma fp_wide_variables` *on-off-switch*

B. Optimization

Many common optimizations for integer arithmetic are not generally valid (nor as useful) for floating-point. This appendix is an implementation note explaining some of the constraints.

B.1 Expression evaluation

For IEEE Implementations

Floating-point arithmetic operations and external function calls may entail side effects which optimization must honor, at least within the scope of an enabling `fenv_access` pragma (§2). The flags and modes in the floating-point environment may be regarded as global variables; floating-point operations (`+`, `*`, etc.) implicitly read the modes and write the flags.

Concern about side effects may inhibit code motion, removal of *useless* code, and certain transformations. For example, in

```
#pragma fenv_access on
double x;
...
for (i = 0; i < n; i++) x + 1;
```

`x + 1` might overflow, so cannot be removed. And since the loop body might not execute (maybe `0 ≥ n`), `x + 1` cannot be moved out of the loop. Of course these optimizations are valid if the implementation can rule out the nettlesome cases.

Within a basic block exceptions need not be precise. Thus the preceding loop could be treated as

```
if (0 < n) x + 1;
```

B.2 Expression transformations

The following transformations are not generally valid.

Removal of parentheses. This is not valid for any floating-point arithmetic. In particular, the implementation cannot apply the associative laws for `+` and `*`.

Factoring. Application of the distributive law is not valid for any floating-point arithmetic. For example, the implementation cannot replace `x + x * y` by `x * (1.0 + y)`.

x - x → 0. The expressions `x - x` and `0` are not equivalent if `x` is a NaN or infinite. (Of course, the two expression are not equivalent if `x` has side-effects.)

x - y ↔ -(y - x). For IEEE machines, in the default rounding direction, `1 - 1` is `+0` but `-(1 - 1)` is `-0`. The expressions `x - y` and `-(y - x)` are not equivalent on the CDC CYBER 205, because of abuse of 1's complement arithmetic. On the other hand, the

expressions `x - y`, `x + (-y)`, and `(-y) + x` are equivalent on IEEE machines, among others.

*x / 5.0 ↔ x * 0.2*. Such transformations involving constants generally do not yield numerically equivalent expressions. If the constants are exact then such transformations can be made on IEEE machines and others that round perfectly.

*0 * x → 0*. The expressions `0 * x` and `0` are not equivalent if `x` is a NaN or infinite.

*1 * x , x / 1 → x*. Neither `1 * x` and `x`, nor `x / 1` and `x`, are equivalent if `x` has almost overflowed on a CRAY-1 or CRAY-2 or if `x` has almost underflowed on a CDC CYBER 170.

x + 0 → x. For IEEE machines, if `x` is `-0` then, in the default rounding direction, `x + (+0)` yields `+0`, not `x`; the implementation cannot replace `x + 0` by `x` unless it can determine that `x` cannot be `-0`.

x - 0 → x. For IEEE machines, `(+0) - (+0)` yields `-0` when rounding is downward (toward $-\infty$), but `+0` otherwise, and `(-0) - (+0)` always yields `-0`; so, if the state of `fenv_access` were *off*, promising default rounding, then the implementation could replace `x - 0` by `x`, even if `x` might be zero.

-x ↔ 0 - x. For IEEE machines, `-(+0)` yields `-0`, but `0 - (+0)` yields `+0` (unless rounding is downward); the expressions are numerically equivalent for all other cases.

The IEEE floating-point standards prescribe a signed zero to preserve mathematical identities across certain discontinuities. Examples include

```
1 / (1 / (±INFINITY)) is ±INFINITY
complex_conjugate(sqrt(z)) is sqrt(complex_conjugate(z))
```

B.3 Relational operators

x != x → false. In IEEE implementations, the statement `x != x` is true if `x` is a NaN. In non-IEEE implementations with NaNs (reserved operands, indefinites) `x != x` may trap.

x == x → true. In IEEE implementations, the statement `x == x` is false if `x` is a NaN. In non-IEEE implementations with NaNs (reserved operands, indefinites) `x == x` may trap.

x < y → isless(x, y) (and similarly for <=, >, >=). Though equal, these expressions are not equivalent on IEEE implementations if `x` or `y` might be a NaN, and the state of `fenv_access` is *on*. This transformation, which would be desirable if extra code were required to cause the invalid exception for unordered cases, could be performed provided the state of `fenv_access` is *off*.

The sense of relational operators must be maintained. For IEEE implementations, this includes handling unordered cases as expressed by the source code. (Handling relationals correctly might not require generating worse code.)

Example

```
if (a < b) f(); else g();   /* calls g and raises invalid if a and
                                                    b are unordered */
```

is not equivalent to

```
if (a >= b) g(); else f();  /* calls f and raises invalid if a and
                                                    b are unordered */
```

nor to

```
if (isgreaterequal(a, b)) g(); else f();        /* calls f without
                            raising invalid if a and b are unordered */
```

nor, unless the state of `fenv_access` is *off*, to

```
if (isless(a, b)) f(); else g();        /* calls g without raising
                                    invalid if a and b are unordered */
```

but is equivalent to

```
if (!(a < b)) g(); else f();
```

B.4 Constant arithmetic

Under the effect of an enabling `fenv_access` pragma, the implementation must honor exceptions raised because of execution-time constant arithmetic. See §3.4, 3.5. Even under the effect of an enabling `fenv_access` pragma, an operation on constants that raises no exception can be folded during translation; implementations that support dynamic rounding precision modes (§4.4.2) should assure further that the result of the operation raises no exception when converted to the semantic type of the operation.

B.5 Wide representation

Implementations employing wide registers must take care to honor appropriate semantics. Values must be independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register must not alter the value. Also, an explicit *store and load* must round to the precision of the storage type. In particular, casts and assignments must perform their specified conversion (ISO §6.2.1.5; X3.159 §3.2.1.5): for

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float)expression;
```

the values assigned to `d1` and `d2` must have been converted to `float`.

C. Library Summary

This appendix summarizes library facilities specified, all or in part, in this document.

C.1 General utilities <stdlib.h> (ISO §D.11; X3.159 §C.11)

```
double atof(const char *nptr);
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

C.2 Input/output <stdio.h> (ISO §D.10; X3.159 §C.10)

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

C.3 Floating-point <fp.h>

```
HUGE_VAL
HUGE_VALF
HUGE_VALL
INFINITY
NAN
FP_NAN
FP_INFINITE
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
fpclassify(floating-expr)
signbit(floating-expr)
isfinite(floating-expr)
isnormal(floating-expr)
isnan(floating-expr)
DECIMAL_DIG
isgreater(floating-expr, floating-expr)
isgreaterequal(floating-expr, floating-expr)
isless(floating-expr, floating-expr)
islessequal(floating-expr, floating-expr)
islessgreater(floating-expr, floating-expr)
isunordered(floating-expr, floating-expr)
float_t
double_t
floating-type acos(floating-type x);
floating-type asin(floating-type x);
floating-type atan(floating-type x);
floating-type atan2(floating-type y, floating-type x);
floating-type cos(floating-type x);
floating-type sin(floating-type x);
floating-type tan(floating-type x);
floating-type acosh(floating-type x);
floating-type asinh(floating-type x);
floating-type atanh(floating-type x);
floating-type cosh(floating-type x);
floating-type sinh(floating-type x);
floating-type tanh(floating-type x);
floating-type exp(floating-type x);
floating-type exp2(floating-type x);
floating-type expm1(floating-type x);
```

```
floating-type frexp(floating-type value, int *exp);
floating-type ldexp(floating-type x, int exp);
floating-type log(floating-type x);
floating-type log10(floating-type x);
floating-type log1p(floating-type x);
floating-type log2(floating-type x);
floating-type logb(floating-type x);
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
floating-type scalb(floating-type x, long int n);
floating-type fabs(floating-type x);
floating-type hypot(floating-type x, floating-type y);
floating-type pow(floating-type x, floating-type y);
floating-type sqrt(floating-type x);
floating-type erf(floating-type x);
floating-type erfc(floating-type x);
floating-type gamma(floating-type x);
floating-type lgamma(floating-type x);
floating-type ceil(floating-type x);
floating-type floor(floating-type x);
floating-type nearbyint(floating-type x);
floating-type rint(floating-type x);
long int rinttol(long double x);
floating-type round(floating-type x);
long int roundtol(long double x);
floating-type trunc(floating-type x);
floating-type fmod(floating-type x, floating-type y);
floating-type remainder(floating-type x, floating-type y);
floating-type remquo(floating-type x, floating-type y, int *quo);
floating-type copysign(floating-type x, floating-type y);
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
floating-type nextafter(floating-type x, long double y);
float nextafterf(float x, float y);
double nextafterd(double x, double y);
long double nextafterl(long double x, long double y);
floating-type fdim(floating-type x, floating-type y);
floating-type fmax(floating-type x, floating-type y);
floating-type fmin(floating-type x, floating-type y);
```

C.4 Floating-point environment <fenv.h>

```
FE_INEXACT
FE_DIVBYZERO
FE_UNDERFLOW
FE_OVERFLOW
FE_INVALID
FE_ALL_EXCEPT
FE_TONEAREST
FE_UPWARD
FE_DOWNWARD
FE_TOWARDZERO
FE_DFL_ENV
fenv_t
fexcept_t
```

```
void feclearexcept(int excepts);
void fegetexcept(fexcept_t *flagp, int excepts);
void feraiseexcept(int excepts);
void fesetexcept(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);
int fegetround(void);
int fesetround(int round);
void fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
void fesetenv(const fenv_t *envp);
void feupdateenv(const fenv_t *envp);
```

D. Implementation Limits

§2.3.1 includes supplementary specification for the Standard C component

```
#define FLT_ROUNDS
```

and specification for additional components

```
#define _MIN_EVAL_FORMAT
#define _WIDEST_NEED_EVAL
```

E. Operator / Function Documentation Guide

Adapted from "Documentation of Special Properties of Programs that Compute Elementary Transcendental Functions in the Context of IEEE Standards 754/854 for Floating-Point Arithmetic, with advice for other systems too" (work in progress), by W. Kahan (July 1991).

This appendix previews a forthcoming guide to the documentation of floating-point operators and functions, including most of those in the C language and libraries. For each operator and function, the guide will discuss in detail the prescribed documentation.

"... Read by a system's user, this guide explores many of the issues that could undermine the portability of numerical software if not addressed. Read by a system's implementor, this guide explains what information ought to be supplied and offers a format for its presentation. An implementation's quality can be gauged by the absence of excuses for some of the sad compromises mentioned here. On the other hand, an implementor who resorted to one of those compromises without documenting it might expect users to regard that compromise more as a bug than a feature and to demand its correction.

"Real functions will be dealt with before complex. At first, only the IEEE's default rounding mode to-nearest will be presumed, and only the default responses to exceptions...

"For systems that do not conform to IEEE 754/854, the number ∞, if unavailable, may be approximated by the largest finite number. If no usable -0 is available, use just 0 instead. And if no NaN-like object is available, replace *NaN* by whatever is the system's usual response to an invalid operation, probably a trap or abortion of

execution. Similarly, serious exceptions like overflow and divide-by-zero, if they cannot raise an appropriate flag testable by the program, may have to trap or abort. In any event, serious disruption of a program should be accompanied by a user visible message more informative than, say, a hexadecimal dump. Subnormal numbers, if unusable, must be replaced by zeros with underflow signaled; of course, this signal may be unavailable or practically inaccessible, and hence usually ignored.

"For each function f(x) considered, implementors should supply the following tableau of information: "

Invalid Operands

Invalid operands are those x for which f(x) becomes a new NaN. Example: $x < 0$ for the square root function.

Exact Unexceptional Values

Document these for transcendental functions. Example: sin(±0) returns ±0. f(NaN) returning a NaN need not be included.

Exact Exceptional Values

f(x) returns ±∞ and raises the divide-by-zero exception. Example: log(0).

Inexact Exceptional Values

Overflow	f(x) returns ±∞ and raises the overflow exception.
Underflow	f(x) is subnormal or 0 and raises the underflow exception.
Divide-by-zero	—signaled when f(x) returns ±∞ though x is only approximately a pole, e.g. tan(π / 2).

Symmetries

Like f(-x) == f(x) if f is an even function.

Monotonicity

"(...more important than accuracy!)"

Accuracy

"Accuracy should be described, if not *correctly rounded*. Accuracy should normally be specified in terms of worst-case error in ulps (units in the last place). (Average error or standard deviation is an abuse of statistics, since worst case error is the killer.) Correctly rounded functions are accurate to within at worst half an ulp; most transcendental functions are very difficult to compute much more accurately than to within less than one ulp. Unfortunately, cases exist for which a kind of backward error-bound may be the best that can be described; this means that the computed value of f(x) may actually be within an ulp or two of f(X) where X is within an ulp or two of the given argument x. Such a bound is acceptable in situations where x is uncertain by a few ulps and uncorrelated with anything else that will be combined with f(x); otherwise such a bound has disagreeable consequences when x comes close to singularities of f, so it is a compromise that should be avoided whenever practical."

Other Noteworthy Properties

"In case *correctly rounded* is an impracticable accuracy goal, implementations may yet preserve important properties that would be preserved if every computed value were correctly rounded. Some, like monotonicity and symmetry, have already been mentioned; others are inequalities and identities that involve related functions, for instance `sqrt(x*x) == fabs(x)` absent over/underflow."

F. <fp.h> for IEEE Implementations (ISO §7.5; X3.159 §4.5)

This appendix contains recommended specification of library facilities that is particularly suited for IEEE implementations.

The Standard C macro `HUGE_VAL` and its `float` and `long double` analogs, `HUGE_VALF` and `HUGE_VALL`, expand to expressions whose values are positive infinities; their evaluations raise no exceptions.

`HUGE_VAL` could not be implemented as

```
#define HUGE_VAL (1.0/0.0)
```

whose use may raise the divide-by-zero exception.

Special cases for functions in `<fp.h>` are covered directly or indirectly by the IEEE floating-point standards. The functions `sqrt`, `remainder`, `rint`, and `rinttol` have direct counterparts in these standards. Appendixes to these standards specify the functions `copysign`, `logb`, `nextafter`, `scalb`, and `nearbyint`. The function `trunc` is equivalent to `rint` (or, at the implementor's option, to `nearbyint`) with rounding-toward-zero. The `ldexp` function is equivalent to the binary version of the `scalb` function specified in an appendix to the IEEE binary floating-point standard 754. The `fmod` function is readily derived from the standards' `remainder` function (see §F.7.1). The other functions are not covered explicitly by the IEEE floating-point standards; however, except as noted, this specification requires that they honor infinities, NaNs, signed zeros, subnormals, and the exception flags in a manner consistent with the basic arithmetic operations covered by the IEEE floating-point standards.

Detectable only through invocation of non-Standard C inquiries or exception handling, the exceptions required here are not in conflict with ISO §7.5.1 (X3.159 §4.5.1).

The overflow exception is raised whenever an infinity—or, because of rounding direction, a maximal-magnitude finite number—is returned in lieu of a value whose magnitude is too large. (See §2.)

The underflow exception is raised whenever a result is tiny (essentially subnormal or zero) and suffers loss of accuracy. This NCEG specification for transcendental functions allows raising the underflow (and inexact) exception when a result is tiny and *probably* inexact.

The IEEE floating-point standards offer the implementation multiple definitions of underflow. All resulting in the same values, the options differ only in that the thresholds when the exception is raised may differ by a rounding error.

According to the IEEE floating-point standards, an enabled underflow trap is taken if the result would be tiny, regardless of loss of accuracy.

The inexact exception is raised whenever the rounded result is not identical to the mathematical result. Except as noted, this NCEG specification for transcendental functions allows raising the inexact exception when a result is *probably* inexact.

For some functions, for example `pow`, determining exactness in all cases may be too costly.

As implied by §2.2.2, the `<fp.h>` functions do not raise spurious exceptions (detectable by the user). For example, the implementation must hide an underflow generated by an intermediate computation of a non-tiny result.

Whether transcendental functions honor the rounding direction mode and any rounding precision mode is implementation-defined.

Generally, one-parameter functions of a NaN argument return that same NaN and raise no exception. (See §3.1.1.1.)

For the Standard C `<math.h>` functions, the specification in the subsections of §F.1 appends to the Standard C definitions. For the NCEG extensions, it appends to the definitions in §4.3 of this document.

F.1 Trigonometric functions (ISO §7.5.2; X3.159 §4.5.2)

F.1.1 The acos function (ISO §7.5.2.1; X3.159 §4.5.2.1)

```
floating-type acos(floating-type x);
```

- `acos(1)` returns `+0`.
- `acos(x)` returns a NaN and raises the invalid exception for `|x| > 1`.

F.1.2 The asin function (ISO §7.5.2.2; X3.159 §4.5.2.2)

```
floating-type asin(floating-type x);
```

- `asin(±0)` returns ±0.
- `asin(x)` returns a NaN and raises the invalid exception for `|x| > 1`.

F.1.3 The atan function (ISO §7.5.2.3; X3.159 §4.5.2.3)

```
floating-type atan(floating-type x);
```

- `atan(±0)` returns ±0.
- `atan(±INFINITY)` returns ±π/2.

F.1.4 The atan2 function (ISO §7.5.2.4; X3.159 §4.5.2.4)

```
floating-type atan2(floating-type y, floating-type x);
```

- If one argument is a NaN then `atan2` returns that same NaN; if both arguments are NaNs then `atan2` returns one of its arguments.
- `atan2(±0, x)` returns ±0, for `x > 0`.
- `atan2(±0, +0)` returns ±0.

- `atan2(±0, x)` returns ±π, for `x < 0`.
- `atan2(±0, -0)` returns ±π.
- `atan2(y, ±0)` returns π/2 for `y > 0`.
- `atan2(y, ±0)` returns -π/2 for `y < 0`.
- `atan2(±y, INFINITY)` returns ±0, for finite `y > 0`.
- `atan2(±INFINITY, x)` returns ±π/2, for finite `x`.
- `atan2(±y, -INFINITY)` returns ±π, for finite `y > 0`.
- `atan2(±INFINITY, INFINITY)` returns ±π/4.
- `atan2(±INFINITY, -INFINITY)` returns ±3π/4.

The more contentious cases are `y` and `x` both infinite or both zeros. See [7] for a justification of the choices above.

`atan2(y, 0)` does not raise the divide-by-zero exception, nor does `atan2(0, 0)` raise the invalid exception.

F.1.5 The cos function (ISO §7.5.2.5; X3.159 §4.5.2.5)

floating-type `cos(`*floating-type* `x);`

- `cos(±INFINITY)` returns a NaN and raises the invalid exception.

F.1.6 The sin function (ISO §7.5.2.6; X3.159 §4.5.2.6)

floating-type `sin(`*floating-type* `x);`

- `sin(±0)` returns ±0.
- `sin(±INFINITY)` returns a NaN and raises the invalid exception.

F.1.7 The tan function (ISO §7.5.2.7; X3.159 §4.5.2.7)

floating-type `tan(`*floating-type* `x);`

- `tan(±0)` returns ±0.
- `tan(±INFINITY)` returns a NaN and raises the invalid exception.

F.2 Hyperbolic functions (ISO §7.5.3; X3.159 §4.5.3)

F.2.1 The acosh function

floating-type `acosh(`*floating-type* `x);`

- `acosh(1)` returns `+0`.
- `acosh(+INFINITY)` returns `+INFINITY`.
- `acosh(x)` returns a NaN and raises the invalid exception if `x < 1`.

F.2.2 The asinh function

floating-type `asinh(`*floating-type* `x);`

- `asinh(±0)` returns ±0.
- `asinh(±INFINITY)` returns `±INFINITY`.

F.2.3 The atanh function

floating-type `atanh(`*floating-type* `x);`

- `atanh(±0)` returns ±0.
- `atanh(±1)` returns ±`INFINITY`.
- `atanh(x)` returns a NaN and raises the invalid exception if `|x| > 1`.

F.2.4 The cosh function (ISO §7.5.3.1; X3.159 §4.5.3.1)

floating-type `cosh(`*floating-type* `x);`

- `cosh(±INFINITY)` returns `+INFINITY`.

F.2.5 The sinh function (ISO §7.5.3.2; X3.159 §4.5.3.2)

floating-type `sinh(`*floating-type* `x);`

- `sinh(±0)` returns ±0.
- `sinh(±INFINITY)` returns ±`INFINITY`.

F.2.6 The tanh function (ISO §7.5.3.3; X3.159 §4.5.3.3)

floating-type `tanh(`*floating-type* `x);`

- `tanh(±0)` returns ±0.
- `tanh(±INFINITY)` returns ±1.

F.3 Exponential and logarithmic functions (ISO §7.5.4; X3.159 §4.5.4)

F.3.1 The exp function (ISO §7.5.4.1; X3.159 §4.5.4.1)

floating-type `exp(`*floating-type* `x);`

- `exp(+INFINITY)` returns `+INFINITY`.
- `exp(-INFINITY)` returns `+0`.

F.3.2 The exp2 function

floating-type `exp2(`*floating-type* `x);`

- `exp2(+INFINITY)` returns `+INFINITY`.
- `exp2(-INFINITY)` returns `+0`.

F.3.3 The expm1 function

floating-type `expm1(`*floating-type* `x);`

- `expm1(±0)` returns ±0.
- `expm1(+INFINITY)` returns `+INFINITY`.
- `expm1(-INFINITY)` returns `-1`.

F.3.4 The frexp function (ISO §7.5.4.2; X3.159 §4.5.4.2)

```
floating-type frexp(floating-type value, int *exp);
```

- `frexp(±0, exp)` returns ±0, and returns 0 in `*exp`.
- `frexp(±INFINITY, exp)` returns ±INFINITY, and returns an unspecified value in `*exp`.
- `frexp` of a NaN argument is that same NaN, and returns an unspecified value in `*exp`.
- Otherwise, `frexp` raises no exception.

On a binary system, `frexp` is equivalent to the comma expression

```
( (*exp = (value == 0) ? 0 : (int)(1 + logb(value))),
   scalb(value, -(*exp)) )
```

F.3.5 The ldexp function (ISO §7.5.4.3; X3.159 §4.5.4.3)

```
floating-type ldexp(floating-type x, int exp);
```

On a binary system, `ldexp` is equivalent to

```
scalb(x, exp)
```

Note that `ldexp` may not provide the full functionality of `scalb` for extended values, because the power required to scale from the smallest (subnormal) to the largest extended value exceeds the minimum INT_MAX allowed by Standard C.

F.3.6 The log function (ISO §7.5.4.4; X3.159 §4.5.4.4)

```
floating-type log(floating-type x);
```

- `log(±0)` returns `-INFINITY` and raises the divide-by-zero exception.
- `log(x)` returns a NaN and raises the invalid exception if `x < 0`.
- `log(+INFINITY)` returns `+INFINITY`.

F.3.7 The log10 function (ISO §7.5.4.5; X3.159 §4.5.4.5)

```
floating-type log10(floating-type x);
```

- `log10(±0)` returns `-INFINITY` and raises the divide-by-zero exception.
- `log10(x)` returns a NaN and raises the invalid exception if `x < 0`.
- `log10(+INFINITY)` returns `+INFINITY`.

F.3.8 The log1p function

```
floating-type log1p(floating-type x);
```

- `log1p(±0)` returns ±0.
- `log1p(-1)` returns `-INFINITY` and raises the divide-by-zero exception.
- `log1p(x)` returns a NaN and raises the invalid exception if `x < -1`.
- `log1p(+INFINITY)` returns `+INFINITY`.

F.3.9 The log2 function

```
floating-type log2(floating-type x);
```

- `log2(±0)` returns `-INFINITY` and raises the divide-by-zero exception.
- `log2(x)` returns a NaN and raises the invalid exception if `x < 0`.
- `log2(+INFINITY)` returns `+INFINITY`.

F.3.10 The logb function

```
floating-type logb(floating-type x);
```

- `logb(±INFINITY)` returns `+INFINITY`.
- `logb(±0)` returns `-INFINITY` and raises the divide-by-zero exception.

F.3.11 The modf functions (ISO §7.5.4.6; X3.159 §4.5.4.6)

```
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

- `modf(value, iptr)` returns a result with the same sign as the argument `value`.
- `modf(±INFINITY, iptr)` returns `±0` and stores `±INFINITY` through `iptr`.
- `modf` of a NaN argument returns that same NaN and also stores it through `iptr`.

`modf` behaves as though implemented by

```
#include <fenv.h>
#include <fp.h>
#pragma fenv_access on
double modf(double value, double *iptr)
{
   int save_round = fegetround();
   fesetround(FE_TOWARDZERO);
   *iptr = nearbyint(value);
   fesetround(save_round);
   return copysign(
      (fabs(value) == INFINITY) ? 0.0 : value - (*iptr),
       value);
}
```

F.3.12 The scalb function

```
floating-type scalb(floating-type x, long int n);
```

- `scalb(x, n)` returns `x` if `x` is infinite, zero, or a NaN.
- `scalb` handles overflow and underflow like the basic IEEE standard arithmetic operations.

F.4 Power and absolute value functions (ISO §7.5.5, 7.5.6; X3.159 §4.5.5, 4.5.6)

F.4.1 The fabs function (ISO §7.5.6.2; X3.159 §4.5.6.2)

floating-type `fabs(`*floating-type* `x);`

- `fabs(±0)` returns `+0`.
- `fabs(±INFINITY)` returns `+INFINITY`.

F.4.2 The hypot function

floating-type `hypot(`*floating-type* `x,` *floating-type* `y);`

- `hypot(x, y)`, `hypot(y, x)`, and `hypot(x, -y)` are equivalent.
- `hypot(x, y)` returns `+INFINITY` if x is infinite.
- If both arguments are NaNs then `hypot` returns one of its arguments; otherwise, if `x` is a NaN and `y` in not infinite then `hypot` returns that same NaN.
- `hypot(x, ±0)` is equivalent to `fabs(x)`.

Note that `hypot(INFINITY, NAN)` returns `+INFINITY`, under the justification that `hypot(INFINITY, y)` is `+INFINITY` for any numeric value `y`.

F.4.3 The pow function (ISO §7.5.5.1; X3.159 §4.5.5.1)

floating-type `pow(`*floating-type* `x,` *floating-type* `y);`

- `pow(x, ±0)` returns `1` for any `x`.
- `pow(x, +INFINITY)` returns `+INFINITY` for `|x| > 1`.
- `pow(x, +INFINITY)` returns `+0` for `|x| < 1`.
- `pow(x, -INFINITY)` returns `+0` for `|x| > 1`.
- `pow(x, -INFINITY)` returns `+INFINITY` for `|x| < 1`.
- `pow(+INFINITY, y)` returns `+INFINITY` for `y > 0`.
- `pow(+INFINITY, y)` returns `+0` for `y < 0`.
- `pow(-INFINITY, y)` returns `-INFINITY` for `y` an odd integer `> 0`.
- `pow(-INFINITY, y)` returns `+INFINITY` for `y > 0` and not an odd integer.
- `pow(-INFINITY, y)` returns `-0` for `y` an odd integer `< 0`.
- `pow(-INFINITY, y)` returns `+0` for `y < 0` and not an odd integer.
- `pow(x, y)` returns one of its NaN arguments if `y` is a NaN, or if `x` is a NaN and `y` is nonzero.
- `pow(±1, ±INFINITY)` returns a NaN and raises the invalid exception.
- `pow(x, y)` returns a NaN and raises the invalid exception for finite `x < 0` and finite nonintegral `y`.
- `pow(±0, y)` returns `±INFINITY` and raises the divide-by-zero exception for `y` an odd integer `< 0`.
- `pow(±0, y)` returns `+INFINITY` and raises the divide-by-zero exception for `y < 0`, finite, and not an odd integer.
- `pow(±0, y)` returns `±0` for `y` an odd integer `> 0`.
- `pow(±0, y)` returns `+0` for `y > 0` and not an odd integer.

See [7].

pow(NaN, 0). [7] provides extensive justification for the value 1. An opposing point of view is that any return value of a function of a NaN argument should be a NaN, even if the

function is independent of that argument—as. in the IEEE floating-point standards, `NAN <= +INFINITY` is false and raises the invalid exception.

F.4.4 The sqrt functions (ISO §7.5.5.2; X3.159 §4.5.5.2)

floating-type `sqrt(`*floating-type* `x);`

`sqrt` is fully specified as a basic arithmetic operation in the IEEE floating-point standards.

F.5 Error and gamma functions

F.5.1 The erf function

floating-type `erf(`*floating-type* `x);`

- `erf(±0)` returns ±0.
- `erf(±INFINITY)` returns ±1.

F.5.2 The erfc function

floating-type `erfc(`*floating-type* `x);`

- `erfc(+INFINITY)` returns +0.
- `erfc(-INFINITY)` returns 2.

F.5.3 The gamma function

floating-type `gamma(`*floating-type* `x);`

- `gamma(+INFINITY)` returns `+INFINITY`.
- `gamma(x)` returns a NaN and raises the invalid exception if `x` is a negative integer or zero.
- `gamma(-INFINITY)` returns a NaN and raises the invalid exception.

F.5.4 The lgamma function

floating-type `lgamma(`*floating-type* `x);`

- `lgamma(+INFINITY)` returns `+INFINITY`.
- `lgamma(x)` returns `+INFINITY` and raises the divide-by-zero exception if `x` is a negative integer or zero.
- `lgamma(-INFINITY)` returns a NaN and raises the invalid exception.

F.6 Nearest integer functions (ISO §7.5.6; X3.159 §4.5.6)

F.6.1 The ceil function (ISO §7.5.6.1; X3.159 §4.5.6.1)

floating-type `ceil(`*floating-type* `x);`

- `ceil(x)` returns `x` if `x` is ±`INFINITY` or ±0.

The `double` version of `ceil` behaves as though implemented by

```
#include <fenv.h>
#include <fp.h>
#pragma fenv_access on
double ceil(double x)
{
   double result;
   int save_round = fegetround();
   fesetround(FE_UPWARD);
   result = rint(x); /* or nearbyint instead of rint */
   fesetround(save_round);
   return result;
}
```

F.6.2 The floor function (ISO §7.5.6.3; X3.159 §4.5.6.3)

floating-type `floor(`*floating-type* `x);`

- `floor(x)` returns `x` if `x` is ±`INFINITY` or ±`0`.

See the sample implementation for `ceil` in §F.6.1.

F.6.3 The nearbyint function

floating-type `nearbyint(`*floating-type* `x);`

The `nearbyint` function differs from the `rint` function only in that the `nearbyint` function does not raise the inexact flag.

F.6.4 The rint function

floating-type `rint(`*floating-type* `x);`

The `rint` function use IEEE standard rounding according to the current rounding direction. It raises the inexact exception if its argument differs in value from its result.

F.6.5 The rinttol function

```
long int rinttol(long double x);
```

The `rinttol` function provides floating-to-integer conversion as prescribed by the IEEE standards [2], [3] §5.4. It rounds according to the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified and the invalid exception is raised. When it raises no other exception and its argument differs from its result, `rinttol` raises the inexact exception.

F.6.6 The round function

floating-type `round(`*floating-type* `x);`

The `double` version of `round` behaves as though implemented by

```
#include <fenv.h>
#include <fp.h>
#pragma fenv_access on
double round(double x) {
   double result;
   fenv_t save_env;
   feholdexcept(&save_env);
   result = rint(x);
   if (fetestexcept(FE_INEXACT)) {
      fesetround(FE_TOWARDZERO);
      result = rint(copysign(0.5 + fabs(x), x));
   }
   feupdateenv(&save_env);
   return result;
}
```

`round` may but is not required to raise the inexact exception for nonintegral numeric arguments, as this implementation does.

F.6.7 The roundtol function

```
long int roundtol(long double x);
```

`roundtol` differs from `rinttol` with the default rounding direction just in that `roundtol` (1) rounds halfway cases away from zero and (2) may but need not raise the inexact exception for nonintegral arguments that round to within the range of `long int`.

F.6.8 The trunc function

```
floating-type trunc(floating-type x);
```

The `trunc` function uses IEEE standard rounding toward zero (regardless of the current rounding direction).

F.7 Remainder functions (ISO §7.5.6; X3.159 §4.5.6)

F.7.1 The fmod function (ISO §7.5.6.4; X3.159 §4.5.6.4)

```
floating-type fmod(floating-type x, floating-type y);
```

- If one argument is a NaN then `fmod` returns that same NaN; if both arguments are NaNs then `fmod` returns one of its arguments.
- `fmod(±0, y)` returns ±0 if `y` is not zero.
- `fmod(x, y)` returns a NaN and raises the invalid exception if `x` is infinite or `y` is zero.
- `fmod(x, ±INFINITY)` returns `x` if `x` is not infinite.

The `double` version of `fmod` behaves as though implemented by

```
#include <fp.h>
double fmod(double x, double y)
{
   double result;
   result = remainder(fabs(x), (y = fabs(y)));
   if (signbit(result)) result += y;
   return copysign(result, x);
}
```

F.7.2 The remainder function

floating-type `remainder(`*floating-type* `x,` *floating-type* `y);`

`remainder` is fully specified as a basic arithmetic operation in the IEEE floating-point standards.

F.7.3 The remquo function

floating-type `remquo(`*floating-type* `x,` *floating-type* `y, int *quo);`

`remquo` follows the specification for `remainder`. It has no further specification special to IEEE implementations.

F.8 Manipulation functions

F.8.1 The copysign function

floating-type `copysign(`*floating-type* `x,` *floating-type* `y);`

`copysign` is specified in appendices to the IEEE floating-point standards. It has no specification special to IEEE implementations.

F.8.2 The nan functions

```
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

All IEEE implementations support quiet NaNs, hence declare all the `nan` functions.

F.8.3 The nextafter functions

```
floating-type nextafter(floating-type x, long double y);
float nextafterf(float x, float y);
double nextafterd(double x, double y);
long double nextafterl(long double x, long double y);
```

- If one argument is a NaN then `nextafter` returns that same NaN; if both arguments are NaNs then `nextafter` returns one of its arguments.
- `nextafter(x, y)` raises the overflow and inexact exceptions if `x` is finite and the function value is infinite.
- `nextafter(x, y)` raises the underflow and inexact exceptions if the function value is subnormal and `x != y`.

F.9 Maximum, minimum, and positive difference functions

F.9.1 The fdim function

floating-type fdim(*floating-type* x, *floating-type* y);

• If one argument is a NaN then fdim returns that same NaN; if both arguments are NaNs then fdim returns one of its arguments.

F.9.2 The fmax function

floating-type fmax(*floating-type* x, *floating-type* y);

• If just one argument is a NaN then fmax returns the other argument; if both arguments are NaNs then fmax returns one of its arguments.

fmax might be implemented as

```
{ return (isgreaterequal(x, y) || isnan(y)) ? x : y; }
```

Ideally, fmax would be sensitive to the sign of zero, for example fmax(-0.0, +0.0) should return +0; however, implementation in software may be impractical.

Some applications may be better served by a *max* function that would return a NaN if one of its arguments were a NaN:

```
{ return (isgreaterequal(x, y) || isnan(x)) ? x : y; }
```

Note that two branches still are required, for symmetry in NaN cases.

F.9.3 The fmin function

floating-type fmin(*floating-type* x, *floating-type* y);

fmin is analogous to fmax. See §F.9.2.

Chapter 6

Complex C Extensions

Chapter 6

Complex C Extensions

Table of Contents

Foreword 6-iii

Introduction

Purpose 6-1
References 6-1
Organization of Document 6-2
Definition of Terms 6-2

Language

Types 6-3
Constants 6-4
Conversions 6-4
Imaginary types 6-4
Real and imaginary types 6-5
Complex types 6-5
Real and complex types 6-5
Imaginary and complex types 6-5
Usual arithmetic conversions 6-5
Expressions 6-7
Multiplicative operators 6-7
Additive operators 6-8
Relational operators 6-8
Equality operators 6-8
Expression evaluation methods 6-9

Libraries

Complex Extensions <complex.h> 6-9
Overloaded <fp.h> functions 6-10
Overload resolution 6-11
Branch cuts 6-11
Trigonometric functions 6-12
The acos overloads 6-12
The asin overloads 6-12
The atan overloads 6-12
The cos overloads 6-13
The sin overloads 6-13
The tan overloads 6-13
Hyperbolic functions 6-14
The acosh overloads 6-14
The asinh overloads 6-14
The atanh overloads 6-15
The cosh overloads 6-15
The sinh overloads 6-15
The tanh overloads 6-16
Exponential and logarithmic functions 6-16
The exp overloads 6-16
The log overloads 6-16
The sqrt overloads 6-17

Power and absolute value functions.....6-17
The fabs overloads.....6-17
The pow overloads.....6-17
Complex-specific functions.....6-18
The arg function.....6-18
The conj function.....6-18
The imag function.....6-19
The proj function.....6-19
The real function.....6-20

Foreword

At its initial meeting in 1989, the Numerical C Extensions Group (NCEG), now incorporated into the ANSI C language committee (X3J11), targeted complex arithmetic as one of the areas requiring supporting extensions, in order for C to become suitable for general numerical programming. The first NCEG complex proposal, "Complex Extension to C" by Knaak [7], followed a traditional FORTRAN-like approach, adding complex data types as ordered pairs of real numbers and specifying the arithmetic of these types. "Augmenting a Programming Language with Complex Arithmetic" by Kahan and Thomas [6] showed certain semantic and efficiency problems inherent in the traditional approach, including fundamental incompatibilities with advances in IEEE standard real arithmetic. To solve these problems, the current specification refines the traditional approach by adding both complex and imaginary types and specifying arithmetic on any combinations of real, imaginary, and complex operands. "Issues Regarding Imaginary Types for C and C++" by Thomas and Coonen [12] provides detailed rationale. Also, the current specification is designed for consistency with Chapter 5, "Floating-Point C Extensions". At its June 1994 meeting, X3J11 approved following the direction of the current specification. At its December 1994 meeting, X3J11 accepted the specification for inclusion in the Technical Report.

People who have made especially substantial contributions to this document include, in alphabetical order: Jerome Coonen, W. Kahan, Clayton Lewis, David Prosser, and Fred Tydeman.

Others who have provided invaluable contributions, reviews, or administrative help include, in alphabetical order: W.M. Gentleman, Bill Gibbons, Ron Guilmette, Doug Gwyn, David Hough, Rex Jaeschke, David Keaton, David Knaak, John Kwan, Tom MacDonald, Stuart McDonald, Colin McMaster, Randy Meyer, Randy Meyers, Adolfo, Nemirovsky, P.J. Plauger, Thomas Plum, and Pat Ricci.

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
jim_thomas@taligent.com

March 26, 1995

215

Chapter 6
Complex C Extensions

1. INTRODUCTION

1.1 Purpose

This document specifies a set of extensions to the C programming language, designed to support arithmetic with complex numbers. The extensions are suitable for any implementation of Standard C [1], augmented according to "Floating-Point C Extensions" [3]. For implementations supporting the IEEE floating-point standards [4], [5], these extensions allow complex arithmetic that is consistent with IEEE arithmetic.

This document is intended for the X3J11 technical report on numerical C extensions.

1.2 References

1. *International Standard Programming Languages—C*, (ISO/IEC 9899:1990 (E)).
2. *American National Standard for Information Systems*—Programming Language C (X3.159-1989).
3. "Floating-Point C Extensions", Jim Thomas (Chapter 5).
4. *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985).
5. *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE Std 854-1987).
6. "Augmenting a Programming Language with Complex Arithmetic", W. Kahan and J. W. Thomas (NCEG/91-039—November 15, 1991).
7. "Complex Extension to C" (Revision 11), David Knaak (WG14/N335/X3J11./94-020—April 28, 1994).
8. "Imaginary Types for Complex Extensions to C", Jim Thomas (X3J11.1/92-071—October 24, 1992).
9. "Merging Complex and IEEE-754", Frederick John Tydeman (X3J11.1/92-061—November, 1992).
10. *Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++* (X3J16/94-0098, WG21/N0485—May 26, 1994).

11. "What To Do About The Complex Extension To C?", David Knaak (X3J11.1/93-016—April 5, 1993).

12. "Issues Regarding Imaginary Types for C and C++", Jim Thomas and Jerome T. Coonen, *The Journal of C Language Translation*, Vol. 5, No. 3, Pages 134-138 (March 1994).

13. "A Proposal for Standard C++ Complex Number Classes", Al Vermeulen (X3J16/93-0165, WG21/N0372—1993).

14. "Templatized C++ Complex Number Classes", Tom Keffer (X3J16/94-01654 WG21/N0551—1994).

15. "Comments on complex vs imaginary types for C", W. M. Gentleman, (X3J11/94-047, WG14/N362—June 8, 1994).

For additional discussion about complex extensions, see Numerical C Extensions Group (NCEG) documents 89-002, 89-026, 89-036, 89-038, 90-005, 90-006, 90-030, 91-005, 91-026, and 91-053; X3J11.1 documents 92-031, 93-020; and WG14/X3J11 document N362/94-047.

1.3 Organization of Document

This document follows the style conventions of [3]. The major subsections are:

1. this introduction;
2. language syntax, constraints, and semantics;
3. library facilities.

1.4 Definition of Terms

Other relevant terms are defined in [3].

- Complex type—a basic type for representing complex values. Any of the types `float_complex`, `double_complex`, or `long_double_complex`.
- Floating type—a real floating, imaginary, or complex type (a change from [1], in which floating type refers to just real floating types).
- Imaginary type—a basic type for representing (pure) imaginary values. Any of the types `float_imaginary`, `double_imaginary`, or `long_double_imaginary`.
- Imaginary unit—the number *i* such that *i***i* = -1.
- Real floating type—any of the types `float`, `double`, or `long double`.
- Real type—an integral or real floating type.

2. LANGUAGE

2.1 Types (ISO §6.1.2.5; X3.159 §3.1.2.5)

The integral types, together with the `float`, `double`, and `long double` types, are collectively called the *real types*.

Inclusion of the header `<complex.h>` (see §3.1) enables designations for imaginary and complex types. The header defines macros

```
float_imaginary
double_imaginary
long_double_imaginary
float_complex
double_complex
long_double_complex
```

There are three *imaginary types*, designated as:

```
float_imaginary
double_imaginary
long_double_imaginary
```

Each has the same representation and alignment requirements as the corresponding real type.

imaginary	real
`float_imaginary`	`float`
`double_imaginary`	`double`
`long_double_imaginary`	`long double`

The value of an object of imaginary type is the value of the real representation times the imaginary unit.

Although not present in older complex arithmetic facilities, e.g. FORTRAN's, the imaginary types naturally model the imaginary axis of complex analysis, promote computational and storage efficiency, and capture the completeness and consistency of IEEE arithmetic for the complex domain. See [6], [12], [15], and rationale in §2.3.6 of this document for more discussion of imaginary types.

Because of their representation and alignment requirements, imaginary arguments can be used like real arguments for `fprintf` and `fscanf`.

There are three *complex types*, designated as:

```
float_complex
double_complex
long_double_complex
```

Each is represented internally by a contiguous pair of representations of the corresponding real type, aligned in an implementation-defined manner appropriate to the real type. The low-address and high-address elements of the pair represent the real part and imaginary part, respectively, of the complex value.

The underlying implementation of the complex types is Cartesian, rather than polar, for overall efficiency and consistency with other programming languages. The implementation is explicitly stated so that characteristics and behaviors can be defined simply and unambiguously.

The Standard C specification for basic and arithmetic types applies to the complex and imaginary types, except as specified otherwise.

An alternative would have been to introduce just two type macros, `complex` and `imaginary`, and to designate the complex and imaginary types as `float complex`, `float imaginary`, `double complex`, ... However, this approach would have ill-suited implementation as a C++ library.

2.2 Constants

Inclusion of the header `<complex.h>` allows designation of an imaginary unit constant. The header defines a macro `I` which expands to `_Imaginary_I`, a keyword (in the implementation's name space) denoting a constant with the imaginary type that corresponds to the minimum evaluation format [3] and with the value of the imaginary unit. For example, if the minimum evaluation format is `double`, then `I` has type `double_imaginary`.

Positing such a constant is a natural analog to the mathematical notion of augmenting the reals with the imaginary unit. It allows writing imaginary and complex expressions in common mathematical style, for example `x + y*I`. Note that the multiplication here affects translated code, but does not necessitate an actual floating-point multiply, nor does the addition necessitate a floating-point add.

The choice of `I` instead of `i` concedes to the widespread use of the identifier `i` for other purposes. The programmer can use a different identifier, say `j`, for the imaginary unit by following the inclusion of `<complex.h>` with

```
#undef I
#define j _Imaginary_I
```

The `i` suffix to designate imaginary constants, proposed in [7], is not required. Multiplication by `I` provides a sufficiently convenient and more generally useful notation for imaginary terms.

2.3 Conversions (ISO §6.2; X3.159 §3.2)

This specification includes conversion between any complex type and any arithmetic type and between any imaginary type and any arithmetic type.

2.3.1 Imaginary types

Conversions among imaginary types follow rules analogous to those for real floating types.

2.3.2 Real and imaginary types

When a value of imaginary type is converted to a real type, the result is a positive zero or an unsigned zero.

When a value of real type is converted to an imaginary type, the result is a positive zero or an unsigned zero.

2.3.3 Complex types

When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for corresponding real types.

2.3.4 Real and complex types

When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero.

When a value of complex type is converted to a real type, the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

2.3.5 Imaginary and complex types

When a value of imaginary type is converted to a complex type, the real part of the complex result value is a positive zero or an unsigned zero and the imaginary part of the complex result value is determined by the conversion rules for corresponding real types.

When a value of complex type is converted to an imaginary type, the real part of the complex value is discarded and the value of the imaginary part is converted according to the conversion rules for corresponding real types.

2.3.6 Usual arithmetic conversions (ISO §6.2.1.5; X3.159 §3.2.1.5)

The real and imaginary parts of imaginary and complex operands and results follow the pattern of the usual arithmetic conversions for corresponding real types. For example, the product of a `float_complex` and a `double` (or a `double_imaginary` or `double_complex`) entails a promotion of the `float_complex` operand to `double_complex`, and the semantic result type is `double_complex`.

Operations do not entail conversion of operands among real, imaginary, and complex types. For example, the product of a `double_imaginary` and a `float_complex` does not entail a conversion of the imaginary operand to complex.

Automatic conversion of real or imaginary operands to complex would require extra computation, while producing undesirable results in certain cases involving infinities, NaNs, and signed zeros.

Examples

With automatic conversion to complex,

$$\begin{aligned} 2.0 * (3.0 + \infty i) &\Rightarrow (2.0 + 0.0i) * (3.0 + \infty i) \\ &\Rightarrow (2.0*3.0 - 0.0*\infty) + (2.0*\infty + 0.0*3.0)i \\ &\Rightarrow \mathrm{NaN} + \infty i \end{aligned}$$

rather than the desired result, $6.0 + \infty i$.

Optimizers for implementations with infinities—including all IEEE ones—would not be able to eliminate the operations with the zero imaginary part of the converted operand.

The following example illustrates the problem pertaining to signed zeros; [6] explains why it matters. With automatic conversion to complex,

$$\begin{aligned} 2.0 * (3.0 - 0.0i) &\Rightarrow (2.0 + 0.0i) * (3.0 - 0.0i) \\ &\Rightarrow (2.0*3.0 + 0.0*0.0) + (-2.0*0.0 + 0.0*3.0)i \\ &\Rightarrow 6.0 + 0.0i \end{aligned}$$

rather than the desired result, $6.0 - 0.0i$.

Imaginary types

The problems illustrated in the examples above have counterparts for imaginary operands. The mathematical product $2.0i * (\infty + 3.0i)$ should yield $-6.0 + \infty i$. With automatic conversion to complex,

$$\begin{aligned} 2.0i * (\infty + 3.0i) &\Rightarrow (0.0 + 2.0i) * (\infty + 3.0i) \\ &\Rightarrow (0.0*\infty - 2.0*3.0) + (0.0*3.0 + 2.0*\infty)i \\ &\Rightarrow \mathrm{NaN} + \infty i \end{aligned}$$

This also demonstrates the need for imaginary types. Without them, $2.0i$ would have to be represented as $0.0 + 2.0i$, implying that $\mathrm{NaN} + \infty i$ would be the semantically correct result—regardless of conversion rules. And optimizers for implementations with infinities—including all IEEE ones—would not be able to eliminate the operations with the zero real part.

In general, the imaginary types, together with the conversion rules and operator specifications (below), allow substantially more efficient implementation. For example, multiplication of a real or imaginary by a complex can be implemented straightforwardly with two multiplications, instead of four multiplications and two additions.

Some programs are expected to use the imaginary types implicitly in constructions with the imaginary unit `I`, such as `x + y*I`, and not explicitly in declarations. This suggests making the imaginary types private to the implementation and not available for explicit program declarations. However, such an approach was rejected as being less in the open spirit of C, and not much simpler. For the same reasons, the approach of treating imaginary-ness as an attribute of certain complex expressions, rather than as additional types, was rejected.

Another approach, put forth in [11], would regard the special values—infinities, NaNs, and signed zeros—as outside the model. This would allow any behavior when special values occur, including much that is prescribed by this specification. However, this approach would not serve the growing majority of implementations, including all IEEE ones, that support the special values. In order to provide a consistent extension of their treatment of special cases in real arithmetic, these implementations would require yet another specification in addition to the one suggested in [11]. On the other hand, implementations

not supporting special values should have little additional trouble implementing imaginary types as proposed here.

The efficiency benefits of the imaginary types goes beyond what the implementation provides. In many cases programmers have foregone a programming language's complex arithmetic facilities, which, lacking an imaginary type, required contiguous storage of both real and imaginary parts; programmers could store and manipulate complex values more efficiently using real arithmetic directly [15]. The imaginary types enable programmers to exploit the efficiency of the real formats, without having to give up support for complex arithmetic semantics.

2.4 Expressions (ISO §6.3; X3.159 §3.3)

2.4.1 Multiplicative operators (ISO §6.3.5; X3.159 §3.3.5)

If one operand has real type and the other operand has imaginary type, then the result has imaginary type.

If both operands have imaginary type, then the result has real type.

If any operand has complex type, then the result has complex type.

The values of the imaginary and complex types are precisely the values of $y*I$ and $x + y*I$, respectively, where x and y are values of the corresponding real floating type and I is the value of the imaginary constant `I`. Hence, the following tables, taken from [6], describe the types and mathematical results of multiplications and divisions involving real, imaginary, and complex operands. x, y, u, and v denote real values.

Multiply

*	x	$y*I$	$x + y*I$
u	$x*u$	$(y*u)*I$	$(x*u) + (y*u)*I$
$v*I$	$(x*v)*I$	$-y*v$	$(-y*v) + (x*v)*I$
$u + v*I$	$(x*u) + (x*v)*I$	$(-y*v) + (y*u)*I$	$(x*u - y*v) +$ $(y*u + x*v)*I$

Divide

/	x	$y*I$	$x + y*I$
u	x/u	$(y/u)*I$	$(x/u) + (y/u)*I$
$v*I$	$(-x/v)*I$	y/v	$(y/v) + (-x/v)*I$
$u + v*I$	$(x*u/(u*u + v*v)) +$ $(-x*v/(u*u + v*v))*I$	$(y*v/(u*u + v*v)) +$ $(y*u/(u*u + v*v))*I$	$((x*u+y*v)/(u*u+v*v))+$ $((y*u-x*v)/(u*u+v*v))*I$

The computation of the product and quotient of complex values should guard against undue overflow and underflow. [6] presents algorithms for computing products and quotients. Also see [9].

2.4.2 Additive operators (ISO §6.3.6; X3.159 §3.3.6)

If one operand has real type and the other operand has imaginary type, then the result has complex type.

If both operands have imaginary type, then the result has imaginary type.

If any operand has complex type, then the result has complex type.

The following table, taken from [6], describes the types and results of addition and subtraction involving real, imaginary, and complex operands. x, y, u, and v denote real values.

Add/subtract

±	x	$v*I$	$x + v*I$
u	$x \pm u$	$\pm u + v*I$	$(x \pm u) + v*I$
$v*I$	$x \pm v*I$	$(v \pm v)*I$	$x + (y \pm v)*I$
$u + v*I$	$(x \pm u) \pm v*I$	$\pm u + (v \pm v)*I$	$(x \pm u) + (v \pm v)*I$

Note that some operations can be handled entirely at translation time, without floating-point arithmetic. Examples include $y * I$, $x + v * I$, and $I * I$.

2.4.3 Relational operators (ISO §6.3.8; X3.159 §3.3.8)

The arithmetic operands of the relational operators are constrained to be real.

Some mathematical practice would be supported by defining the relational operators for complex operands so that `z1` *op* `z2` would be true if and only if both `real(z1)` *op* `real(z2)` and also `imag(z1) == imag(z2)`. Believing such use to be uncommon, NCEG voted against including this specification.

2.4.4 Equality operators (ISO §6.3.9; X3.159 §3.3.9)

Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types (including imaginary and complex) are equal if and only if the results of their conversion to the complex type of width determined by the usual arithmetic conversions are equal.

For example,

```
0 == -0.0*I
```

is true, because (1) the usual arithmetic conversions promote the integer `0` to `double` (to match the other operand), (2) the values 0.0 and -0.0*I convert to the `double_complex` type as 0.0 + 0.0*I and 0.0 - 0.0*I, and (3) -0.0 equals 0.0 arithmetically, even if not bitwise.

2.4.5 Expression evaluation methods

The expression evaluation methods described in [3] govern evaluation formats of expressions involving imaginary and complex types. For example, if the minimum evaluation format is `double`, then the product of two `float_complex` operands is represented in the `double_complex` format, and its parts are evaluated to `double`.

3. LIBRARIES

3.1 Complex Extensions <complex.h>

The header `<complex.h>` defines macros and declares functions that support complex arithmetic.

The macro

```
__COMPLEX__
```

expands to the constant `1`, indicating inclusion of `<complex.h>`.

The macros

```
float_imaginary
double_imaginary
long_double_imaginary
float_complex
double_complex
long_double_complex
```

are defined.

They might expand to distinct keywords in the implementation's name space, for example,

```
#define float_complex _Float_complex
```

If the macro names later become standard keywords, then the implementor can remove the macro definitions from the header and directly support the names in the translator.

The macro

```
I
```

expands to `_Imaginary_I`, a keyword in the implementation's name space, denoting a constant with the imaginary type that corresponds to the minimum evaluation format and with the value of the imaginary unit.

See rationale in §2.2.

Lacking an imaginary type, [7] required macros in order to create certain special values. For example, an "imaginary" infinity could be created by `CMPLX(0.0,INFINITY)`. With the imaginary type, imaginary infinity is simply the value of `INFINITY*I`. ([3] defines the

INFINITY macro.) And, in general, the values of $y*I$ and $x + y*I$, where x and y are real floating values, cover all values of the imaginary and complex types, hence eliminating this need for the complex macros.

The subsequent sections specify the declarations for functions involving complex and imaginary types. Semantic details, though outside the scope of this document, should be documented thoroughly by the implementation. [9] specifies semantics for IEEE implementations, and for other implementations with similar features.

3.1.1 Overloaded <fp.h> functions

When the header `<complex.h>` is included, some function designators from `<fp.h>` [3] become overloaded with additional prototypes having complex and imaginary parameters. The most common form for the extra prototypes is

floating-type function-designator (*complex-or-imaginary-type*) ;

Functions with extra prototypes of this form are:

acos	cos	acosh	cosh	exp	fabs
asin	sin	asinh	sinh	log	
atan	tan	atanh	tanh	sqrt	

For each of these functions, the implementation provides a set of prototypes—one for each complex and imaginary parameter type whose parts have the minimum evaluation format or wider format. The format of the part(s) of the result matches the format of the part(s) of the parameter. The return type for `fabs` is always real. Otherwise, for all these functions, if the parameter is complex then so is the return type. If the parameter is imaginary then the return type is real, imaginary, or complex, as appropriate for the particular function; in particular, if the parameter is imaginary, then the return types of `cos` and `cosh` are real, the return types of `sin`, `tan`, `sinh`, `tanh`, `asin`, and `atanh` are imaginary, and the return types of the others are complex.

Example

If the minimum evaluation format were `double` then for `cos` the implementation would have

```
double_complex cos(double_complex);
long_double_complex cos(long_double_complex);
double cos(double_imaginary);                    /* cos(x*I) == cosh(x) */
long double cos(long_double_imaginary);
```

Exploiting the fact that some functions map the imaginary axis onto the real or imaginary axis gains more efficient calculation involving imaginaries, and better meets user expectations in some cases. However, dropping out of the complex domain may lead to surprises as subsequent operations may be done with real functions, which generally are more restrictive than their complex counterparts. For example, `sqrt(-cos(I))` invokes the real `sqrt` function, which is invalid for the negative real value `-cos(I)`, whereas the complex `sqrt` is valid everywhere.

The `pow` function has extra prototypes of the form:

complex-type `pow(`*floating-type, floating-type*`)`

Each form represents prototypes such that the parts of both parameters and the part(s) of the return type have the same format. For each combination of parameter kinds—real-floating, imaginary, and complex—with at least one imaginary or complex parameter, there is a prototype for the minimum evaluation format and each wider format. Thus the implementation provides 8 prototypes for each pertinent format.

Example

If the minimum evaluation format is `long double`, then the implementation provides

```
long_double_complex pow(long double, long_double_imaginary);
long_double_complex pow(long double, long_double_complex);
long_double_complex pow(long_double_imaginary, long double);
long_double_complex pow(long_double_imaginary, long_double_imaginary);
long_double_complex pow(long_double_imaginary, long_double_complex);
long_double_complex pow(long_double_complex, long double);
long_double_complex pow(long_double_complex, long_double_imaginary);
long_double_complex pow(long_double_complex, long_double_complex);
```

3.1.1.1 Overload resolution

For a call to one of these functions, the implementation determines the evaluation format for the call according to the overload resolution rules described in [3], §4.3.1. These rules are based on argument width, which for complex arguments means the width of the parts. Then the implementation chooses from among its prototypes the one matching the evaluation format and whose parameters are of the same kind—real, imaginary, or complex—as the arguments.

Example

If the minimum evaluation format is `float`, then in

```
double d;
float_complex fc;
float_imaginary fi;
...
fc = d * sin(fi);
```

an implementation without widest-need would choose the `float_imaginary` prototype for `sin`. An implementation with widest-need would choose the `double_imaginary` prototype for `sin`.

3.1.1.2 Branch cuts

Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEEE implementations), the sign of zero distinguishes one side of a cut from another so the function is continuous (except for format limitations) as the cut is approached from either side. For example, for the square root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part +0, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part -0, maps to the negative imaginary axis.

Implementations with an unsigned zero cannot distinguish the sides of branch cuts. They map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the

functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

3.1.2 Trigonometric functions

The header `<complex.h>` declares complex and imaginary overloads for the following Standard C trigonometric functions.

3.1.2.1 The acos overloads

Synopsis

```
#include <complex.h>
complex-type acos(complex-or-imaginary-type z);
```

Description

The `acos` overloads compute the complex arc cosine of `z`, with branch cuts outside the interval [-1, 1] along the real axis.

Returns

The `acos` overloads return the complex arc cosine of `z`, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

3.1.2.2 The asin overloads

Synopsis

```
#include <complex.h>
imaginary-type asin(imaginary-type z);
complex-type asin(complex-type z);
```

Description

The `asin` overloads compute the complex arc sine of `z`, with branch cuts outside the interval [-1, 1] along the real axis.

Returns

The `asin` overloads return the complex arc sine of `z`, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-π/2, π/2] along the real axis.

3.1.2.3 The atan overloads

Synopsis

```
#include <complex.h>
complex-type atan(complex-or-imaginary-type z);
```

Description

The `atan` overloads compute the complex arc tangent of `z`, with branch cuts outside the interval $[-i, i]$ along the imaginary axis.

Returns

The `atan` overloads return the complex arc tangent of `z`, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, \pi/2]$ along the real axis.

3.1.2.4 The cos overloads

Synopsis

```
#include <complex.h>
real-type cos(imaginary-type z);
complex-type cos(complex-type z);
```

Description

The `cos` overloads compute the complex cosine of `z`.

Returns

The `cos` overloads return the complex cosine of `z`.

3.1.2.5 The sin overloads

Synopsis

```
#include <complex.h>
imaginary-type sin(imaginary-type z);
complex-type sin(complex-type z);
```

Description

The `sin` overloads compute the complex sine of `z`.

Returns

The `sin` overloads return the complex sine of `z`.

3.1.2.6 The tan overloads

Synopsis

```
#include <complex.h>
imaginary-type tan(imaginary-type z);
complex-type tan(complex-type z);
```

Description

The `tan` overloads compute the complex tangent of `z`.

Returns

The `tan` overloads return the complex tangent of `z`.

3.1.3 Hyperbolic functions

The header `<complex.h>` declares complex and imaginary overloads for the following Standard C hyperbolic functions and their arc counterparts, defined in [3].

3.1.3.1 The acosh overloads

Synopsis

```
#include <complex.h>
complex-type acosh(complex-or-imaginary-type z);
```

Description

The `acosh` overloads compute the complex arc hyperbolic cosine of `z`, with a branch cut at values less than 1 along the real axis.

Returns

The `acosh` overloads return the complex arc hyperbolic cosine of `z`, in the range of a half-strip of non-negative values along the real axis and in the interval [$-i\pi$, $i\pi$] along the imaginary axis.

3.1.3.2 The asinh overloads

Synopsis

```
#include <complex.h>
complex-type asinh(complex-type z);
```

Description

The `asinh` overloads compute the complex arc hyperbolic sine of `z`, with branch cuts outside the interval [$-i$, i] along the imaginary axis.

Returns

The `asinh` overloads return the complex arc hyperbolic sine of `z`, in the range of a strip mathematically unbounded along the real axis and in the interval [$-i\pi/2$, $i\pi/2$] along the imaginary axis.

3.1.3.3 The atanh overloads

Synopsis

```
#include <complex.h>
imaginary-type atanh(imaginary-type z);
complex-type atanh(complex-type z);
```

Description

The `atanh` overloads compute the complex arc hyperbolic tangent of `z`, with branch cuts outside the interval [-1, 1] along the real axis.

Returns

The `atanh` overloads return the complex arc hyperbolic tangent of `z`, in the range of a strip mathematically unbounded along the real axis and in the interval [$-i\pi/2$, $i\pi/2$] along the imaginary axis.

3.1.3.4 The cosh overloads

Synopsis

```
#include <complex.h>
real-type cosh(imaginary-type z);
complex-type cosh(complex-type z);
```

Description

The `cosh` overloads compute the complex hyperbolic cosine of `z`.

Returns

The `cosh` overloads return the complex hyperbolic cosine of `z`.

3.1.3.5 The sinh overloads

Synopsis

```
#include <complex.h>
imaginary-type sinh(imaginary-type z);
complex-type sinh(complex-type z);
```

Description

The `sinh` overloads compute the complex hyperbolic sine of `z`.

Returns

The `sinh` overloads return the complex hyperbolic sine of `z`.

3.1.3.6 The tanh overloads

Synopsis

```
#include <complex.h>
imaginary-type tanh(imaginary-type z);
complex-type tanh(complex-type z);
```

Description

The `tanh` overloads compute the complex hyperbolic tangent of `z`.

Returns

The `tanh` overloads return the complex hyperbolic tangent of `z`.

3.1.4 Exponential and logarithmic functions

The header `<complex.h>` declares complex and imaginary overloads for the following Standard C exponential and logarithmic functions.

3.1.4.1 The exp overloads

Synopsis

```
#include <complex.h>
complex-type exp(complex-or-imaginary-type z);
```

Description

The `exp` overloads compute the complex base-e exponential of `z`.

Returns

The `exp` overloads return the complex base-e exponential of `z`.

3.1.4.2 The log overloads

Synopsis

```
#include <complex.h>
complex-type log(complex-or-imaginary-type z);
```

Description

The `log` overloads compute the complex natural (base-e) logarithm of `z`, with a branch cut along the negative real axis.

Returns

The `log` overloads return the complex natural logarithm of `z`, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi, i\pi]$ along the imaginary axis.

3.1.4.3 The sqrt overloads

Synopsis

```
#include <complex.h>
complex-type sqrt(complex-or-imaginary-type z);
```

Description

The `sqrt` overloads compute the complex square root of `z`, with a branch cut along the negative real axis.

Returns

The `sqrt` overloads return the complex square root of `z`, in the range of the right half-plane.

3.1.5 Power and absolute value functions

The header `<complex.h>` declares complex and imaginary overloads for the Standard C power and absolute value functions.

3.1.5.1 The fabs overloads

Synopsis

```
#include <complex.h>
real-type fabs(complex-or-imaginary-type z);
```

Description

The `fabs` overloads compute the absolute value (also called norm, modulus, or magnitude) of `z`.

See §3.1.6.4 rationale.

Returns

The `fabs` overloads return the absolute value of `z`.

3.1.5.2 The pow overloads

Synopsis

```
#include <complex.h>
complex-type pow(floating-type, floating-type)
```

Description

The `pow` overloads compute the complex power function of `z`, with a branch cut for the first parameter along the negative real axis.

Returns

The `pow` overloads return the complex power function of `z`.

3.1.6 Complex-specific functions

The header `<complex.h>` declares functions pertaining specifically to complex arithmetic. They are overloaded to allow arguments of real, complex, or imaginary type. Note that in the overloading forms below, *`floating-type`* includes complex and imaginary types (unlike in [3]). For a call to one of these functions, the implementation follows the overloading resolution rules described in §3.1.1.

3.1.6.1 The arg function

Synopsis

```
#include <complex.h>
real-type arg(floating-type z);
```

Prototypes follow the same pattern as for `fabs` (§3.1.1, [3] §4.3.5.1).

Description

The `arg` function computes the argument or phase angle of `z`, with a branch cut along the negative real axis.

Returns

The `arg` function returns the argument or phase angle of `z`, in the range $[-\pi, \pi]$.

3.1.6.2 The conj function

Synopsis

```
#include <complex.h>
floating-type conj(floating-type z);
```

Return types match parameter types.

Description

The `conj` function computes the complex conjugate of `z`, by reversing the sign of its imaginary part, if any.

Note that `conj(3.0)` yields 3.0, not 3.0 - 0.0*i*.

Returns

The `conj` function returns the complex conjugate of `z`.

3.1.6.3 The imag function

Synopsis

```
#include <complex.h>
real-type imag(floating-type z);
```

Prototypes follow the same pattern as for `fabs` (§3.1.1, [3] §4.3.5.1).

Description

The `imag` function computes the imaginary part of `z`.

Returns

The `imag` function returns the imaginary part of `z`.

3.1.6.4 The proj function

Synopsis

```
#include <complex.h>
floating-type proj(floating-type z);
```

The return type is real if the parameter is real; the return type is complex if the parameter is complex or imaginary.

Description

The `proj` function computes a projection of `z` onto the Riemann sphere: `z` projects to `z` except that all infinities, even ones with one infinite part and one NaN part, project to positive infinity on the real axis.

Returns

The `proj` function returns a projection of `z` onto the Riemann sphere.

Two topologies are commonly used in complex mathematics: the complex plane with its continuum of infinities and the Riemann sphere with its single infinity. The complex plane is better suited for transcendental functions, the Riemann sphere for algebraic functions. The complex types with their multiplicity of infinities provide a useful (though imperfect) model for the complex plane. The `proj` function helps model the Riemann sphere by mapping all infinities to one, and should be used just before any operation, especially comparisons, that might give spurious results for any of the other infinities.

Note that a complex value with one infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is infinite, the complex value is infinite independent of the value of the other part. For the same reason, `fabs` returns an infinity if its argument has an infinite part and a NaN part.

3.1.6.5 The real function

Synopsis

```
#include <complex.h>
real-type real(floating-type z);
```

Return types follow the same pattern as for `imag` (§3.1.6.3).

Description

The `real` function computes the real part of `z`.

Returns

The `real` function returns the real part of `z`.

Chapter 7

Data Parallel C Extensions

Chapter 7

Data Parallel C Extensions

Table of Contents

FOREWORD v
1. INTRODUCTION 1
1.1 PURPOSE 1
1.2 SCOPE 1
1.3 REFERENCES 1
1.4 ORGANIZATION OF THE DOCUMENT 3
1.5 BASE DOCUMENTS 3
1.6 DEFINITION OF TERMS 4
1.7 COMPLIANCE 5
1.8 FUTURE DIRECTIONS 5
2. ENVIRONMENT 6
2.1 CONCEPTUAL MODELS 6
2.1.2 Execution environments 6
3. LANGUAGE 7
3.1 LEXICAL ELEMENTS 7
3.1.1 Keywords 7
3.1.2.5 Types 7
3.1.2.6 Compatible and composite types 9
3.1.5 Operators 11
3.2 CONVERSIONS 11
3.2.1 Arithmetic Operands 11
3.2.1.1 Characters and integers 11
3.2.1.5 Usual arithmetic conversions 11
3.2.2 Other Operands 12
3.2.2.1 Lvalues and Function Designators 12
3.2.2.3 Pointers 12
3.2.3 Parallel Operands and Contextualization 13
3.3 EXPRESSIONS 15
3.3.1 Primary Expressions 15
3.3.2 Postfix Operators 16
3.3.2.1 Array subscripting 16
3.3.2.2 Function calls 18
3.3.2.3 Structure and union members 26
3.3.2.4 Postfix increment and decrement operators 27
3.3.3 Unary Operators 28
3.3.3.1 Prefix increment and decrement operators 29
3.3.3.2 Address and indirection operators 30
3.3.3.3 Unary arithmetic operators 34
3.3.3.4 The **sizeof** operator 35
3.3.3.5 The **shapeof** operator 36
3.3.3.6 Parallel indexing 36
3.3.3.7 Unary reduction operators 44
3.3.4 Cast Operators 45
3.3.5 Multiplicative Operators 47
3.3.6 Additive Operators 48
3.3.7 Bitwise Shift Operators 51
3.3.8 Relational Operators 52

3.3.9 Equality Operators 54
3.3.10 Bitwise **AND** Operator 55
3.3.11 Bitwise Exclusive **OR** Operator 55
3.3.12 Bitwise Inclusive **OR** Operator 56
3.3.13 Logical **AND** Operator 56
3.3.14 Logical **OR** Operator 57
3.3.15 Conditional Operator 57
3.3.16 Assignment Operators 59
3.3.16.1 Simple assignment 60
3.3.16.2 Compound assignment 61
3.3.17 Comma Operator 63
3.4 CONSTANT EXPRESSIONS 63
3.5 DECLARATIONS 64
3.5.2 Type Specifiers 66
3.5.4 Declarators 67
3.5.4.1 Pointer declarators 68
3.5.4.2 Array declarators 70
3.5.4.3 Function declarators 70
3.5.4.4 Shape declarators 71
3.5.4.5 Parallel object declarators 74
3.5.5 Type names 75
3.5.7 Initialization 76
3.6 STATEMENTS 77
3.6.4 Selection statements 77
3.6.5 Iteration statements 77
3.6.5.3 The **for** statement 77
3.6.6 Jump statements 78
3.6.6.4 The **return** statement 78
3.6.7 Contextualization 82
3.6.7.1 The **where** statement 83
3.6.7.2 The **everywhere** statement 84
3.7 EXTERNAL DEFINITIONS 85
3.7.1 Function Definitions 85
4. LIBRARY 92
4.1 INTRODUCTION 92
4.1.2 Standard Headers 92
4.1.6 Use of Library Functions 92
4.14 DATA PARALLEL UTILITIES 92
4.14.1 General purpose DPCE utilities 92
4.14.1.1 The **dimof** Function 94
4.14.1.2 The **layoutof** Function 95
4.14.1.3 The **newshape** Function 96
4.14.1.4 The **nodeof** Function 97
4.14.1.5 The **nodepositionsof** Function 98
4.14.1.6 The **palloc** Function 99
4.14.1.7 The **pcoord** Function 100
4.14.1.8 The **pfree** Function 101
4.14.1.9 The **positionsof** Function 101

4.14.1.10 The **prand** Function 102
4.14.1.11 The **psrand** Function 102
4.14.1.12 The **rankof** Function 103
4.14.2 DPCE mathematics 104
4.14.3 DPCE string handling 104
APPENDICES
A. OTHER PROPOSED EXTENSIONS 105
B. EXISTING IMPLEMENTATIONS 108
INDEX 109

Foreword

This technical report is the result of discussions that began within the Array Syntax subgroup of the Numerical C Extensions Group (NCEG) in May 1989. The subgroup, to emphasize its primary focus, became known as the Data Parallel C Extensions (DPCE) subcommittee.

This report addresses only a subset of the issues that were discussed by the subcommittee. There were many areas of contention in defining these extensions. The subcommittee chose to consider only those areas where consensus could be reached. Hence, not every proposed extension or viewpoint is represented in this report: this should not be construed as denying their utility or merit. For example, there was great interest in including some form of parallel control flow, but it was agreed that since no consensus could be reached after considerable debate on this topic, that the subcommittee would leave it for a future extension. There was also interest in including overloading, to simplify the extension of library functions to handle data parallel objects, but its inclusion at this time was rejected, since it was not essential to the overall goals of the language extensions. Appendix A describes proposed extensions that were not included, along with a summary of why those extensions were not accepted.

The scope of this report is intentionally limited to detailing only the most fundamental concepts of a consistent data parallel model. The extensions described in this report have been designed to allow further extensions to be cleanly added in the future. The subcommittee believes that further experience with the data parallel paradigm will eventually allow consensus to be obtained for a more broadly defined C extension.

The subcommittee began by adopting a general model that included the basic concept of parallel data aggregates that have structure (rank and dimensions), memory layout (possibly noncontiguous), and context (active or participating elements). The subcommittee then began deliberations on how parallel aggregates should be accessed and used in expressions and statements.

As these concepts were already embodied in the C* language developed at Thinking Machines Corporation, the subcommittee adopted the C* reference manual [4] as its base document. It proceeded with removing features of the C* language that were not considered essential to the model, and adding more extensions, notably in the area of elemental and nodal functions, pointers to parallel, and parallel pointers.

The process has been a lengthy one, and the culmination of the subcommittee's work is described here in a thorough examination of the needed extensions to each section of the C standard. This exercise itself has identified and addressed inconsistencies in the model, and has improved and focused the report.

The DPCE subcommittee wishes to thank the following persons for their valuable contributions to its deliberation process:

Analog Devices, Marc Hoffman
Analog Devices, Kevin Leary
Analog Devices, Alex Zatsman
Control Data Corporation, Azar Hashemi
Convex Corporation, Austin Curley
Convex Corporation, Bill Torkelson
Cray Research, Incorporated, Tom MacDonald
Cray Research, Incorporated, Dave Becker
David Sarnoff Research Center, Maya Gokhale
Digital Equipment Corporation, Randy Meyers
Digital Equipment Corporation, Jeffrey Zeeb
Farance Inc, Frank Farance
Hewlett Packard, John Kwan
HyperParallel Technologies, Christian Fortunel
HyperParallel Technologies, Nicolas Paris
IBM, Pawel Molenda
IBM, Bill O'Farrell
Keaton Consulting, David Keaton
Lawrence Livermore National Laboratory, Linda Stanberry
MasPar, David Alpern
Mimosa Systems, Incorporated, Hugh Redelmeier
Open Software Foundation, Mike Meissner
Pacific-Sierra Research, David McNamara
Plum Hall, Tom Plum
SunPro, Bob Jervis
SunPro, Marino Segnan
Supercomputing Research Center, Howard Gordon
Supercomputing Research Center, Aaron Naiman
Syracuse University, Pankaj Kumar
Syracuse University, Nancy McCracken
Thinking Machines Corporation, James L. Frankel
Thinking Machines Corporation, Gary Sabot
Thinking Machines Corporation, Guy Steele
Tydeman Consulting, Fred Tydeman
University of Arizona, Peter Bigot
University of New Hampshire, Phil Hatcher
Unix System Laboratories, David Prosser
US Army, Doug Gwyn

Linda Stanberry, Technical Editor
Lawrence Livermore National Laboratory
P.O. Box 808, L-300
Livermore, CA 94551
lstanberry@llnl.gov

Data Parallel C Extensions

1. INTRODUCTION [ISO §1, X3.159 §1]

1.1 PURPOSE [ISO §1, X3.159 §1.1]

This document describes a set of extensions to Standard C that supports programming of data parallel applications. The intent is to provide a set of machine-independent extensions that permit an efficient mapping to high-performance archictectures, especially massively parallel architectures.

1.2 SCOPE [ISO §1, X3.159 §1.2]

This document describes only the data parallel extensions to the C Standard. It presents those extensions in the context of the relevant sections of the Standard to be modified, and introduces new subsections of the Standard where appropriate. It does not provide a tutorial on data parallel programming, nor on Standard C.

1.3 REFERENCES [ISO §2, Annex A, X3.159 §1.3]

1. *American National Standard for Information Systems–Programming Language C* (X3.159-1989). Note: this is now withdrawn and replaced by [2].

2. *International Standard Programming Languages–C* (ISO/IEC 9899:1990(E)).

3. "C* Programming Guide," Thinking Machines Corporation (X3J11.1/90-032).

4. "A Reference Description of the C* Language," James L. Frankel (X3J11.1/91-023).

5. "ASX Evaluation Method - Revision 2," Frank Farance (X3J11.1/92-004).

6. "Massively Parallel C: Architectures and Data Distribution," Tom MacDonald (X3J11.1/92-007).

7. "C* Language Model," James L. Frankel (X3J11.1/92-010).

8. "C* answers to evaluation criteria," James L. Frankel (X3J11.1/92-011).

9. "Expressing Communication Costs in an Array Syntax," Dave Becker (X3J11.1/92-025).

10. "Focusing the ASX Base Document," Bob Jervis (X3J11.1/92-026).

11. "Issues concerning the use of C* as a base document," Frank Farance (X3J11.1/92-028).

12. "Distributing Data Using the 'block' Qualifier in C (revision 2)," Dave Becker, Kent Zoya, Bill Homer (X3J11.1/92-033).

13. "Elemental Execution," Phil Hatcher (X3J11.1/92-041).

14. "Left Indexing versus Right Indexing," Frank Farance (X3J11.1/92-044).

(CONTINUED)

15. "ASX Ten Commandments," Frank Farance (X3J11.1/92-045).

16. "A Critique of the Programming Language C*," Walter F. Tichy, Michael Phillipsen, and Phil Hatcher (X3J11.1/92-050).

17. "Commentary on 'A Critique of the Programming Language C*'," Phil Hatcher (X3J11.1/92-051).

18. "A Detailed Response to the C* Critique by Tichy, Phillipsen, and Hatcher," James Frankel (X3J11.1/92-053).

19. "The Pros and Cons of Current Shape in C*," James L. Frankel (X3J11.1/92-054).

20. "A Proposed Worklist of Extensions/Changes to C*," James L. Frankel (X3J11.1/92-055).

21. "Parallel Processing Model for High Level Programming Languages (3/92)," Cherri Pancake (X3J11.1/92-056).

22. "MasPar's C Directions and Reasons," David Alpern (X3J11.1/92-062).

23. "Parallel Control Flow Constructs," David Alpern (X3J11.1/92-073).

24. "Elemental Functions," Phil Hatcher (X3J11.1/92-076).

25. "HyperC, A C language for Data Parallelism," HyperParallel Technologies (X3J11.1/92-081).

26. "FORALL Proposal for Base Document," Gary Sabot (X3J11.1/93-008).

27. "A Parallel Extension to ANSI C," Rob E. H. Kurver (X3J11.1/93-009).

28. "Nodal Functions: A Strawman," Phil Hatcher (X3J11.1/93-011).

29. "Parallel Pointer Handles," James L. Frankel (X3J11.1/93-013).

30. "Using Iterators to Express Parallel Operations in C (revision 4)," Dave Becker, Kent Zoya, Bill Homer (X3J11.1/93-050).

31. "dbC Reference Manual," J.D. Schlesinger and M. Gokhale (Technical Report SRC-TR-93-109, November, 1993).

32. "Adding Shapes to Iterators," Bill Homer (X3J11/94-021).

33. "DPCE Array Slicing Proposal," Frank Farance (X3J11/94-025).

1.4 ORGANIZATION OF THE DOCUMENT [X3.159 §1.4]

This document is organized into sections that correspond to the relevant sections to be modified within the Standard C document. Included with each extension is a brief rationale or example for the extension.

> If rationale for an extension is included, it is distinguished by indentation and a change of font such as this.

The major sections of the document are:

1. Introduction
2. Environment
3. Language
4. Library

Each subsection of these major sections follows the structure of ANSI C [1] and ISO C [2], and indicates which subsections are modified. Cross references are noted at the beginning of each subsection, enclosed in square brackets–e.g., [ISO §7.1, X3.159 §4.1]. The numbering of all subsections directly corresponds to the numbering within the cited ANSI/ISO standard. Subsections for which there is no corresponding ANSI or ISO subsection are new [NEW]. Subsections of the standard which are not affected are skipped, so the numbering of subsections within this proposal will not necessarily be consecutive.

In each subsection of this document, the text is to be considered as amplifying the existing text of that subsection of the Standard, not replacing or modifying it. Where parts of specific definitions in existing subsections of the standard are modified, the modification is introduced by an italicized and underlined heading to that effect, such as:

Revise:

These headings are used for clarity as needed, and omitted where it is obvious that the entire change is reflected in the text. For brevity, an ellipsis (...) is used to indicate that omitted text is the same as in the Standard.

Change bars (|) are affixed in the right margin on each paragraph that has changed since the previous version of this document.

1.5 BASE DOCUMENTS [ISO §2, X3.159 §1.5]

This set of extensions represents the composition of multiple proposals from participating representatives, as reflected in the list of references in §1.3. Early in the deliberations, the committee elected to adopt the C* language reference manual [4] as its base document, and with this as its foundation, derived the current set of extensions by deleting some features of C* and adding new features.

1.6 DEFINITION OF TERMS [ISO §3, X3.159 §1.6]

The following new terms are used throughout this document. Although it would be more natural to define each term in the subsection where it is first introduced, it is also convenient to have the new terms dealing specifically with the data parallel extensions collected in one place. Hence, the most widely used new terms are defined here.

- active position – a position whose values participate in elemental execution.
- context – the component of a shape that designates the active positions of a parallel operand.
- dimension – the number of positions along an axis of a shape.
- element – the value or object at a position within a parallel operand, respectively; or a member of an array.
- elemental execution – execution of a function or operation on elements within corresponding active positions of parallel operands.

  An operation performed under a context is executed elementally. That is, it is executed on each value or object at the positions designated as active for a given context.
- layout – information specifying a distribution of a parallel object or parallel value onto memory.

  Memory refers to the total composite memory of a computing system.
- node – an actual processing unit in the execution environment.

  A node usually refers to a hardware processing unit, but is implementation dependent to allow for diverse parallel environments.
- parallel indexing – selecting elements of a parallel operand; single elements or multiple elements may be selected.
- parallel object – a structured collection of one or more identically-sized objects where the structure is defined by a shape.

  A parallel object is distinct from an ordinary C object in that although it is composed of C objects, the collection itself is not guaranteed to be contiguously allocated.
- parallel operand – parallel value or parallel object.
- parallel pointer – a parallel operand whose element type is pointer type.
- parallel value – a structured collection of one or more identically-typed values where the structure is defined by a shape.
- physical – a predefined variable of type shape which is of rank 1 and dimension equal to the number of nodes in the execution environment.
- pointer to parallel – a pointer type whose referenced type is a parallel type.

- position – a point within the index space defined by the Cartesian product of the dimensions of a shape.

  A position of a given shape denotes a point in all variables of that shape.

- rank – the number of dimensions or axes in a shape.
- reduction – an operation that when applied to a parallel operand produces a single, nonparallel value, such as the sum of all the elements of a parallel object.
- shape – a type whose values consist of the following components: rank, dimensions, layout, and context.

  Objects and values of type shape are descriptors or templates for parallel objects or parallel values. Variables may be declared to denote objects of type shape. See §3.1.2.5 and §3.5.4.4.

1.7 COMPLIANCE [ISO §4, X3.159 §1.7]

In order to comply with this set of extensions, an implementation shall provide for all the extensions detailed in this document.

1.8 FUTURE DIRECTIONS [X3.159 §1.8]

The set of extensions here is intended as the minimal set of extensions needed to support data parallel programming. As this is a relatively new area of expertise, the DPCE subcommittee chose not to propose extensions in those directions where more experience is needed to evaluate alternate proposals. As such experience is gained, further data parallel extensions will be desirable to codify developing practice and promote portability of data parallel applications.

2. ENVIRONMENT [ISO §5, X3.159 §2]

2.1 CONCEPTUAL MODELS [ISO §5.1, X3.159 §2.1]

These extensions are based on a data parallel model of programming. This model provides a single thread of control while allowing the manipulation of parallel objects. Parallel objects are manipulated by applying in parallel an operation across all the elements of the objects. In addition to the single thread model, DPCE also offers two mechanisms which allow the programmer to directly utilize a multithreaded model: nodal function and elemental function invocations.

The data parallel model supports a large class of parallel computations while being easy to learn and use. The ease of use is derived from its emphasis on a single thread of control, as used in serial programming. That is, this model is easier for users with respect to program design, debugging, and maintenance. The wide applicability is due in part to the demonstrated ability of compilers to translate data parallel programs for efficient execution on a variety of both serial and parallel hardware platforms.

Although the programmer's model presents the illusion of a single thread of control to simplify the program design, implementation, debugging, and maintenance tasks, the execution model may utilize many independent, unsynchronized threads or processes.

Examples

The following illustrates a comparison of programming style that one would use to perform the same operations on a parallel object using DPCE as one would use in ISO C with arrays. The ISO C example is not truly equivalent to the DPCE example since the DPCE operations are not ordered as the operations are ordered in the ISO C loops. Further, in DPCE, operations are performed under context and the granularity is at the operation level rather than at the statement level. Although the two code segments are not equivalent for the reasons noted, both demonstrate the same effect.

```
/* DPCE */
shape [100]S;
int:S x, y, z;


x = y + z;

x += 17;
```

```
/* ISO C */
typedef int ShapedInt[100];
ShapedInt x, y, z;
int i;

for (i=0; i<100; i++)
   x[i] = y[i] + z[i];
for (i=0; i<100; i++)
   x[i] += 17;
```

2.1.2 Execution environments [ISO §5.1.2, X3.159 §2.1.2]

At program startup, the DPCE execution environment is established by the inclusion of the header file **<dpce.h>**. This environment shall define **physical** to denote a predefined variable of type shape which is of rank 1 and dimension equal to the number of nodes in the execution environment. The layout of **physical** is implementation-defined.

3. LANGUAGE [ISO §6, X3.159 §3]

3.1 LEXICAL ELEMENTS [ISO §6.1, X3.159 §3.1]

3.1.1 Keywords [ISO §6.1.1, X3.159 §3.1.1]

The following keywords are added to the language only if the **<dpce.h>** header file is included. In addition to these keywords, **<dpce.h>** defines the **physical** shape identifier, and the functions described in §4.14 of this document.

Add the following new keywords:

```
block
elemental
everywhere
nodal
scale
shape
shapeof
where
```

3.1.2.5 Types [ISO §6.1.2.5, X3.159 §3.1.2.5]

Revise object types and incomplete types to include parallel objects:

... Types are partitioned into *object types* (types that describe objects or parallel objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects or parallel objects but lack information needed to determine their sizes).

Add shape type:

There is one *shape type*, designated as **shape**.

The shape type is an object type whose values consist of the following components: rank, dimensions, layout, and context. A value of type shape is referred to as "a shape."

A shape type whose rank and dimensions are not known is *fully unspecified*. A shape type whose rank is known, but whose dimensions are not, is *partially specified*. A shape type whose rank and dimensions are known is *fully specified*. These three categories of shape types form three distinct subsets of the shape type. Note: in a declaration of a shape, either none or all of the dimensions must be specified (see §3.5.4.4).

A **void** shape designator can be used to specify a generic shape type. See §3.5 and §3.3.2.2.

Examples

```
shape S;                    /* Fully unspecified shape */
shape []T;                  /* Partially specified shape */
shape [100]U;               /* Fully specified shape */
shape [2][]V;               /* Constraint violation */

int:void f(int:void arg); /* Function that takes a parallel
                             int and returns a parallel int
                             of generic shape */
```

Revise derived types to include parallel types:

Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows:

...

- A *parallel type* describes a nonempty, structured collection of objects or values with a particular member type, called the *element type*. The structure of the collection is defined by an *associated shape*. Parallel types are characterized by their element type and their shape. A parallel type is said to be derived from its element type and its shape. If its element type is *T* and its shape is *S*, the parallel type is called "a parallel *T* " or "a parallel *T* of shape *S*." The construction of a parallel type from an element type and a shape is called "parallel type derivation."

  For example, a parallel type with element type `int` and shape S is called "a parallel `int`" or "a parallel `int` of shape S." A parallel type with element type `struct` is called "a parallel `struct`." A parallel type with pointer element type is called "a parallel pointer."

Revise recursive applicability of constructing derived types:

These methods of constructing derived types can be applied recursively, except:

- the element type of a parallel type shall not be or contain a parallel type or shape type;
- a structure or union type shall not contain a member that has parallel type or a member that is a shape;

  Hence, you can have arrays of parallel types, functions returning parallel types or having parallel-typed arguments, pointers to parallel types, parallel types whose elements are structs or unions or functions or pointers. But you can't have parallel types whose elements are parallel types or contain parallel types, or are shape types or contain shape types. Nor can you have parallel types whose elements are array types (see §3.5).

  Note that the element type of a parallel type may be or contain a pointer to parallel type, as a pointer to parallel is not a parallel type. Hence, a parallel pointer may have element type "pointer to parallel."

Revise integral and floating types to include parallel types:

Parallel types whose element types are integral types are called *parallel integral types*. Parallel types whose element types are floating types are called *parallel floating types*.

Revise arithmetic and aggregate types to include parallel types:

Parallel integral and floating types are collectively called *parallel arithmetic types*. Array, structure, and parallel types are collectively called *aggregate types*.

Revise pointer storage requirements:

A pointer to a parallel type need not have the same representation as a pointer to the corresponding nonparallel type. The equivalent for parallel types of a pointer to **void** is a pointer to **void:void**.

3.1.2.6 Compatible and composite types [ISO §6.1.2.6, X3.159 §3.1.2.6]

Add for compatible shape types:

Two shape types are compatible under the following conditions:

- A fully unspecified shape type is compatible with any other shape type.
- Two partially specified shapes are compatible only if they specify the same rank.
- A partially specified shape type is compatible with a fully specified shape type if the partially specified shape type has the same rank as the fully specified shape type.
- Two fully specified shape types are compatible only if they specify the same rank, dimensions, and layout (see §3.2.3).
- A **void** shape type is compatible with any other shape type.

Examples

```
/* Compatible and incompatible shape types */
shape [10]S;
shape [10]T;              /* Compatible with S */
shape []U;                /* Compatible with S and T */
shape [][]V;              /* Incompatible with S, T, and U */
shape [2][5]W;            /* Incompatible with S, T, and U */
shape [][][]X;            /* Incompatible with S, T, U, V, and W */
shape Y;                  /* Compatible with S, T, U, V, W, and X */
```

Add for compatible parallel types:

Two parallel types are compatible if they are derived from compatible element types and compatible shape types.

Revise for qualifiers in composite function types and composite shape types:

A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

...

- If one type is a qualified function type and the other is an unqualified function type, the composite type is the qualified function type.

  To form a composite function type from an **elemental**-qualified or **nodal**-qualified function type and an unqualified function type, the unqualified versions of the function types must be compatible, and the composite type will be the qualified function type. This allows the inclusion of <dpce.h> to coexist with the unqualified declarations of the standard C library functions that it redeclares as **elemental**.

Examples

```
void *memset();
#include <string.h>
#include <dpce.h>          /* Provides elemental definitions */
#include <math.h>
...
{
  shape [100][100]S;
  double x;
  double:S y;

  ... sin(x) ...           /* Invokes nonparallel version of sin */
  ... sin(y) ...           /* Invokes sin elementally for each
                              element of y */

  memset(&x,0,sizeof(x));
                           /* Invokes nonparallel version of
                              memset */

  memset((double:S *:S)&y,0,sizeof(y));
                           /* Invokes memset elementally for each
                              element of y */
}
```

- If one type is a fully unspecified shape and the other is a fully specified shape, the composite type is identical to the fully specified shape type.
- If one type is a fully unspecified shape and the other is a partially specified shape, the composite type is identical to the partially specified shape type.
- If one type is a partially specified shape and the other is a fully specified shape, the composite type is identical to the fully specified shape type.
- If one type is a void shape and the other is a nonvoid shape, the composite type is the nonvoid shape type.

To form a composite shape type, the shapes have to be compatible, and the composite will always be the more completely specified shape type.

Examples

```
shape [10]S;
shape T;                   /* Composite type of S and T is S */
shape []U;                 /* Composite type of S and U is S */
                           /* Composite type of T and U is U */
void f(int:void x) { ... }
int:S y;
f(y);                      /* Composite type of void and S is S */
```

3.1.5 Operators [ISO §6.1.5, X3.159 §3.1.5]

Add new operators:

`<?` `<?=` `>?` `>?=` `%%`

3.2 CONVERSIONS [ISO §6.2, X3.159 §3.2]

3.2.1 Arithmetic Operands [ISO §6.2.1, X3.159 §3.2.1]

3.2.1.1 Characters and integers [ISO §6.2.1.1, X3.159 §3.2.1.1]

Add to the integral promotions:

A parallel **`char`**, a parallel **`short int`**, or a parallel **`int`** bit field, or their signed or unsigned varieties, or a parallel enumeration type, may be used in an expression wherever a parallel **`int`** or parallel **`unsigned int`** may be used. If a parallel **`int`** can represent all values of the original type, the value is converted to a parallel **`int`**; otherwise, it is converted to a parallel **`unsigned int`**; the promoted parallel value will be of the same shape as the original expression.

Integral promotions are applied elementally to parallel operands.

3.2.1.5 Usual arithmetic conversions [ISO §6.2.1.5, X3.159 §3.2.1.5]

Add for parallel operands:

Arithmetic conversions performed on a parallel operand result in a parallel value of the same shape as the operand and whose value at each position is the result of performing the usual arithmetic conversions on the value at the corresponding position of the operand.

The usual arithmetic conversions are applied elementally to parallel operands, and the result is an homogenous parallel operand.

3.2.2 Other Operands [ISO §6.2.2, X3.159 §3.2.2]

3.2.2.1 Lvalues and Function Designators [ISO §6.2.2.1, X3.159 §3.2.2.1]

Add for parallel lvalues:

A *parallel lvalue* is an lvalue that designates a parallel object.

3.2.2.3 Pointers [ISO §6.2.2.3, X3.159 §3.2.2.3]

Revise:

A pointer to `void` may be converted to or from a pointer to any incomplete nonparallel type or nonparallel object type. A pointer to any incomplete nonparallel type or nonparallel object type may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.

Add:

A pointer to `void:void` may be converted to or from a pointer to any incomplete parallel type or parallel object type. A pointer to any incomplete parallel type or parallel object type may be converted to a pointer to `void:void` and back again; the result shall compare equal to the original pointer.

A pointer to an array of, or containing, any incomplete parallel type or parallel object type may be converted to a pointer to `void:void` and back again; the result shall compare equal to the original pointer.

> Since a pointer to an array is also a pointer to the first element of the array, a pointer to an array of parallel is also a pointer to parallel.

A null pointer constant cast to type `void:void *` is called a *null pointer-to-parallel constant*. If a null pointer constant or a null pointer-to-parallel constant is assigned to or compared for equality to a pointer to parallel, the constant is converted to a pointer to parallel of that type. Such a pointer, called a *null pointer to parallel* is guaranteed to compare unequal to a pointer to any parallel object.

Two null pointers to parallel, converted through possibly different sequences of casts to pointers to parallel types, shall compare equal.

3.2.3 Parallel Operands and Contextualization [NEW]

Except as specified later for the array subscripting, parallel indexing, and ?: operators, the following conditions shall hold.

In binary operations involving operands of parallel types, there are both compile-time and run-time requirements. At compile time, the shape types of the operands shall be compatible (see §3.1.2.6). At run time, the shapes of the operands shall be fully assigned (see §3.5.4.4) and their values shall be the same; otherwise the behavior is undefined. That is, for parallel operands a and b,

```
if (*shapeof(a) == *shapeof(b))
                /* a op b is defined */
else
                /* a op b is undefined */
```

Two shapes are the same only if their values are identical; that is, only if they have the same rank, dimensions, layout and context. The following function returns 1 if the two shapes `S1` and `S2` are the same, and 0 otherwise.

```
int same_shape(shape S1, shape S2)
{
  int rank;
  int i, j;
  int:S1 context1;
  int:S2 context2;
  int same_context;

  rank = rankof(S1);
  if (rank != rankof(S2)) return(0);

  for (i=0; i<rank; i++) {
    if (dimof(S1,i) != dimof(S2,i)) return(0);
    if (layoutof(S1,i) != layoutof(S2,i)) return(0);
  }

  everywhere (S1)
    context1 = 0;
  context1 = 1;
  everywhere (S2)
    context2 = 0;
  context2 = 1;
  everywhere(S1)
    everywhere(S2)
      same_context = &= (context1 == context2);

  return(same_context);
}
```

Since shapes may be dynamically specified and/or modified, it may not be possible to determine at compile time if two shapes will be the same at run time; the compiler must check, however, that the types of the two shapes are compatible. No requirement is placed on the generated code to perform run-time tests that shapes are equivalent; the user must guarantee the equivalence of shapes at run time to avoid undefined behavior. See §A.10 for compiler options for shape equivalence testing.

The shape type of the result of an operation is the composite shape type of the operands. The shape of the result of an operation is the shape of the operands.

If one operand is parallel and one is nonparallel, the nonparallel operand is promoted to a parallel value of the other operand's shape by replicating the nonparallel operand's value.

If the operator is an assignment operator, this replication applies only when the left operand or destination is parallel and the right operand or source is nonparallel.

Other exceptions to these rules are noted for each operator in the sections that follow.

The context component of a shape is a specification of which positions of a parallel operand of the shape are active for a given operation. The context component of a shape is conceptually a parallel integral value of the shape, where a position is indicated as active by a nonzero in the corresponding context element, and as inactive by a zero in the corresponding context element. The elements of the context of a given shape are initially all nonzero, indicating that all positions are active.

The context of the shape associated with a parallel operation determines which positions of the operands participate in the operation. Active positions participate in the operation, and inactive ones do not.

The context of a given shape is altered by context-modifying statements and expressions which assign elements of the context. The **where** and **everywhere** statements modify the context component of a shape (see §3.6.7). Expressions involving the `&&`, `||`, and `:?` operators also modify the context component of a shape.

3.3 EXPRESSIONS [ISO §6.3, X3.159 §3.3]

Add for sliced expressions:

An expression that contains an operand that is a sliced expression (see §3.3.3.6) shall contain only operands that are sliced expressions or are of nonparallel type.

3.3.1 Primary Expressions [ISO §6.3.1, X3.159 §3.3.1]

Revise as indicated:

Syntax

primary-expression:
identifier
.
constant
string-literal
(*expression*)

Constraints

A '.' shall only be used in a *parallel-index-expression* (see §3.3.3.6).

Semantics

Add:

The type of the expression '.' is the parallel `int` equivalent to `pcoord(s,a)`, where `a` is the axis specifier (0–(n-1)) of the *parallel-index-expression* in which it is used, and `s` is the shape of the nearest enclosing parallel expression being indexed (see §3.3.3.6).

3.3.2 Postfix Operators [ISO §6.3.2, X3.159 §3.3.2]

3.3.2.1 Array subscripting [ISO §6.3.2.1, X3.159 §3.3.2.1]

Revise to allow arrays subscripted by nonparallel or by parallel ints:

Constraints

One of the following shall hold:

- One of the expressions shall have type "pointer to object *type*," the other expression shall have integral type, and the result has type "*type*."
- One of the expressions shall have type "pointer to object *type*," the other expression shall have parallel integral type of shape *S*, and the result has type "parallel operand of type *type* and shape *S*."

Semantics

A postfix expression `E1` followed by an expression `E2` in square brackets `[]` is either a subscripted designation of an element of an array object, or a parallel subscripted designation of elements of an array object.

If `E1` or `E2` is an integral expression, then the definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*(E1 + (E2)))`. ...

> Integral subscripts with pointers to parallel objects behave as do subscripts with pointers to nonparallel objects.

If `E1` or `E2` is a parallel integral expression, then `E1[E2]` is also identical to `(*(E1 + (E2)))`. In this case `(E1 + (E2))` denotes adding a parallel `int` to a pointer, and results in a parallel pointer. If the pointer expression is a pointer to nonparallel, the result is a parallel pointer to nonparallel, and if the pointer expression is a pointer to parallel, the result is a parallel pointer to parallel. The value at each position of the resulting parallel pointer is the sum of the pointer and the integral value at the corresponding position of the parallel integral expression, as described in §3.3.6. The behavior of dereferencing the resulting parallel pointer has position-oriented semantics as described in §3.3.3.2.

Examples

```
shape [20]S;
shape sarray[5];   /* Array of shapes */
int:S x;           /* Parallel int */
int:S y[50];       /* Array of parallel int */
int:S *z;          /* Pointer to parallel int */
int i;
int iarray[100];   /* Array of int */

sarray[i];         /* Designates the i-th member of sarray, which
                      is a shape */

z = &y[39];        /* z designates a subarray of y */

y[i];              /* Designates the i-th member of y, which is a
                      parallel int of shape S */

z[i];              /* Designates the i-th member of z, which is the
                      (39+i)-th member of y */

y[x];              /* Denotes an int:S, the elements of y
                      designated by the parallel int x */

iarray[x];         /* Denotes the elements of iarray designated by
                      the parallel int x */
```

3.3.2.2 Function calls [ISO §6.3.2.2, X3.159 §3.3.2.2]

Constraints

Add for elemental and nodal functions:

A value of type *T1* is *elementally assignable* to an object of type *T2* if *T2* is a nonparallel type and one of the following is true:

- a value of the type *T1* may be assigned to an object of the unqualified version of the type *T2*; or
- *T1* is a parallel type and a value of the element type of *T1* may be assigned to an object of the unqualified version of the type *T2*; or
- *T1* and *T2* are pointer types such that *T1* is either *pointer to T1'* or *parallel pointer to T1'*, *T2* is *pointer to T2'*, and a value of type *T1'* is elementally assignable to an object of type *T2'*; or
- *T1* and *T2* are array types with the same array size specifications and a value of the element type of *T1* is elementally assignable to an object of the element type of *T2*; or
- *T1* and *T2* are compatible elemental or nodal function types.

Add for elemental functions:

Each parallel argument to an elemental function shall be elementally assignable to an object of the type of its corresponding parameter.

Add for nodal functions:

Each parallel argument to a nodal function shall have a corresponding parameter such that one of the following holds (see §3.5.4.3):

- the parameter type is a parallel type of shape `void`, such that an element value of the argument is elementally assignable to an object of the type of an element of the parameter; or
- the parameter type is a pointer to nonparallel type, such that an element value of the argument is elementally assignable to an object of the type pointed to by the parameter; or
- the parameter type is a nonparallel type, such that an element value of the argument is elementally assignable to an object of the type of the parameter.

Each nonparallel argument to a nodal function shall have a type such that it can be promoted to the corresponding parallel type of **physical** shape.

An argument to a nodal function shall not be a pointer to a nonparallel type.

Semantics

Add for parallel parameters in prototypes, parallel arguments, parallel return types:

Both shapes and parallel operands may be passed to and returned from functions. Shape and parallel operand arguments are passed by value; creating the local copy of these arguments to a function can be inefficient. The usual rules for function calls with and without prototypes applies.

Parallel arguments and return values are evaluated under the context of the expression in which the function call occurs. Parallel operands passed as arguments behave as if assigned to the local copy, following the semantics of assignment under context (see §3.3.16). Only active positions are assigned, and the shape of the argument must be the same as the shape of the parameter. A parameter or return value declared as having **void** shape assumes the shape of the corresponding argument or return expression, respectively.

To allow access to all positions of a parallel object, use an **everywhere** statement around the call, or pass a pointer to the object and use an **everywhere** statement around accesses to the object within the function body.

Examples

```
shape [100]S;
int:S a, b;

int count_active_positionsof(int:void x)
{
      return(+= (int:(*shapeof(x))) 1);
}

void print_sum(int:S x)
{
      printf("Sum of parallel argument is %d\n", +=x);
}

int:S add_one(int:S x) {
      return(x + 1);
}

/* Examples of use */
print_sum(a);

b = add_one(a);

where (a > 0) {
    printf("Number of positive elements in a is %d\n",
           count_active_positionsof(a));
    everywhere (S) {
        printf("Total number of elements in a is %d\n",
               count_active_positionsof(a));
    }
}
```

Add for elemental functions:

A function qualified with the **elemental** qualifier is called an *elemental function*. The function must be so-qualified at both the function definition and the function call; if not, the behavior is undefined. See also §3.5.4.3, §3.6.6.4, and §3.7.1.

An elemental function can be executed either elementally or nonelementally. If none of the arguments are parallel operands, and the calling function is not executing elementally, the function is executed nonelementally (i.e., as a normal function). If one or more of the arguments are parallel operands, or the calling function is executing elementally, then the called function is executed elementally. All the parallel arguments must be of the same shape; otherwise, the behavior is undefined. Nonparallel arguments to a function being executed elementally are promoted to parallel in the usual manner.

When an elemental function is executing elementally, a shape is established at run time for the function. This shape is the shape of the parallel arguments. An instance of the code contained in an elemental function is executed for each active position of the established shape. An elemental function is indivisible with respect to synchronization. That is, an elemental function is treated as if it was a basic operator (like addition). All instances of the function execute as if in parallel; there are no assumptions about the synchronization of the intermediate steps, and there is a conceptual synchronization upon exit of the function.

When an elemental function is called from within an elemental function that is not executing elementally, the inner function call also executes as if it were a nonelemental function.

When an elemental function is called from within an elemental function that is executing elementally, the positions that execute the inner function call execute the body of the inner called function elementally.

An elemental function that is declared to return a type other than **void** may return either a parallel value or a nonparallel value. If the function is not executing elementally, or if the function is executing elementally and is returning to a function that is executing elementally, then a nonparallel value is returned. If the function is executing elementally and returning to a function that is not executing elementally, then a parallel value of the returning function's established shape is returned.

Examples

```
shape [10]S;
int:S x, y;

int f(int a, int b) elemental
{
   return(a+b);
}

int g(int a) elemental
{
   return(f(a,a));
}

f(x,y);      /* Returns parallel int of shape S with sum of
                active positions of x and y; inactive
                positions are unspecified. */

f(x,1);      /* Returns parallel int of shape S with sum of 1
                and active positions of x; inactive positions
                are unspecified. */

f(1,2);      /* Returns int with sum of 1 and 2. */

g(x);        /* Returns parallel int of shape S with values
                equal to two times the values of x in the
                active positions; inactive positions are
                unspecified. */

g(1);        /* Returns two times 1. */
```

Add for nodal functions:

A function qualified with the **nodal** qualifier is called a *nodal function*: The function must be so-qualified at both the function definition and the function call; if not, the behavior is undefined. See also §3.5.4.3, §3.6.6.4, and §3.7.1.

An invocation of a nodal function occurs as if the function is invoked once on each node of the execution environment in Single Program, Multiple Data (SPMD) style; that is, as if a separate thread is spawned on each node to execute the function body. Nodal functions therefore provide an escape to a multithreaded programming model. These threads, one per node, are only required by the execution model to synchronize upon return from the nodal function.

On each node the body of the nodal function executes in a temporarily established single-node environment called the *nodal execution environment*. That is, during the execution of a nodal function, a call to **positionsof(physical)** will return 1. The execution environment of the caller is reestablished upon return from the nodal function.

A nonparallel argument to a nodal function is first promoted to a parallel value of **physical** shape by replication. Then the argument is processed as if it was originally a parallel argument.

For a parallel argument to a nodal function, a thread of the nodal function will receive exactly those positions of the argument that are stored on the node executing the thread.

For a parallel argument to a nodal function, if the corresponding parameter is of parallel type, then for each thread executing the nodal function, a shape object will be created whose rank, dimensions, context and layout are derived from the shape of the argument. A pointer to this shape will be returned when the **shapeof** operator is applied to the parallel parameter during execution of the nodal function. The corresponding parameter will receive the values of its argument in those positions stored on the node executing the thread. The correspondence of positions in the shape of the argument and positions in the shape of the parameter is implementation-defined, but will be the same for all arguments of the same shape. The shape object created is derived as follows:

- The rank of the shape of the parameter is equal to the rank of the shape of the argument.
- The dimensions of the shape of the parameter are derived from the layout of the shape of the argument. For each dimension the number of positions is equal to the number of distinct parallel-index values in that dimension for the set of elements of the argument mapped to the node executing the thread.
- The context of the shape of the parameter will be initialized from the context of the shape of the argument: a position will be active in the parameter's shape if its corresponding position in the argument's shape is active; otherwise the position will be inactive.
- The layout of the shape of the parameter will reflect that the nodal function executes in a single-node environment. All positions will be mapped to the single node.

If there is more than one parallel argument of the same shape, all being passed to parameters of parallel type, then the same shape value shall be returned by dereferencing the return value of **shapeof** applied to any of these parameters; otherwise the behavior is undefined.

Note that parallel arguments are evaluated under context. If a reference is made to an element of a parameter that corresponds to an inactive position of its argument, the behavior is undefined.

For a parallel argument to a nodal function, if the corresponding parameter is of nonparallel pointer type, then for each thread executing the nodal function, an array object will be created that contains the values of the argument in those positions stored on the node executing the thread. Memory for the array will be allocated at the time of the call to the nodal function and will be freed upon return from the nodal function. The corresponding parameter will receive the created array object. The correspondence of positions in the shape of the argument and elements in the passed array is implementation-defined, but will be the same for all arguments of the same shape.

For a parallel argument to a nodal function, if the corresponding parameter is of nonparallel type and is not of pointer type, then the argument must be of **physical** shape and each thread of the nodal function will receive as the parameter the single value of the argument stored on the node executing the thread. If the argument is not of **physical** shape, then the behavior is undefined.

When the element type of a parallel argument to a nodal function is a pointer type, the pointer value at each position is converted so that the corresponding element of each thread's parameter receives a pointer to the object stored at the corresponding position of the parallel object pointed to by the argument's pointer value. That is, the argument is a parallel pointer to parallel and each thread's parameter is derived from the argument's value according to the position-oriented semantics described in §3.3.3.2.

If no positions of a parallel argument to a nodal function are stored on the node executing a thread of the nodal function, then the behavior is undefined.

Examples

```
void f1(int:void a, int b[], int *c, int d) nodal;
void f2(int:void *:void a, int *b[], int ** c) nodal;

shape [100]S;
int:S x;
int:physical y;
int i;
int:physical *:physical physical_Pp2Pi;
int:S *:S Pp2Pi;

call_one: f1(x, x, x, y);
      /* For each thread executing f1:  the first parameter
         receives a parallel of a new shape with number of
         positions equal to the number of positions of x
         stored on the node executing the thread; the second
         and third parameters receive arrays with lengths
         equal to the number of positions of x stored on the
         node; the fourth parameter receives a single value
         corresponding to the single position of physical
         stored on the node.  Note that if the shape of x
         has inactive positions, then there will be
         undefined values in the corresponding elements of
         the first three parameters.  That is, only the
         values of the active elements of x are assigned to
         the corresponding elements of the first three
         parameters.  Similarly, if the position of physical
         stored on the node is inactive, the value of the
         fourth parameter will be undefined; otherwise the
         value of y at that position is assigned to the
         fourth parameter. */

call_two: f1(y, y, y, y);
      /* For each thread executing f1:  the first parameter
         receives a parallel with a single element, which is
         the element of y stored on the node executing the
         thread, if that position is active; the second and
         third parameters each receive an array with a
         single element; the fourth parameter is as in
         call_one. */

f1(i, i, i, i);
      /* First, i is promoted to an int:physical for each
         argument.  Then each thread executing f1 gets the
         single value of i in each parameter, if the
         position of physical stored on the node executing
         the thread is active. */

f1(x, x, x, x);
      /* Undefined; x is not of shape physical */
```

```
f1(x, x, physical_Pp2Pi, y);
      /* The first two and the fourth parameters will be as
         in call_one.  For each thread executing f1, the
         third parameter will receive a single value derived
         from the element of the third argument stored on
         the node, if that element is active, and an
         undefined value otherwise.  When this parameter
         value is dereferenced inside f1, the object
         accessed will be the element stored on this node of
         the parallel int pointed to by the argument element
         from which the parameter was derived. */

f2(Pp2Pi, Pp2pi, Pp2Pi);
      /* For each thread executing f2:  each parameter will
         receive pointer values derived from the values of
         Pp2Pi stored on the node executing the thread
         (assuming all elements of Pp2Pi are active).  The
         value of each element of a parameter has position-
         oriented semantics and will reference objects only
         at its corresponding position. */
```

3.3.2.3 Structure and union members [ISO §6.3.2.3, X3.159 §3.3.2.3]

Constraints

Revise:

The first operand of the . operator shall have a qualified or unqualified, parallel or nonparallel structure or union type, and the second operand shall name a member of that structure or union type.

The first operand of the -> operator shall have type "pointer to qualified or unqualified, parallel or nonparallel structure" or "pointer to qualified or unqualified, parallel or nonparallel union," and the second operand shall name a member of the structure or union of the type pointed to.

Semantics

Add:

If the first operand of the . or -> operator is of parallel structure or union type, the result is a parallel value of the same type as the member designated by the second operand; the value at each position of the result is the designated member at the corresponding position of the first operand. If the first operand is an lvalue, the result is an lvalue.

Examples

```
shape [10]S;
struct Struct { int i; float f; };
struct Struct:S s;
struct Struct:S *p;

s.i;          /* Denotes a parallel int value whose elements are
                 the corresponding int members of the parallel
                 struct s */

p->f;         /* Denotes a parallel float value whose elements are
                 the corresponding float members of the parallel
                 struct pointed to by p */
```

3.3.2.4 Postfix increment and decrement operators [ISO §6.3.2.4, X3.159 §3.3.2.4]

Constraints

Revise:

The operand of the postfix increment or decrement operator shall have qualified or unqualified, parallel or nonparallel scalar type and shall be a modifiable lvalue.

Semantics

Add:

If the operand of the postfix increment or decrement operator is of parallel type, each position of the parallel operand is incremented or decremented, respectively.

Examples

```
shape [10]S;
int:S x;
int:S y[20];
int:S *p2Pi = &y[10];     /* Points to parallel int which is
                             the 10-th element of y */
int:S *:S Pp2Pi = y;      /* Every element of Pp2Pi points to
                             parallel int y[0] */

x++;                      /* Increments each element of x */

y[i]--;                   /* Decrements each element of y[i] */

p2Pi++;                   /* Increments p2Pi to point to the next
                             parallel int in the array y, namely
                             y[11] */

Pp2Pi++;                  /* Increments each element of Pp2Pi, which
                             would result in each element of Pp2Pi
                             pointing to the next parallel int in
                             the array y, namely y[1] */

[3]Pp2Pi = &y[5];         /* Assigns element 3 of Pp2Pi to point to
                             y[5] */

Pp2Pi++;                  /* Increments each element of Pp2Pi; now
                             all elements point to y[2], except
                             [3]Pp2Pi points to y[6] */
```

3.3.3 Unary Operators [ISO §6.3.3, X3.159 §3.3.3]

Revise as indicated:

Syntax

unary-expression:
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof **(** *type-name* **)**
shapeof **(** *parallel-variable-identifier* **)**
parallel-index postfix-expression
reduction-operator postfix-expression

parallel-variable-identifier:
identifier

parallel-index:
[*index-or-slice* **]** *parallel-index*
[*index-or-slice* **]**

index-or-slice:
index
slice

index:
expression

slice:
first **:** *last* **:** *stride*
first **:** *last*

first:
expression

last:
expression

stride:
expression

reduction-operator: one of

+=	**-=**	***=**	**/=**
&=	**^=**	**|=**	
<?=	**>?=**		

3.3.3.1 Prefix increment and decrement operators [ISO §6.3.3.1, X3.159 §3.3.3.1]

Constraints

Revise:

The operand of the prefix increment or decrement operator shall have qualified or unqualified, parallel or nonparallel scalar type and shall be a modifiable lvalue.

Semantics

Add:

If the operand of the prefix increment or decrement operator is of parallel type, each position of the parallel operand is incremented or decremented, respectively.

Examples

```
shape [10]S;
int:S x;
int:S y[20];              /* An array of parallel ints */
int:S *p2Pi = &y[5];      /* Pointer to the parallel int which is
                             the 5-th element of y */
int:S *:S Pp2Pi = &x;     /* Parallel pointer to parallel int where
                             every pointer points to the parallel
                             int x */

--x;                      /* Decrements every element of x */

--y[10];                  /* Decrements every element of y[10] */

--p2Pi;                   /* Decrements p2Pi to point to the
                             preceding parallel int, y[4] */

--(*p2Pi);                /* Decrements every element of the
                             parallel int y[4] */

--(*Pp2Pi)                /* Decrements the elements of the parallel
                             ints denoted by *Pp2Pi */
```

3.3.3.2 Address and indirection operators [ISO §6.3.3.2, X3.159 §3.3.3.2]

Constraints

Revise for pointer to parallel:

The operand of the unary `&` operator shall be either a function designator, or an lvalue that designates an object that is not a bit-field and is not declared with the `register` storage-class specifier.

Add:

The `&` operator shall not be applied to a parallel lvalue that results from parallel indexing or from array subscripting with a parallel subscript.

> Note: you cannot take the address of an element of a parallel operand, nor of a parallel lvalue produced by parallel indexing or by parallel subscripting.

Semantics

Add for pointer to parallel:

The application of `&` to a parallel lvalue produces a pointer to that lvalue, a *pointer to parallel*. If the operand has type "parallel *T* of shape *S*" the result has type "pointer to parallel *T* of shape *S*."

> A parallel pointer is not produced by application of `&` to a parallel object, although a parallel pointer value may be produced by replication of a pointer to parallel or during parallel subscripting of an array (see §3.3.2.1).
>
> Note that the application of `&` to a shape object produces a pointer to that shape.

Examples

```
shape [10]S;                    /* Shape S */
shape *sp;                      /* Pointer to shape */
float a, b;                     /* Nonparallel floats */
int i;                          /* Nonparallel int */
int:S x;                        /* Parallel int */
double:S y, z;                  /* Parallel double */
int:S *p2Pi;                    /* Pointer to parallel int */
float *:S Pp2f;                 /* Parallel pointer to float */
double:S *:S Pp2Pd;             /* Parallel pointer to parallel double */
int:S A[20];                    /* Array of parallel int */
double:S B[30];                 /* Array of parallel double */

sp = &S;                        /* Assigns sp to point to shape S */

p2Pi = &x;                      /* Assigns p2Pi to point to parallel int x
                                   */

p2Pi = A;                       /* Since A is an array, it decays to a
                                   pointer to parallel int.  p2Pi is
                                   assigned that pointer, which is the
                                   address of the 0-th element of A */

Pp2f = &a;                      /* Assigns each element of Pp2f to point
                                   to float a */

[3]Pp2f = &b;                   /* Assigns 3rd element of Pp2f to point to
                                   float b */

Pp2Pd = &y;                     /* Assigns each element of Pp2Pd to point
                                   to parallel double y */

[2]Pp2Pd = &z;                  /* Assigns 2nd element of Pp2Pd to point
                                   to parallel double z */

Pp2Pd = B+x;                    /* Assigns Pp2Pd to be the parallel
                                   pointer to double returned as the sum
                                   of B and the parallel int x */

&[3]x;                          /* Constraint violation */
&[x]y;                          /* Constraint violation */
&B[x];                          /* Constraint violation */
```

Add for pointer to parallel and parallel pointers:

The result of the * operator applied to a pointer to parallel type T of shape S is a parallel lvalue of parallel type T and of shape S.

The result of the * operator applied to a parallel pointer of shape S to nonparallel type T is a parallel value of type T and shape S. The value at each position of the resulting parallel T value is the result of dereferencing the pointer at the corresponding position of the parallel pointer.

The result of the * operator applied to a pointer P of type "parallel pointer of shape S to parallel type T also of shape S" is a parallel value of parallel type T and shape S. The value at each position of the resulting parallel T value is the result of position-oriented dereferencing of the parallel pointer, yielding the value at the corresponding position of the parallel T object referenced by the pointer at that position of P. That is, for each position i in shape S, the value at position i is:

```
[i](*[i]P)  /* Where P is the parallel pointer */
```

Add for pointer to shape:

The result of the * operator applied to a pointer to shape produces the shape.

Examples

```
float a,b;
shape [10]S;
shape *sptr = &S;          /* Initialized to point to S */
int:S x = 3;               /* Initializes all elements to 3 */
int:S *p2Pi = &x;          /* Initialized to point to
                              parallel int x */

double:S B[30];
float *:S Pp2f = &a;       /* Initializes all elements to point to
                              float a */

double:S *:S Pp2Pd;

[7]Pp2f = &b;              /* Assigns the 7th element of Pp2f to
                              point to float b */

                           /* Assigns 2 to the 5th element of x */
[5]x = 2;

Pp2Pd = B+x;               /* Assigns corresponding elements of Pp2Pd
                              to point to parallel double elements
                              of B as directed by parallel int x;
                              all elements get &B[3], except the
                              5th element of Pp2Pd will get &B[2] */

*sptr;                     /* Denotes the shape pointed to by sptr;
                              in this case S */

*p2Pi;                     /* Denotes the parallel int x */

*Pp2f;                     /* Denotes a parallel float of shape S; in
                              this case equal to a in all positions
                              except equal to b in position 7 */

*Pp2Pd;                    /* Denotes a parallel double of shape S;
                              in this case equal to the corresponding
                              element of B[3] in all positions except
                              equal to the corresponding element of
                              B[2] in position 5--i.e., the value
                              in position i will be [i]B[3], except
                              in position 5, the value will be
                              [5]B[2]. */
```

3.3.3.3 Unary arithmetic operators [ISO §6.3.3.3, X3.159 §3.3.3.3]

Constraints

Revise:

The operand of the unary + or - operator shall have parallel or nonparallel arithmetic type; of the ~ operator, parallel or nonparallel integral type; of the ! operator, parallel or nonparallel scalar type.

Semantics

Add:

If the operand of a unary arithmetic operator is of parallel type, the result is a value of the parallel type, where the value at each position of the result is determined by applying the operator to the operand's value at the corresponding position.

Examples

```
shape [20][20]S;
float:S f;

-f;          /* Denotes a parallel float value whose value at each
                position is the negation of the value at the
                corresponding position of f */
```

3.3.3.4 The `sizeof` operator [ISO §6.3.3.4, X3.159 §3.3.3.4]

Constraints

Revise:

The **`sizeof`** operator shall not be applied to an expression that has function type or an incomplete type, to a parallel type with incomplete element type, to the parenthesized name of such a type, or to an lvalue that designates a bit-field object.

Semantics

Add:

The result of the **`sizeof`** operator applied to a parallel type or a variable of parallel type is the size of the element type; it reflects the storage requirements in bytes for an element, including possible alignment constraints.

> Hence, it may not be the same as the result of **`sizeof`** applied to its nonparallel counterpart.

Examples

```
shape [10]S, [20]T;
int:S a;

/* The following expressions evaluate to 1 */
sizeof(int:S) == sizeof(int:T);
sizeof(a) == sizeof(int:S);

/* The following does not necessarily evaluate to 1 */
sizeof(int:S) == sizeof(int);
```

The result of the **`sizeof`** operator applied to an array object whose element type is parallel is the product of the array length and the **`sizeof`** operator applied to the element type.

Examples

```
shape [100]S;
int:S x[10];

/* The following expression evaluates to 1 */
sizeof(x) == (10 * sizeof(int:S));
```

The result of the **`sizeof`** operator applied to a shape variable or to a shape type is the number of bytes in a shape object.

Examples

```
shape *Sptr;
shape [10]S;

Sptr = (shape *) malloc(sizeof(shape));
                  /* Allocates new shape object */

sizeof(S);        /* Denotes the size of the shape object S */
```

3.3.3.5 The shapeof operator [NEW]

Constraints

The **shapeof** operator shall be applied only to an identifier of a previously declared variable of parallel type.

Semantics

The **shapeof** operator yields a pointer to the shape object associated with its operand.

The **shapeof** operator may be used wherever a pointer to a shape may be used.

Examples

```
shape [10]S;
int:S x;
shape *sp = shapeof(x);
int:(*sp) y;                    /* x and y are associated with the
                                   same shape object */
int:S *z;

/* The following evaluates to 1 */
dimof(*shapeof(x),0) == dimof(S,0) == 1;

/* Allocates an array of 20 parallel ints */
z = (int:S *)palloc(sp,20*sizeof(int:S));
```

3.3.3.6 Parallel indexing [NEW]

Constraints

The *postfix-expression* shall be of parallel type.

The *parallel-index* shall consist of *n index-or-slice* expressions where *n* is the rank of the shape of the nearest enclosing parallel expression being indexed. Each *index* expression shall be of integral or parallel integral type. Each subexpression of a *slice* expression shall be of integral type.

All *index* expressions of parallel integral type in the same *parallel-index* shall be of compatible shape type.

If any *index-or-slice* expression in a *parallel-index* is a *slice* expression, all *index* expressions in the same *parallel-index* shall be of integral type, and the *postfix-expression* shall denote a parallel lvalue.

If the *postfix-expression* is being sliced, it shall not be a sliced expression.

An expression may be sliced only once; nested slicing is not allowed.

If the *postfix-expression* is being parallel indexed and is itself a parallel-indexed expression, it must be enclosed in parentheses.

This allows the compiler to determine the rank of each of the *parallel-index* expressions. Note that in order to parallel index an expression more than once, the intermediate results must be parallel, which occurs when one or more index expressions is of parallel type. See the examples below.

Semantics

A *postfix-expression*, **E2**, preceded by a *parallel-index*, **E1**, which consists of a sequence of one or more *index* expressions enclosed in square brackets **[]**, is a *parallel-indexed expression*. The shape of **E2** is herein denoted by **S2**, and the shape of each of the *index* expressions of parallel integral type in **E1**, if any exist, is herein denoted by **S1**.

Note that such an expression is called a parallel-indexed expression because the operand being indexed is parallel, regardless of whether any *index* expression is of parallel integral type.

A *postfix-expression*, **E2**, preceded by a *parallel-index*, **E1**, which contains one or more *slice* expressions, is a *sliced expression*. The shape of **E2** is herein denoted by **S2**.

Parallel-indexed expressions with a nonparallel index:

If all the *index* expressions of **E1** are of integral type, a parallel-indexed expression designates an individual position of the parallel object being indexed. The type of the parallel-indexed expression is the element type of **E2**. The result designates a selected element of **E2**; if **E2** is an lvalue, the result is an lvalue.

Note that parallel indexing by a nonparallel index always selects the designated position of the expression being indexed, independent of the context of the shape of that expression.

If the *index* expression designates a nonexistent position of the expression being indexed, the behavior is undefined.

Examples

```
shape [10]S, [5][5]T;
int:S a;
float:T b;
int i,j;

[8]a;              /* Selects the int at the 8-th position of a */

[i][j]b;           /* Selects the float at position (i,j) of b */

[100]a;            /* Undefined behavior since index is 100 */

[5]b;              /* Constraint violation - improper indexing for
                      rank 2 object */
```

Parallel-indexed expressions with a parallel index:

In a *parallel-index*, *index* expressions of parallel integral type need not be the same shape as the parallel expression being indexed.

> This is an exception to the general rule that parallel operands to binary operations must have the same shape.

If more than one of the *index* expressions in **E1** is of parallel type, all the parallel *index* expressions must be of the same shape; otherwise, the behavior is undefined.

If one or more of the *index* expressions in **E1** is of parallel type, each nonparallel integral *index* expression is first promoted to a parallel **int** of the same shape as the parallel *index* expressions by replicating the integral expression value. The resulting *parallel-index* **E1** designates a mapping from the shape **S2** of the *postfix-expression* **E2** to the shape **S1** of **E1**. The type of the result is a parallel object of shape **S1** whose element type is the element type of **E2**. The result is a parallel value designating elements of **E2** whose selection and order is as directed by the *index* expressions **E1**; if **E2** is an lvalue, the result is an lvalue.

When a *parallel-index* is of parallel type, the mapping behaves according to the following:

```
/* When used as a modifiable lvalue, updated by E3 */
for (i1=0; i1<dimof(S1,0); i1++)
  ...
    for (im=0; im<dimof(S1,m-1); im++)
      [[i1]...[im]index1]...[[i1]...[im]indexn] E2 =
        [i1]...[im]E3;

/* When used as an rvalue, as if temporary T3 created */
Type:S1 T3;
for (i1=0; i1<dimof(S1,0); i1++)
  ...
    for (im=0; im<dimof(S1,m-1); im++)
      [i1]...[im]T3 =
        [[i1]...[im]index1]...[[i1]...[im]indexn] E2;
```

where **S1** is of rank m , **S2** is of rank n, index_j indicates the j-th parallel index value of **E1**, and Type is the element type of **E2**. That is:

- when used where a modifiable lvalue is expected (i.e., on the left side of an assignment), the mapping selects elements of the object designated by **E2** to be updated. It may select the same element to be updated more than once, and some elements may not be selected to be updated at all. The result for an element selected more than once is unspecified; it will be chosen in an implementation-defined manner from one of the values of the elements selected from **E3** to be mapped to that position of **E2**.

- when used as an rvalue, the mapping produces a temporary parallel value of shape **S1** whose elements are selected from **E2** according to the *index* expressions, where the same element of **E2** may be selected multiple times, and not all elements of **E2** need be selected.

Examples

```
shape [4]S, [4][8]T;
int:S index1, index2;
float:T a;
float:S b;
int i;

index1 = 3 - pcoord(S,0);
                          /* Assigns index1 to be {3,2,1,0} */

index2 = 2;               /* Assigns 2 to all elements of
                             index2; {2,2,2,2} */

[index2]index1;           /* Denotes a parallel int of shape
                             S whose values at each position
                             are the value of [2]index1;
                             {1,1,1,1} */

[3]([index2]index1);      /* Denotes element 3 of the parallel
                             int denoted by [index2]index1 */

[2][index2]index1;        /* Constraint violation: improperly
                             parenthesized */

b = [index1][index2]a;    /* Assigns some elements of a to b as
                             directed by the parallel ints index1
                             and index2;  equivalent to the loop
                             assigning b below; b gets
                             {[3][2]a,[2][2]a,[1][2]a,[0][2]a} */
for (i=0; i<4; i++)
  [i]b = [[i]index1][[i]index2]a;

[index1][index2]a = b;    /* Assigns the elements of b into a as
                             directed by the parallel ints index1
                             and index2; equivalent to the loop
                             assigning a below */
for (i=0; i<4; i++)
  [[i]index1][[i]index2]a = [i]b;
```

The context of **S1** also affects which positions are selected in a parallel-indexed expression. There is an implicit contextualization of the resulting parallel **int** denoted by **E1**.

If a value in the parallel **int** denoted by **E1** selects a nonexistent position of **E2**, the behavior is undefined.

Examples

```
shape [5]S;
int:S a, b, c;

a = pcoord(S,0) + 1;     /* a gets {1, 2, 3, 4, 5} */

[2]a = 10;               /* a is now {1, 2, 10, 4, 5} */

b = 1;                   /* b gets {1, 1, 1, 1, 1} */

where (a<5) {            /* Updates elements 0,1, and 3 of b */
   [a]b = 0;             /* to be 0; elements 2 and 4 are not */
}                        /* updated; b is now {0, 0, 1, 0, 1} */

a = pcoord(S,0) + 5;     /* a gets {5, 6, 7, 8, 9} */

b = pcoord(S,0) %% 3;    /* b gets {0, 1, 2, 0, 1} */

c = pcoord(S,0);         /* c gets {0, 1, 2, 3, 4} */

[b]a = c;                /* Updates elements 0, 1, and 2 of a;
                            element 0 gets either 0 or 3;
                            element 1 gets either 1 or 4  */

a = pcoord(S,0) + 5;     /* a gets {5, 6, 7, 8, 9} */

c = [b]a;                /* Updates all elements of c;  elements 0
                            and 3 of c get [0]a; elements 1 and 4
                            of c get [1]a; element 2 of c gets
                            [2]a; c is now {5, 6, 7, 5, 6} */

[a]b;                    /* Undefined; tries to select nonexistent
                            positions of b */
```

Sliced expressions:

Each component *slice* expression or *index* expression in the *parallel-index* of a sliced expression denotes a sequence of positions along the axis of shape S2 that corresponds to the placement of the component within the sequence of components comprising the *parallel-index*.

The sequence denoted by a *slice* expression is as follows:

- The *first* expression denotes the first position in the sequence.
- The *last* expression denotes a limit (ceiling or floor) for the last position in the sequence.
- The *stride* expression denotes a stride or increment between successive positions in the sequence.

If the *stride* expression is omitted, a stride of 1 is assumed.

If the stride is 0, the sequence denoted is empty.

If the stride is positive, the sequence consists of the regularly spaced sequence of positions beginning with that denoted by the *first* expression and proceeding in increments of the stride up to the largest position not greater than that denoted by the *last* expression.

If the stride is negative, the sequence begins with the position denoted by the *first* expression and proceeds in increments of the stride down to the smallest position equal to or greater than that denoted by the *last* expression.

The sequence denoted by an *index* expression consists of the single position denoted by the expression.

Examples

```
shape [100][100]S;
shape [10][10][10]T;
int:S x;
float:T y;

[1][0:99:10]x;     /* Denotes the sequence {1} in the first axis,
                      and {0,10,20,30,40,50,60,70,80,90} in the
                      second axis; the shape of the result is
                      "shape [1][10]". */

[9:5:-1][3:7:2]x; /* Denotes the sequence {9,8,7,6,5} in the first
                      axis, and {3,5,7} in the second axis; the
                      shape of the result is "shape [5][3]". */

[1:10:2][5][10:5:-1]y;
                  /* Denotes the sequence {1,3,5,7,9} in the first
                      axis, {5} in the second axis, and
                      {10,9,8,7,6,5} in the third axis.  Since 10
                      is a nonexistent position in the third axis
                      of T, the behavior is undefined. */
```

If the sequence denoted by any component of a *parallel-index* is empty or contains nonexistent positions for the corresponding axis in shape **S2**, the behavior is undefined.

Otherwise, these sequences designate a mapping from the shape **S2** of **E2** to a conceptual shape **S3**, which has rank and dimensions as described below, an unspecified layout, and context in which all positions are active.

> No shape object is created for the shape **S3**; the concept of the shape **S3** is used only for expository purposes in describing the semantics of sliced expressions. A sliced expression denotes a subobject of a parallel object and may only be used in expressions involving nonparallel operands and other sliced expressions of shape **S3** (see §3.3).

The shape **S3** has rank equal to that of **S2**, but the dimension along each axis is equal to the number of positions in the sequence denoted by the corresponding *slice* expression for that axis in the *parallel-index*.

The type of the result is a parallel type of shape **S3** whose element type is the element type of **E2**. The result is a parallel value designating elements of **E2** whose selection and order is as directed by the sequences of positions denoted by the components of **E1**; if **E2** is an lvalue, the result is an lvalue.

The mapping designated by a sliced expression behaves according to the following:

```
/* When used as a modifiable lvalue, updated by E3 */
for (i1=0; i1<dimof(S3,0); i1++)
  ...
    for (ir=0; ir<dimof(S3,r-1); ir++)
      [seql_i1]...[seqr_ir]E2 = [i1]...[ir]E3;

/* When used as an rvalue, as if temporary T3 created */
Type:S3 T3;
for (i1=0; i1<dimof(S3,0); i1++)
  ...
    for (ir=0; ir<dimof(S3,r-1); ir++)
      [i1]...[ir]T3 = [seql_i1]...[seqr_ir]E2;
```

where *r* is the **rankof(S3)**, Type is the element type of **E2**, and $seqk_j$ is the j-th element of the k-th sequence of positions denoted by the *parallel-index*, corresponding to axis (k-1).

Note that element selection from **E2** is independent of the context of **S2**; the elements at the positions designated by the mapping are always selected.

If the result of the sliced expression **E1 E2** is used in an expression in which another parallel operand is the result of a sliced expression, but does not have shape equivalent to **S3** as described, the behavior is undefined.

Examples

```
shape [10][10]S;
shape [1024]T;
shape [512]U;
int:S x, y;
float:T z;
float:U w;
float result;

[0:9:2][0:9:2]x = 5;      /* Assigns elements at even positions of x
                             to be 5 */

[1:9:2][1:9:2]x = 10;     /* Assigns elements at odd positions of x
                             to be 10 */

[0:4:1][0:4:1]x = [5:9:1][5:9:1]x;
                          /* Assigns upper corner of x from the
                             lower corner of x */

[9:0:-1][0:9:1]x = [0:9:1][0:9:1]y;
                          /* Assigns x to be the transpose of y */

[9:0:-1][0:9:1]x = [0:9:1][0:9:1]x;
                          /* Assigns x to be its own transpose; note
                             that all reads from x will be done
                             before writes to x */

[0][0:9:1]x = [1][0:9:1]x + [2][0:9:1]x;
                          /* Assigns first row of x to be the sum of
                             the second and third rows of x */

[9:0:-1][0:9:1]x = x;     /* Undefined behavior; all operands must
                             be nonparallel or sliced expressions of
                             equivalent shape */

[0:1023:1]z += 1;         /* Add one to every element of z */

result = += [0:511]z;     /* Finds sum of first 512 elements of z */

result = (+= [0:511:1]z) - (+= [512:1023:1]z);
                          /* Finds difference of sums of first and
                             second halves of z */

w = [0:511:1]z + w;       /* Undefined behavior; all operands must
                             be nonparallel or sliced expressions of
                             equivalent shape */
```

3.3.3.7 Unary reduction operators [NEW]

Constraints

The operands of unary +=, -=, *=, and /= shall be of parallel arithmetic type.

The operands of unary &=, ^=, and |= shall be of parallel integral type.

The operands of unary <?= and >?= shall be of parallel arithmetic or parallel pointer type.

Semantics

These operators perform a reduction: the specified operation is performed on the operand values, resulting in a nonparallel value.

The result of the += operator is the sum of all the elements of the operand. If no positions of the operand are active, the result is 0, cast to the element type of the operand.

The result of the -= operator is the negation of the sum of all the elements of the operand. If no positions of the operand are active, the result is 0, cast to the element type of the operand.

The result of the *= operator is the product of all the elements of the operand. If no positions of the operand are active, the result is 1, cast to the element type of the operand.

The result of the /= operator is the reciprocal of the product of all the elements of the operand. If no positions of the operand are active, the result is 1, cast to the element type of the operand.

The result of the &=, ^=, and |= operators are the bitwise AND, XOR, and OR, respectively, of all the elements of the operand. If no positions of the operand are active, the result of &= is ~0, and the result of ^= and |= is 0.

The result of the <?= and >?= operators is the minimum and maximum, respectively, of all the elements of the operand. If no positions of the operand are active, the result is the minimum or maximum value for the element type, respectively, as defined in **<limits.h>** for integral types and **<float.h>** for floating types.

Examples

```
shape [1000]S;
int:S x;
int result;

result = +=x;          /* Assigns result to be the sum of all the
                          elements of x */

result = <?=x;         /* Assigns result to be the minimum valued
                          element of x */
```

3.3.4 Cast Operators [ISO §6.3.4, X3.159 §3.3.4]

Constraints

Revise:

Unless the *type-name* specifies void type, the *type-name* shall specify qualified or unqualified, parallel or nonparallel, scalar type and the operand shall have parallel or nonparallel scalar type.

A pointer to parallel type shall not be cast to a pointer to nonparallel type. A pointer to nonparallel type shall not be cast to a pointer to parallel type.

A parallel pointer shall not be cast to a nonparallel pointer.

Semantics

Add:

If the *type-name* specifies a parallel type and the operand is also of parallel type, then if the shape of the parallel type and the shape of the operand are the same, the result is a parallel value where the value at each position is the result of the conversion (if any) represented by a cast to the nonparallel type counterpart of the *type-name*; if the shape of the type and the shape of the operand are not the same, the result is undefined.

Parallel to parallel casts should not imply data movement or communication.

Examples

```
shape [20]S;
shape [10][2]T;
int:S a;

(float:S) a;              /* Denotes the parallel float whose
                             elements are the converted int values
                             of corresponding positions of a */

(int:physical) a;         /* Undefined behavior */
(int:T) a;                /* Undefined behavior */
```

If the *type-name* specifies a parallel type and the operand is of nonparallel type, the result is a parallel value of the same shape as the parallel type that has the operand value replicated and converted as indicated at each position.

Examples

```
shape [200]S;

(int:S) 1.0;              /* Denotes a parallel int of shape S
                             whose elements all have value 1 */
```

If the *type-name* specifies a nonparallel type and the operand is of parallel type, the result is to select an element of the operand and apply the indicated conversion to the selected element. The method of selecting the operand element is implementation-defined.

Examples

```
shape [200]S;
int:S a;

(double) a;                    /* Selects one of the 200 elements of a,
                                  and converts it to double */
```

3.3.5 Multiplicative Operators [ISO §6.3.5, X3.159 §3.3.5]

Syntax

Revise:

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*
multiplicative-expression %% *cast-expression*

Constraints

Revise:

Each of the operands shall have arithmetic or parallel arithmetic type. The operands of the % and %% operators shall have integral or parallel integral type.

Semantics

Add:

The result of the %% (modulus) operator is the remainder on division of the first operand by the second, but unlike the % operator, the result has the same sign as the first operand. The modulus operator evaluates the following formula to compute "a %% b":

```
a - (b * floor(a / b))
```

Examples

```
shape [10][20]S;
float:S f;
int:S i;

f * 2;              /* Denotes a parallel float value whose value at
                       each position is 2 times the value at the
                       corresponding position of f .  Note that the
                       nonparallel operand 2 will be promoted to a
                       parallel int value of shape S whose value
                       at each position will be 2.  The usual
                       arithmetic conversions will be applied to
                       this parallel int value to convert it to a
                       parallel float value. */

i %% 2;             /* Denotes a parallel int value of shape S
                       whose value at each position is the true
                       modulus resulting from dividing the values at
                       the corresponding positions of i by 2. */
```

3.3.6 Additive Operators [ISO §6.3.6, X3.159 §3.3.6]

Constraints

Revise to include pointer to parallel types and parallel pointer types:

For addition, one of the following shall hold:

- both operands shall have arithmetic or parallel arithmetic type;
- one operand shall be a pointer to an object type, and the other shall have integral or parallel integral type; or
- one operand shall be a parallel pointer to an object type, and the other shall have integral or parallel integral type.

For subtraction, one of the following shall hold:

- both operands have arithmetic or parallel arithmetic type;
- both operands are pointers to qualified or unqualified versions of compatible object types;
- the left operand is a pointer to an object type, and the right operand has integral type or parallel integral type; or
- the left operand is a parallel pointer to an object type, and the right operand has integral or parallel integral type.

Semantics

Add for pointer to parallel types and parallel pointer types:

Adding an integral value to, or subtracting an integral value from, a pointer to parallel type results in a pointer to a subsequent or preceding element of an array whose element type is the parallel type. Subtracting two pointers to parallel, when both point to elements of the same array object, results in the difference of the subscripts of the two array elements. Subtracting two pointers to parallel which do not point to elements of (or one past the last element of) the same array object will result in undefined behavior.

> Pointer arithmetic with pointers to parallel objects behaves just as does pointer arithmetic with pointers to nonparallel objects.

Adding a parallel integral value to, or subtracting a parallel integral value from, a parallel pointer results in a parallel pointer of the same type and shape whose value at each position is the sum or difference of the pointer value at the corresponding position of the parallel pointer and the integral value at the corresponding position of the parallel integral value. As usual, both operands must be of the same shape or the behavior is undefined.

If a parallel integral value is added to or subtracted from a nonparallel pointer value, or if a nonparallel integral value is added to or subtracted from a parallel pointer value, the nonparallel operand is first promoted to parallel, and then the addition or subtraction proceeds as described above.

Subtracting two parallel pointers results in a parallel `int` whose value at each position is the difference of the subscripts of two array elements, provided the pointer values at the corresponding positions of the operands point to elements of the same array object; otherwise the value at this position is undefined.

Examples

```
shape [10]S, [20]T;
float:S f;
double:S g;
int:S h[100];
int:S *p2Pi;
int:S index;
int:T index2;
int:S *:S Pp2Pi;
int:S *:S Pp2Pi_2;

f - g;                  /* Denotes a parallel double whose
                           value at each position is the
                           difference of the values at the
                           corresponding positions of f and g. */

p2Pi = &h[5];           /* Assigns p2Pi to point to 5th element of
                           array of parallel ints h */

Pp2Pi = &h[50];         /* Assigns every element of Pp2Pi to point
                           to the 50th element of h */

Pp2Pi_2 = &h[100];      /* All point to h[100] */

[5]Pp2Pi_2 = &index;    /* The 5th element points to index */

p2Pi + 5;               /* Denotes a pointer to the 10th element
                           of h */

Pp2Pi - 5;              /* Denotes a parallel pointer where every
                           element points to the 45th element of h
                           */

p2Pi - index;           /* Denotes a parallel pointer whose value
                           at each position is the difference of
                           &h[5] and the int at the corresponding
                           position of index */

Pp2Pi + index;          /* Denotes a parallel pointer whose value
                           at each position is the sum of the
                           pointer value and the int value in the
                           corresponding positions of Pp2Pi and
                           index, respectively */

&h[0] - p2Pi;           /* Denotes the difference of subscripts;
                           in this case, the integral value -5 */

Pp2Pi_2 - Pp2Pi;        /* Denotes a parallel int whose value at
                           each position is the difference of the
                           subscripts; in this case the difference
                           is 50 in every position, except in
                           position 5, where the value is
                           undefined */

Pp2Pi_2 + index2;       /* Undefined, shapes are not the same */

p2Pi - &index;          /* Undefined, not same array object */
```

3.3.7 Bitwise Shift Operators [ISO §6.3.7, X3.159 §3.3.7]

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

```
shape [100]S;
int:S  bits;

bits >> 2;          /* Denotes a parallel int value of shape S,
                       whose value at each position is the value at
                       the corresponding position of bits shifted
                       right by 2. */
```

3.3.8 Relational Operators [ISO §6.3.8, X3.159 §3.3.8]

Syntax

Revise as indicated:

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*
relational-expression <? *shift-expression*
relational-expression >? *shift-expression*

Constraints

Revise:

One of the following shall hold:

- both operands have arithmetic or parallel arithmetic type;
- both operands are pointers or parallel pointers to qualified or unqualified versions of compatible object types; or
- both operands are pointers or parallel pointers to qualified or unqualified versions of compatible incomplete types.

Semantics

If one operand is of parallel type and the other operand is of nonparallel type, the operand of nonparallel type is first promoted to a parallel type of the same shape as the parallel operand, as described in §3.2.3.

The <? operator results in the minimum value of its operands.

The >? operator results in the maximum value of its operands.

Operands which are pointers to parallel observe the same semantics as pointers to nonparallel.

Operands which are parallel pointers observe the same semantics elementally as ordinary pointers, producing a parallel result whose value at each position is the result of applying the operator to the values at the corresponding positions of the operands.

Examples

```
shape [100]S;
int:S x, y;
int:S z[100];
int:S *ptr1, *ptr2;
int:S *:S pptr1, *:S pptr2;

x > 0;              /* Denotes a parallel int value of shape S
                       whose value at each position is the result
                       (0 or 1) of the > operator applied to the
                       value at the corresponding position of
                       x and 0. */

x <? y;             /* Denotes a parallel int value of shape S
                       whose value at each position is the minimum
                       of the values at the corresponding positions
                       of x and y. */

ptr1 > ptr2;        /* Denotes an int value, which is 1 if ptr1
                       points to an element of the same array as
                       ptr2 but with a higher subscript, and 0 or
                       undefined otherwise. */

pptr1 > pptr2;      /* Denotes a parallel int value of shape S
                       whose value at each position is the result of
                       comparing the pointer values at the
                       corresponding positions of the operands. */
```

3.3.9 Equality Operators [ISO §6.3.9, X3.159 §3.3.9]

Constraints

Revise:

One of the following shall hold:

- both operands have arithmetic or parallel arithmetic type;
- both operands are pointers or parallel pointers to qualified or unqualified, parallel or nonparallel, versions of compatible types;
- one operand is a pointer or a parallel pointer to an object or an incomplete type, and the other is a pointer or a parallel pointer to a qualified or unqualified version of **void**;
- one operand is a pointer or a parallel pointer to nonparallel type and the other is a null pointer constant; or
- one operand is a pointer to parallel type and the other is a null pointer-to-parallel constant (see §3.2.2.3).

Examples

```
shape [100][100]S, [100][100]T;
shape [100] U, V;
shape *ps1, *ps2;
float:S x, y;
float:S *ppf = &x;

x == y;              /* Denotes a parallel int value of shape S
                        whose value at each position is the result
                        (0 or 1) of the == operator applied to the
                        values at the corresponding positions of x
                        and y. */

/* Assign the following values */
V = T;
ps1 = &S;
ps2 = &T;
x = 1.0;
[0][0]x = 0.0;

/* The following all evaluate to 1 */
S == T;
S != U;
S == V;
where (x > 0) {
   S != T;
}
*shapeof(x) == *shapeof(y);
ps1 != ps2;
*ps1 == *ps2;
ppf == &x;
ppf != &y;
```

3.3.10 Bitwise AND Operator [ISO §6.3.10, X3.159 §3.3.10]

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

```
shape [1000]S;
int:S x;

x & 0x01;          /* Denotes a parallel int value of shape S
                      whose value at each position is the result of
                      the & operator applied to the values at the
                      corresponding positions of x and 0x01.
                      Hence, the value at each position is 1 if the
                      least significant bit of x at that position
                      is 1, and 0 otherwise. */
```

3.3.11 Bitwise Exclusive OR Operator [ISO §6.3.11, X3.159 §3.3.11]

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

```
shape [20][20]S;
int:S x, y;

x ^ y;             /* Denotes a parallel int value of shape S
                      whose value at each position is the result of
                      the ^ operator applied to the values at the
                      corresponding positions of x and y. */
```

3.3.12 Bitwise Inclusive OR Operator [ISO §6.3.12, X3.159 §3.3.12]

Constraints

Revise:

Each of the operands shall have integral or parallel integral type.

Examples

```
shape [20][20]S;
int:S x, y;

x | y;              /* Denotes a parallel int of shape S  whose
                      value at each position is the result of
                      the | operator applied to the values at the
                      corresponding positions of x and y. */
```

3.3.13 Logical AND Operator [ISO §6.3.13, X3.159 §3.3.13]

Constraints

Revise:

Each of the operands shall have parallel or nonparallel scalar type.

Semantics

Add:

If one of the operands is parallel, after the normal promotions have been performed (see §3.2.3), the first operand (LHS) is evaluated to determine a narrowed context for evaluating the second operand (RHS), and the second operand is evaluated only in the positions indicated by this context. That is, the active positions in which the LHS evaluates to zero are made inactive for evaluating the RHS. The result is a parallel `int` which is 1 in those positions for which the values in the corresponding active positions of both the LHS and RHS are nonzero, and 0 in all other active positions.

Examples

```
shape [10][10][100]S;
int:S x, y;

x && y;             /* Denotes a parallel int of shape S whose
                       value at each position is the result (0 or 1)
                       of applying the && operator to the values
                       at the corresponding positions of x and y.
                       Note that the second operand is only
                       evaluated in those active positions for which
                       the first operand is nonzero. */
```

3.3.14 Logical OR Operator [ISO §6.3.14, X3.159 §3.3.14]

Constraints

Revise:

Each of the operands shall have parallel or nonparallel scalar type.

Semantics

Add:

If one of the operands is parallel, after the normal promotions have been performed (see §3.2.3), the first operand (LHS) is evaluated to determine a narrowed context for evaluating the second operand (RHS), and the second operand is evaluated only in the positions indicated by this context. That is, the active positions in which the LHS evaluates to one are made inactive for evaluating the RHS. The result is a parallel value which is 0 in those active positions for which the values in the corresponding positions of both the LHS and RHS are 0, and 1 in all other active positions.

Examples

```
shape [10][10][100]S;
int:S x, y;

x || y;              /* Denotes a parallel int of shape S whose
                        value at each position is the result (0 or 1)
                        of applying the || operator to the values at
                        the corresponding positions of x and y.
                        Note that the second operand is only
                        evaluated in those active positions for which
                        the first operand is zero. */
```

3.3.15 Conditional Operator [ISO §6.3.15, X3.159 §3.3.15]

Constraints

Revise:

The first operand shall have nonparallel scalar type or parallel type with scalar element type.

If both of the second and third operands are of parallel type, they shall be of compatible parallel types.

One of the following shall hold for the second and third operands or, if the second or third operands are of parallel type, the following shall hold for the element types of the second and third operands:

...

Semantics

Add:

If the first operand has parallel type, the context of the shape of the first operand is constrained by the expression denoting the first operand for the purpose of evaluating the second and third operands. That is, the behavior is as if the second and third operands are evaluated in the scope of a **where** statement whose mask expression is the first operand. For example,

```
opnd1 ? opnd2 : opnd3
```

behaves as if evaluated with the same contextualization of the following:

```
where ( opnd1 )
    opnd2;
else
    opnd3;
```

Note that both opnd2 and opnd3 are evaluated in this case, whereas if opnd1 is nonparallel, only one of opnd2 and opnd3 is evaluated.

Examples

```
shape [100]S;
int:S x;
int y, z;

x < 0 ? -x : x          /* Denotes a parallel int whose value at
                           each position is the absolute value of
                           the corresponding value of x. Note that
                           both the true and false parallel
                           operands will be evaluated at each
                           position. */

x < y ? x : z           /* Denotes a parallel int whose value at
                           each position is the value at the
                           corresponding position of x, if that
                           value is less than y, and otherwise it
                           is the value of z. */
```

3.3.16 Assignment Operators [ISO §6.3.1, X3.159 §3.3.16]

Syntax

Revise as indicated:

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of

```
=     *=    /=    %=    %%=   +=    -=
<<=   >>=   &=    ^=    |=    <?=   >?=
```

Constraints

Add:

The assignment operators =, %=, %%=, <<=, and >>= shall not be used with a left operand that is nonparallel and a right operand that is parallel.

3.3.16.1 Simple assignment [ISO §6.3.16.1, X3.159 §3.3.16.1]

Constraints

Revise to allow assignment to shapes and parallel objects:

One of the following shall hold:

...

- the left operand is of parallel type *T1* and the right operand is of nonparallel type *T2*, such that an operand having the element type of *T1* can be the left operand of simple assignment where the right operand is of type *T2*;
- the left operand is of parallel type *T1* and the right operand is of parallel type *T2* such that *T1* and *T2* have compatible shapes and such that an operand having the element type of *T1* can be the left operand of simple assignment where the right operand has the element type of *T2*;
- the left operand is the identifier of a shape variable which is fully unspecified or partially specified when declared, and the right operand has compatible shape type;

...

A shape variable shall not be assigned a new shape value after it has been used in declaring objects of parallel type.

Semantics

Add:

If the left operand is of parallel type and the right operand is of nonparallel type, the value of the right operand is replicated to form a parallel value before assignment, including any necessary type conversions.

If the left operand is of parallel type and the right operand is of parallel type, the object denoted by the left operand is assigned elementally by the right operand, including any necessary type conversions.

If the left operand is a parallel type of shape `void`, which can only happen if the left operand is a parameter in a function prototype specification, the left operand assumes the shape of the right operand.

If the left operand is the identifier of a shape variable, and the right operand denotes a shape object of compatible type, the left operand is replaced by the value denoted by the right operand. The type of the left operand becomes the composite type of the two operands, and the value of the shape object becomes fully unassigned, partially assigned, or fully assigned, according to the type of the right operand. If the right operand is not shape compatible with the left operand, the behavior is undefined.

Note that assignment of shapes involves copying of the shape value.

Examples

```
shape [10]S, []T, [][]U, V;
int:S x;
float:S y;
typedef struct { int a, b; float f; } stype;
stype z = { 5, 8, 2.5 };
stype:S w;

x = 10;              /* 10 is first converted to a parallel int */

w = z;               /* Each element of w is assigned a copy of z */

z = w;               /* Constraint violation */

y = x;               /* Assigns elements of y from corresponding
                        elements of x, converting ints to floats */

V = S;               /* Composite type is type of S */
V = T;               /* Composite type is type of T */
V = U;               /* Composite type is type of U */
T = S;               /* Composite type is type of S */

U = S;               /* Constraint violation, incompatible shapes */
T = U;               /* Constraint violation, incompatible shapes */
S = T;               /* Constraint violation, can't assign a shape
                        variable that is fully specified when
                        declared */
```

3.3.16.2 Compound assignment [ISO §6.3.16.2, X3.159 §3.3.16.2]

Constraints

Revise:

For the operators += and -= only, either the left operand shall be a pointer or a parallel pointer to an object type, and the right shall have integral or parallel integral type; or the left operand shall have qualified or unqualified, parallel or nonparallel arithmetic type, and the right shall have parallel or nonparallel arithmetic type.

For the other operators, each operand shall have parallel or nonparallel arithmetic type consistent with those allowed by the corresponding binary operator.

Recall that the %=, %%=, <<=, and >>= operators were previously constrained in §3.3.16.

Semantics

Add:

When the left operand is of nonparallel scalar type and the right operand is of parallel type, the operation is a *scalar reduction assignment*. The value of the left operand is included in the reduction value computation (see §3.3.3.7). That is, the behavior of :

```
LHS op= RHS
```

is equivalent to:

```
LHS op= op= RHS
```

When the left operand is of parallel type and the right operand is of compatible parallel type, a *parallel reduction assignment* is performed. If the left operand is parallel indexed and specifies collisions (two positions of the parallel lvalue designate the same object), then values with the same destination are combined with each other and with the value at the destination using the associated operator. If there are no collisions, then a compound assignment is performed elementally on the values at the corresponding positions of the operands.

Examples

```
shape [10][20]S;
int:S x, y;
int:S *p2Pi;
int:S *:S Pp2Pi;
int result;
shape [10]T;
int ai[5];
int:T aP[5];

result += x;              /* Equivalent to result += +=x */

result -= x;              /* Equivalent to result += -=x */

result <?= x;             /* Equivalent to result <?= <?= x */

p2Pi += 1;                /* Increments pointer by 1 */

Pp2Pi += 1;               /* Increments all elements of parallel
                             pointer by 1*/

Pp2Pi += x;               /* Increments all elements by ints at
                             corresponding positions of x */

y *= x;                   /* Equivalent to y = y * x */

[x]y += x;                /* Equivalent to the following loop, in
                             the absence of collisions */

{ int i, j;
  for (i=0; i<10; i++)
    for (j=0; j<20; j++)
      [[i][j]x]y += [i][j]x;
}
```

```
ai[pcoord(T,0 % 5] += x;
                          /* ai[0] <-- ai[0] + [0]x + [5]x;
                             ai[1] <-- ai[1] + [1]x + [6]x;
                             ...
                             (there are collisions) */

aP[pcoord(T,0 % 5] += x;
                          /* [0]aP[0] <-- [0]aP[0] + [0]x;
                             [1]aP[1] <-- [1]aP[1] + [1]x;
                             ...
                             (there are no collisions) */

result %= x;              /* Constraint violation */
```

3.3.17 Comma Operator [ISO §6.3.17, X3.159 §3.3.17]

Examples

```
shape [10][10]S;
int:S x;

x++, x;              /* Denotes the parallel int value represented by
                        x after it has been postincremented. */
```

3.4 CONSTANT EXPRESSIONS [ISO §6.4, X3.159 §3.4]

Semantics

Add for parallel constants:

Parallel constants are introduced implicitly by use of nonparallel constant expressions in binary or ternary operations involving a parallel operand; in this case the nonparallel constant is promoted to a parallel constant by replicating the constant value.

Parallel constants may also be introduced explicitly by the use of cast expressions where the operand of the cast is a nonparallel constant.

A parallel address constant is an address constant designating a parallel object.

Examples

```
shape [10][10]S;
float:S x;
static int:S y;
int:S *p = &y;            /* Parallel address constant */

x = 1.0;                  /* Constant 1.0 is replicated to form
                             parallel double of shape S */

x = (float:S)1.0;         /* Explicit cast to parallel float */
```

3.5 DECLARATIONS [ISO §6.5, X3.159 §3.5]

Syntax

Revise as indicated:

declaration:
declaration-specifiers init-declarator-list$_{opt}$;

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier shape-specifier$_{opt}$ *declaration-specifiers*$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

shape-specifier:
: *shape-expression*

shape-expression:
(*expression*)
identifier
physical
void

Constraints

Add:

In shape expressions of the form "(*expression*)" used in file scope declarations, the *expression* shall be a *constant-expression*.

Shape expressions of the form "**void**" shall be used only in function prototypes or in pointer declarations.

The *declaration-specifiers* portion of a declaration shall contain at most one *shape-specifier* (see also §3.5.4.4).

A **void** shape designator can be used to specify a generic shape for parameter and return value types of a function prototype.

A *type*:**void** * type specifier can be used to specify a pointer to a parallel type of generic shape. A **void:void** * type specifier can be used to specify a pointer to a generic parallel type.

Semantics

Add:

If the specification of an array type includes any shape specifiers, the element type is so-specified, not the array type.

This can only happen through the use of a **typedef**. The resulting type is an array of parallel.

Examples

```
shape [10]S;

typedef:S a;                    /* Syntax error */

typedef int:S Pi;               /* Type Pi is a parallel int */

extern int:S b;                 /* b is an extern parallel int */

static int:physical c;          /* c is a static parallel int of
                                   physical shape */

auto float:S d;                 /* d is an auto parallel float */

register int:S e;               /* e is a parallel int */

volatile:S f;                   /* Syntax error */

int:(*shapeof(e)) g;            /* g is a parallel int */

short:S int:S h;                /* Constraint violation, multiple
                                   shape specifiers */

int:void f1();                  /* f1 returns a parallel int of
                                   unspecified shape */

int:void *p2Pigen;              /* p2Pigen is a pointer to parallel
                                   int of generic shape */

void:void *p2Pgen;              /* p2Pgen is a pointer to a generic
                                   parallel type */

typedef int atype[5];           /* atype is an array of int */

atype:S x;                      /* x is an array of 5 parallel ints
                                   of shape S */

int:S y[5];                     /* y is also an array of 5 parallel
                                   ints of shape S */
```

3.5.2 Type Specifiers [ISO §6.5.2, X3.159 §3.5.2]

Syntax

Add:

type-specifier:
...
shape-type-specifier

shape-type-specifier:
shape

Constraints

Add to the type specifier sets:

- shape-type-specifier

Add:

A shape type specifier shall only be used in a declaration of a shape object (see §3.5.4.4).

3.5.4 Declarators [ISO §6.5.4, X3.159 §3.5.4]

Syntax

Revise as indicated:

declarator:
shape-dimension-and-layout$_{opt}$ *pointer*$_{opt}$ *direct-declarator*

direct-declarator:
...
direct-declarator (*parameter-type-list*) *function-qualifier*$_{opt}$

function-qualifier:
elemental
nodal

pointer:
* *type-qualifier-list*$_{opt}$ *shape-specifier*$_{opt}$ *pointer-qualifier*$_{opt}$
* *type-qualifier-list*$_{opt}$ *shape-specifier*$_{opt}$ *pointer-qualifier*$_{opt}$ *pointer*

pointer-qualifier:
elemental

...

Constraints

A *shape-dimension-and-layout* shall only be used in a shape declarator (see §3.5.4.4).

3.5.4.1 Pointer declarators [ISO §6.5.4.1, X3.159 §3.5.4.1]

Constraints

Add for parallel pointers, pointers to parallel, and elemental functions:

If, in the declaration "`T D1`," `D1` has the form

* *type-qualifier-list*$_{opt}$ *shape-specifier* `D`

and `D` contains a function declarator, then the corresponding function type must be qualified as **`elemental`**. If `T` or `D` contains pointer declarators containing more than one shape specifier, then all the shape specifiers shall have compatible shape types.

Semantics

A declaration of the form "`T D1`" such that `T` is a parallel type and `D1` has the form

* *type-qualifier-list*$_{opt}$ *pointer-qualifier*$_{opt}$ `D`

declares a pointer to parallel.

A declaration of the form "`T D1`" such that `T` is a nonparallel type and `D1` has the form

* *type-qualifier-list*$_{opt}$ *shape-specifier pointer-qualifier*$_{opt}$ `D`

declares a parallel pointer to nonparallel (see §3.5.4.5).

A declaration of the form "`T D1`" such that `T` is a parallel type and `D1` has the form

* *type-qualifier-list*$_{opt}$ *shape-specifier pointer-qualifier*$_{opt}$ `D`

declares a parallel pointer to parallel (see §3.5.4.5).

If, in the declaration "`T D1`," `D1` has the form

* *type-qualifier-list*$_{opt}$ *shape-specifier pointer-qualifier*$_{opt}$ `D`

and `T` or `D` contains pointer declarators with shape specifiers, then all the shape specifiers shall denote the same shape, or the behavior is undefined.

Examples

```
shape [10]S, [20]T;
int:S * p2Pi;           /* p2Pi is a pointer to parallel int */

int *:S Pp2i;           /* Pp2i is a parallel pointer to int */

int:S *:S Pp2Pi;        /* Pp2Pi is a parallel pointer to parallel
                           int */

int *:S *:S Pp2Pp2i;    /* Pp2Pp2i is a parallel pointer to a
                           parallel pointer to int */
```

```
int *:S *:T ptr1;        /* Constraint violation, incompatible
                            shapes */
```

Add for elemental pointers:

If a pointer declaration specifies the type "**elemental**-qualified pointer to *T*" and the pointer type is parallel, either directly or indirectly, then *T* also is considered to be a parallel type of the same shape as the pointer type. Cases where the pointer type is considered to be indirectly parallel are illustrated below.

If *T* has a shape specifier, then the **elemental** pointer qualifier adds no additional meaning to the declaration.

The **elemental** pointer qualifier only has meaning when determining the type of the enclosing declaration. In particular, it adds no meaning with regard to operations upon values.

The **elemental** pointer qualifier is removed from the type when a variable, parameter, etc., is declared. It can only remain in a type for a **typedef** name or a structure member. However, when an object is declared using the **typedef** name or the structure type, the **elemental** pointer qualifier is removed from the type.

Examples

```
shape [10]S;
typedef int * elemental PTR;
struct s1 {
  PTR m1;
  int *m2;
};

int *:S elemental p1;    /* Pointer is directly parallel so p1 is
                            parallel pointer to parallel int */

int * elemental *:S p2; /* Inner pointer is directly parallel so
                           outer pointer indirectly becomes
                           parallel and therefore p2 is parallel
                           pointer to parallel pointer to parallel
                           int */

PTR:S p3;                /* p3 is parallel pointer to parallel
                            int */

PTR p4;                  /* p4 is pointer to int */

struct s1:S Ps;          /* m1 is indirectly parallel because
                            enclosing struct was declared parallel.
                            Therefore Ps.m1 is parallel pointer to
                            parallel int.  Note that Ps.m2 is
                            parallel pointer to nonparallel int. */

struct s1 s;             /* s.m1 is pointer to int, s.m2 is pointer
                            to int */

int:S *:S elemental p5; /* p5 is parallel pointer to parallel; no
                           additional meaning is added by the
                           elemental qualifier */
```

3.5.4.2 Array declarators [ISO §6.5.4.2, X3.159 §3.5.4.2]

Semantics

Add:

An array whose element type is parallel follows the usual semantics of arrays, requiring (conceptual) contiguity of array elements that allow subscripting and usual pointer arithmetic to be used interchangeably. However, since the element type is parallel, the elements of an element of the array are allowed to be discontiguous. This is consistent with the semantics of parallel objects. (See also §3.3.2.1, §3.3.3.2, and §3.3.6.)

Examples

```
shape [100][200]S;
int:S array[50];

/* The following evaluates to 1 */
&array[0] == &array[1] + sizeof(array[0]);
```

3.5.4.3 Function declarators (including prototypes) [ISO §6.5.4.3, X3.159 §3.5.4.3]

Constraints

Add for qualified functions:

A function declarator qualified with the **elemental** or **nodal** qualifiers shall contain a parameter type list.

A qualified function must provide a prototype specification of its parameters.

Add for elemental functions:

An **elemental**-qualified function shall not declare parameters of parallel or shape types, nor shall it return a parallel type.

Add for nodal functions:

A **nodal**-qualified function shall declare parameters only of the following types:

- parallel type of shape **void**; or
- pointer to nonparallel type; or
- nonparallel type.

Semantics

Add for elemental functions:

A function qualified with the **elemental** qualifier is called an *elemental function.* (See also §3.3.2.2, §3.6.6.4, and §3.7.1.)

Examples

```
int f1(int) elemental;    /* Elemental function returning int */
int *f2(int) elemental;   /* Elemental function returning pointer
                             to int */
```

Add for nodal functions:

A function qualified with the **nodal** qualifier is called a *nodal function*. (See also §3.3.2.2, §3.6.6.4, and §3.7.1.)

Examples

```
shape [1024]S;

int f1(int:S) nodal;      /* Nodal function returning int:physical
                             if called from nonnodal, and an int if
                             called from nodal */

int:S f2(int:S) nodal;    /* Nodal function returning an int:S */
```

3.5.4.4 Shape declarators [NEW]

Syntax

shape-dimension-and-layout:
shape-dimension-and-layout-specifier *shape-dimension-and-layout*$_{opt}$

shape-dimension-and-layout-specifier:
[*expression*$_{opt}$]
[*expression* **block** (*block-expression*)]
[*expression* **scale** (*scale-expression*)]

block-expression:
expression

scale-expression:
expression

Constraints

The *expression* used to specify the dimensions of a shape shall be an integral expression having a value greater than zero; if used at file scope or in an **extern** or **static** declaration, the *expression* shall be a constant integral expression. If any *shape-dimension-and-layout-specifier* in a *shape-dimension-and-layout* specifies a dimension, all the specifiers shall specify a dimension.

If at file scope or in an extern or static declaration, **block** and **scale** expressions shall be constant integral expressions. If any *shape-dimension-and-layout-specifier* includes a **block** layout specification, then no other *shape-dimension-and-layout-specifier* in the same *shape-dimension-and-layout* shall include a **scale** layout specification, and vice versa.

Mixing of **block** and **scale** specifiers in a shape declaration is not allowed.

Semantics

A shape declarator denotes an object of type shape. As discussed in §3.1.2.5, a shape is a type whose values consist of the following components: rank, dimensions, context, and layout. A shape type may be fully unspecified, partially specified, or fully specified, depending on the rank and dimensions specified.

The **block** and **scale** expressions used to specify the layout of a shape shall be nonnegative integral expressions, and **scale** expressions shall be greater than zero; otherwise, the behavior is undefined.

The object denoted by a shape declarator is *fully unassigned* if its shape type is fully unspecified; all component values of the shape object are undefined. The object denoted by a shape declarator is *partially assigned* if its shape type is partially specified; its rank component is assigned the rank of the shape type, and all other components are undefined. The object denoted by a shape declarator is *fully assigned* if its shape type is fully specified; its rank, dimensions, and layout components are assigned the rank, dimensions, and layout specified in the shape type, and its context is assigned a value such that all positions are initially active.

The rank of a shape is determined from the number of *shape-dimension-and-layout-specifier*'s given. The dimension in each axis is optionally specified as the *expression* in a *shape-dimension-and-layout-specifier*.

The **block** and **scale** specifiers suggest a layout or data distribution for parallel operands associated with this shape declarator. The **block** specifier suggests that data is to be distributed in blocks of the indicated size. The **scale** specifier suggests that data be partitioned into maximal blocks with block dimensions proportional to the scale expressions in the corresponding dimensions; each partition receives the indicated block size, if the shape is multidimensional; or a single block, if the shape is one-dimensional. Partitions of blocks are distributed in round-robin fashion to the nodes of the target architecture.

> Note: if the number of partitions is not a multiple of the number of nodes, not all nodes receive an equal number of partitions; if the data size is not a multiple of the determined block size, not all partitions are necessarily the same size.

A **block** specifier equal to the dimension in a given axis suggests that no distribution is desired across that axis of an object.

If no **block** or **scale** specifier is given, or if a **block** specifier is equal to zero, the implementation determines an appropriate distribution for parallel operands of this shape on the target architecture.

The implementation is free to use a different layout than what is suggested. However, for identical layout specifications, an implementation shall use identical layout. (See §4.14.1.2 and §4.14.1.4.)

Examples

```
shape [100 block (10)]S;
            /* Suggests objects of shape S are to be distributed
               in blocks of 10 elements per partition; each
               partition will have blocks of 10 contiguous
               elements. */

shape [100 block (100)]T;
            /* Suggests objects of shape T are not to be
               distributed; all elements are in the same
               partition */

shape [100 block (0)]U;
shape [100] V;
            /* Suggests objects of shape U or V are to be
               distributed as determined by the implementation */

shape [100 scale (1)]W;
            /* Suggests objects of shape W are to be
               distributed with 1 block per partition where
               the implementation determines the block size */

shape [100 block (10)][100 block (20)]X;
            /* Suggests objects of shape X are to be
               distributed in blocks of 10x20 elements */

shape [100 scale (1)][100 scale (2)]Y;
            /* Suggests objects of shape Y are to be distributed
               in blocks with kx2k elements; k will be
               approximately sqrt((100x100)/2n) where n is the
               number of partitions */

shape [n block (m)]Z;
            /* Constraint violation if at file scope */

shape [0 scale(0)][10 block (-10)]ZZ;
            /* Constraint violations:  dimension and scale
               expressions must be greater than 0, and can't mix
               block and scale specifiers in a declaration */
```

3.5.4.5 Parallel object declarators [NEW]

Constraints

The element type of a parallel object declarator shall not be, directly or indirectly, another parallel type.

The element type of a parallel object declarator shall not be, directly or indirectly, a structure or union type that has a member that has parallel type or a member that has shape type.

The element type of a parallel object declarator shall not be, directly or indirectly, a shape type.

Each *shape-specifier* in the declaration shall be an lvalue of type shape.

> If an expression is used in the *shape-specifier*, the expression must denote an lvalue of type shape. A shape-valued expression must denote an object in addressable memory to be used in a parallel object declaration.

Semantics

A declaration that includes a *shape-specifier* is a parallel object declarator. Parallel object declarators denote parallel objects–i.e., variables of parallel type. If the *shape-specifier* is a part of a *pointer*, the parallel object is a *parallel pointer*, a parallel object whose element type is a pointer type (see §3.5.4.2).

If the shape object denoted by the *shape-specifier* is not fully assigned when the parallel object declarator is declared, the behavior is undefined.

Examples

```
shape [100][4][25]S;
shape []T;
typedef int:S parint_t;
shape f();

int:S x;                    /* x is a parallel int */

int *:S Pp2i;               /* Pp2i is a parallel pointer to int */

parint_t:S y;               /* Constraint violation; element type is
                               parallel type */

struct { parint_t e; }:S w;
                            /* Constraint violation; element type is
                               struct with parallel member */

int:(f()) z;                /* Constraint violation, f returns a shape
                               value, not a shape object */

int:T z;                    /* Undefined behavior, T is not fully
                               assigned */
```

3.5.5 Type names [ISO §6.5.5, X3.159 §3.5.5]

Syntax

The syntax for abstract declarators was not extended to allow a *shape-specifier* as with declarators, due to ambiguities that would otherwise be introduced in recognizing (optional) expressions in square brackets `[]`. These could be either a shape dimension specifier or an array dimension specifier. Therefore, abstract declarators for shape types or arrays of shape types must first define the shape type with a **`typedef`**.

Semantics

Add:

In a *direct-abstract-declarator* containing square brackets `[]` the abstract declarator being constructed designates an array type and not a shape type.

Examples

```
typedef shape [] shape_t1;
typedef shape [10] shape_t2;

void f1(shape);           /* Abstract declarator of fully
                             unspecified shape type */

void f2(shape []);        /* Abstract declarator of type array of
                             fully unspecified shape type */

void f3(shape_t1 [])      /* Abstract declarator of type array of
                             partially specified shape type */

void f4(shape_t2 [])      /* Abstract declarator of type array of
                             fully specified shape type */

void f5(int:void);        /* Abstract declarator of type parallel
                             int of void shape */

void f6(int:void []);     /* Abstract declarator of type array of
                             parallel int of void shape */

void f7(int *:void);      /* Abstract declarator of type parallel
                             having element type pointer to int and
                             void shape */

void f8(int:void *);      /* Abstract declarator of type pointer to
                             parallel int of void shape */
```

3.5.7 Initialization [ISO §6.5.7, X3.159 §3.5.7]

Constraints

Revise as indicated:

...

All the expressions in an initializer for an object or parallel object that has static storage duration, or in an initializer list for an object or parallel object that has aggregate or union type shall be constant expressions.

...

Semantics

Add for parallel object declarators:

An initializer for a parallel object declarator specifies the initial value of every element of the parallel object. The initializer value is replicated and assigned to the element object values at each position.

An initializer for an array whose element type is a parallel type initializes each element of the parallel object with the initializer value specified for the corresponding array element; that is, each array element is initialized by replicating the corresponding (nonparallel) value in the initializer list.

Examples

```
shape [200]S;

int:S x = 10;                  /* Initializes all int elements to 10 */

int:S a[5] = { 9, 4, 6, 2, 5 };
                               /* Initializes a[0] to be a parallel int
                                  whose values are all 9, a[1] to be
                                  parallel int whose values are all 4,
                                  etc. */

struct { int i; float f; }:S ss = { 10, 1.5};
                              /* Initializes all elements to have member
                                 i initialized to 10 and member f
                                 initialized to 1.5 */

union { int i; float f; }:S uu = { 1 };
                              /* Initializes member i of all elements
                                 to be 1 */
```

3.6 STATEMENTS [ISO §6.6, X3.159 §3.6]

Syntax

Revise as indicated:

statement:
...
contextualization-statement

3.6.4 Selection statements [ISO §6.6.4, X3.159 §3.6.4]

Add clarification:

Constraints

The controlling expression of an `if` or `switch` statement shall be of nonparallel type. The expression of each `case` label shall be of nonparallel type.

3.6.5 Iteration statements [ISO §6.6.5, X3.159 §3.6.5]

Revise for clarification:

Constraints

The controlling expression of an iteration statement shall have nonparallel scalar type.

3.6.5.3 The `for` statement [ISO §6.6.5.3, X3.159 §3.6.5.3]

Add for clarification:

Both *expression-1* and *expression-3* are allowed to have parallel type.

3.6.6 Jump statements [ISO §6.6.6, X3.159 §3.6.6]

Semantics

Add for clarification:

If a jump statement is used to leave the nested context of a **where** or **everywhere** statement, the contextualization at the destination is restored for all shapes that were constrained by the contextualization statement(s) exited to the contextualization of the destination.

If a jump statement is used to enter the nested statement part of a contextualization statement or to enter a block declaring a parallel variable or shape, the behavior is implementation-defined.

3.6.6.4 The return statement [ISO §6.6.6.4, X3.159 §3.6.6.4]

Semantics

Add for elemental functions:

When an elemental function that is executed nonelementally executes a **return** statement, the semantics are the same as a return for a nonelemental function. When an elemental function that is executed elementally executes a return to an elemental caller, the semantics are the same as a return for a nonelemental function.

An elemental function that is executed elementally executes as if the function was called once per active position for the established shape of its parallel argument(s) (see §3.3.2.2). When a **return** statement with an expression is executed at a position, and the caller is a nonelemental function, the value of the expression is assigned to that position in a parallel return value. If the element type of the parallel return value is different from the type of the return expression, then the expression value is converted as if by assignment. Control is not returned to the caller until all active positions have executed a **return** statement.

Examples

```
typedef struct coord { double s, y; } COORD;

double distance(COORD pt1, COORD pt2) elemental
{
  double x_delta, y_delta;

  x_delta = pt1.x - pt2.x;
  y_delta = pt1.y - pt2.y;
  return sqrt(x_delta*x_delta + y_delta*y_delta);
}

main()
{
  COORD a, b, sresult;
  COORD:S c, d, presult;

  sresult = distance(a,b);     /* Executes in nonparallel mode */

  presult = distance(c,d);     /* Executes in parallel mode,
                                  elementally */
}
```

Add for nodal functions:

When a nodal function that is declared to return a nonparallel value is returning to a nonnodal execution environment, then a parallel value of **physical** shape is returned. Each thread executing the nodal function contributes one element of the return value.

When a nodal function that is declared to return a nonparallel value is returning to a nodal execution environment, then a nonparallel value is returned in the usual manner.

For a nodal function that is declared to return a parallel value, the shape of the return expression shall be a shape that was created when passing an argument to the nodal function (see §3.3.2.2); otherwise the behavior is undefined. If S is the shape of the return expression and T is the shape of the argument that caused S to be created, then the shape of the parallel value returned to the caller will be T. Note that if returning to a nodal environment, S will be identical to T. The correspondence of positions of the shape of the return expression and positions in the shape of the value returned to the caller is implementation-defined, but the mapping used will be the inverse of the mapping used when passing arguments of the latter shape to a nodal function.

In all cases, when a nodal function executes a **`return`** statement with an expression, the expression value is converted as if by assignment if the element type of the return expression is different than the element type of the declared return type.

In addition, if one or more of the threads executes a **`return`** statement without an expression, and the value of the function call is used by the caller, the behavior is undefined.

This is the multithreaded version of the single-threaded semantics for C.

The threads executing a nodal function synchronize upon return to a nonnodal execution environment: control is not returned to the caller until all threads have executed a return.

Examples

```
int [10]S;
int [100]T;
int:S x,y,z;
int:T w;
int:physical one_per_node;

int:void nodal_add(int:void a, int:void b) nodal
{
  return(a + b);          /* Shape of return value as seen by caller
                             will be the same as the shape of the
                             two arguments, which in this case must
                             be the same */
}
int:physical physical_increment(int a) nodal
{
  return(a + 1);          /* Caller will pass an int:physical and
                             get an int:physical in return */
}
int:void nodal_bug(int:void a) nodal
{
  shape [64]S2;

  return((int:S2)19);     /* Undefined behavior:  shape of return
                             value must be derived from shape of a
                             parameter */
}

main()
{
  x = nodal_add(y,z);     /* OK, because x, y, z have same shape */

  w = nodal_add(y,z);     /* Undefined behavior, w not of same shape
                             as y, z */

  one_per_node = physical_increment(one_per_node);
                          /* OK */

  nodal_bug(w);           /* Undefined behavior */
}
```

3.6.7 Contextualization [NEW]

Syntax

contextualization-statement:
where (*mask-expression*) *statement*
where (*mask-expression*) *statement* **else** *statement*
everywhere (*shape-expression*) *statement*

Semantics

A contextualization statement modifies the context of a shape for the duration of the substatement(s) of the statement. The context component of the shape is restored to its previous value at the end of the substatement. Note that context-modifying statements may be nested, and the effects on the contextualization are recursively defined for the nested statements.

All parallel values have an associated shape and this shape contains its current context. Therefore, a contextualization statement first modifies the context of the shape of its mask expression, and then a function in the substatement is called. If this function accesses a value of the same shape as the mask expression, then the modified context will be used when evaluating expressions of this shape.

3.6.7.1 The `where` statement [NEW]

Constraints

The mask expression shall evaluate to a parallel scalar value of shape *S*.

The mask expression must be a parallel type whose element type is a scalar, which is either an arithmetic type or a pointer type.

Semantics

The context of shape *S* will be constrained by the value of the mask expression for the duration of the first substatement, and if there is an **`else`** and a second substatement, the context of shape *S* will be constrained by the value of the logical complement of the value of the mask expression for the duration of the second substatement. The mask expression (or its complement) constrains the context by further limiting what positions are active: it can only make active positions inactive for the duration of the substatement(s); it cannot make inactive positions active.

Each parallel operand of shape *S* accessed within the substatement(s) of the **`where`** statement will have the context narrowed by the mask expression. Parallel operands that are not of shape *S* within the substatement(s) are not affected by the narrowed context.

Examples

```
shape [100]S, [10][10]T;
int:S x, y, mask;
int:T z;

where (mask > 0) {
   x++;                  /* Affected by the narrowed context */
   y++;                  /* Affected by the narrowed context */
   z++;                  /* Not affected by the narrowed context */
}
else {
   x++;                  /* Affected by the narrowed context */
   y++;                  /* Affected by the narrowed context */
   z--;                  /* Not affected by the narrowed context */
}
```

3.6.7.2 The everywhere statement [NEW]

Constraints

The shape expression shall evaluate to a shape *S*.

Semantics

The context of the designated shape *S* will be as if assigned a parallel value of 1 for the duration of the substatement. This will widen the context so that all positions are active.

Each parallel operand of shape *S* within the substatement of the **everywhere** statement will have a widened context in which all positions are active. Parallel operands that are not of shape *S* are not affected by the widened context.

Examples

```
shape [100]S, [10][10]T;
int:S x, y, mask;
int:T z;

everywhere (S) {
   x++;                    /* Affected by the widened context */
   y++;                    /* Affected by the widened context */
   z--;                    /* Not affected by the widened context */
}
```

3.7 EXTERNAL DEFINITIONS [ISO §6.7, X3.159 §3.7]

3.7.1 Function Definitions [ISO §6.7.1, X3.159 §3.7.1]

Constraints

Add for elemental functions:

An elemental function shall not contain any variable declarations of parallel or shape types, or types derived from parallel or shape types.

An elemental function shall not contain a **where** or **everywhere** statement.

An elemental function shall not contain reduction or parallel-indexing operations.

An elemental function shall call only elemental functions.

An elemental function shall not reference a parallel variable with file scope.

A variable declared within an elemental function shall not have static storage class.

Add for nodal functions:

A nodal function shall call only nodal functions and elemental functions.

A nodal function shall not reference any identifier with file scope, except nodal or elemental functions, and any of the functions listed in §4.14.

Semantics

Add for elemental functions:

An elemental function that is executed nonelementally has the same semantics as a nonelemental function.

An elemental function that is executed elementally executes as if the function was called once per active position of the shape of its parallel argument(s). Assignment of arguments to parameters also occurs on a per active position basis, with conversion, if necessary, between the element type of a parallel argument and the type of the corresponding parameter.

If an elemental function accesses a nonparallel object that was not defined within an elemental function, the behavior is undefined.

Since an elemental function executing elementally executes as if called once per active position, pointer dereferences have special position-oriented semantics. When the function is being executed as if for a particular active position, a dereference of a pointer that was created outside the elemental function as a "parallel pointer to parallel" accesses only the object in the corresponding position of the pointer target.

Examples

```
/* Linked list node for use either in parallel or
   nonparallel setting */

#include <dpce.h>

typedef struct node * elemental LINK;

struct node {
    int count;
    LINK next;     /* Pointer to either parallel or
                      nonparallel struct node */
};

/* Elemental function to build a linked list:
     n linked nodes are built
     count fields are initialized from values argument
*/
struct node *build(int n, int values[]) elemental
{
  /* Automatic "flow through" for elemental function locals:
     in elemental function everything is parallel or
     everything is nonparallel
  */
  struct node *last = NULL, *new;
  int i;

  for (i = n-1; i >= 0; i--) {
    new = (struct node*) malloc(sizeof(struct node));
    new->count = values[i];
    new->next = last;
    last = new;
  }
  return last;
}

/* Elemental function to sum "count" fields of a "struct
   node" list
*/
int sum(struct node *list) elemental
{
  int ret = 0;

  while (list != NULL) {
    ret += list->count;
    list = list->next;
  }
  return ret;
}

shape [100]S;

int n;
int values[10];
int:S many_values[10];
int:S many_n;
```

```
LINK head;          /* Nonparallel pointer to nonparallel
                       struct node */

LINK:S many_head; /* Parallel pointer to parallel struct
                     node */

main()
{
  /* Call to init values, many_values, many_n */
  init_stuff();

  head = build(n,values);                    /* nonparallel */
  many_head = build(many_n,many_values);  /* parallel */

  printf("sum of head is %d\n", sum(head));

  printf("sum of many_head is %d\n", +=(sum(many_head)));

  printf("sum of just [4]many_head is %d\n",[4](sum(many_head)));

  /* Or perhaps more efficiently */
  where (pcoord(S,0) == 4) {
    printf("sum of just [4]many_head is %d\n",+=(sum(many_head)));
  }
}
```

Add for nodal functions:

The body of a nodal function executes in a single-node environment (see §3.3.2.2).

If a pointer to a nonparallel object created in a nonnodal execution environment is dereferenced in a nodal environment, the behavior is undefined.

Examples

```
struct x {
     int *p;              /* p is a pointer to nonparallel */
};

struct x a;
int n;

void nodal_f(struct x param) nodal
{
  struct x local;

  local = param;
  *local.p = 19;          /* Undefined behavior if value of a
                             pointer to nonparallel originated in
                             nonnodal environment; therefore can't
                             dereference it in nodal environment */
}

void another_nodal_f(void) nodal
{
  struct x local;
  int i;

  local.p = &i;
  nodal_f(local);         /* OK, the pointer value in local.p
                             originated in nodal environment */
}

main()
{
  a.p = &n;
  nodal_f(a);             /* Argument will be promoted to struct
                             x:physical but undefined behavior will
                             result when nodal_f dereferences the
                             member */

  another_nodal_f();      /* OK */
}
```

Inside nodal functions, pointer values that were created in a nonnodal execution environment as "pointer to parallel" have special position-oriented semantics. All pointer values accessed, directly or indirectly, for a parameter value derived for a particular position reference only objects in that position of the pointer target.

Since nodal functions may not reference identifiers with file scope except for nodal and elemental functions, all pointer values that were created in a nonnodal environment may only be accessed via a data value passed as an argument to the nodal function that caused nodal execution to begin. All values of arguments to a nodal function are derived for a particular position of a particular shape. Therefore parameter values also have this association with positions of shapes.

Examples

```
/* Using a nodal function to implement the
   Sieve of Eratosthenes */

#define N 4800000

typedef unsigned char BOOL;

shape [N scale (1)] natural;

void StrikeOut ( int local_dim;
                 BOOL *local_primes[],
                 BOOL *local_active[],
                 int local_coords[],
                 int current_prime
               ) nodal
{
  int lowbound = local_coords[0];
  int offset = lowbound % current_prime;
  int i;

  /* Where is the first number that is a multiple of the
     current prime? */
  if (offset == 0) i = 0;
  else i = current_prime - offset;

  /* Stride through to knock out multiples of current prime */
  for ( ; i < local_dim; i += current_prime) {
    *local_primes[i] = 0;
    *local_active[i] = 0;
  }
}

main ()
{
  /* Each position of prime represents itself and is to be
     evaluated for primality.  The algorithm requires each
     node to have a single contiguous block of numbers. */

  int next_prime = 2;
  BOOL:natural active;
  BOOL:natural prime;

  where (pcoord(natural,0) < 2)
    prime = active = 0;
  else where (pcoord(natural,0) == 2) {
    prime = 1;
    active = 0;
  }
  else
    prime = active = 1;
```

```
while (|= active)
    where (active) {

      /* Could strike out nonprimes by testing all active
         elements using %, but nodal function can apply more
         efficient method exploiting contiguous
         distribution.  Casts are needed on first two
         parameters to cause pointers to each active prime
         to be generated. */

      StrikeOut(nodepositionsof(natural),
                (BOOL:natural *:natural) &prime,
                (BOOL:natural *:natural) &active,
                pcoord(natural,0),
                next_prime);

      where (prime)
        next_prime = (<?= pcoord(natural,0));

      where (next_prime == pcoord(natural,0))
        active = 0;
    }

    printf("There are %d primes less than %d\n",
           (+= prime), N);
  }
}
```

4. LIBRARY [ISO §7, X3.159 §4]

4.1 INTRODUCTION [ISO §7.1, X3.159 §4.1]

4.1.2 Standard Headers [ISO §7.1.2, X3.159 §4.1.2]

Add to standard headers:

<dpce.h>

This header file defines the predefined shape identifier **physical** and declares new functions that are available for implementations supporting DPCE. In addition, it redeclares specific functions declared in **<math.h>**, **<stdlib.h>**, and **<string.h>** to be elemental functions; if these header files are also included, the correct composite type will be formed for these functions for DPCE implementations (see §3.1.2.6). This header file must be included by all translation units that use DPCE.

4.1.6 Use of Library Functions [ISO §7.1.7, X3.159 §4.1.6]

There is one exception to the rule that library functions may be declared without the use of their associated header. If a translation unit declares DPCE functions without first including **<dpce.h>**, its behavior is undefined.

4.14 DATA PARALLEL UTILITIES <dpce.h> [NEW]

The header **<dpce.h>** defines the predefined shape identifier **physical** and the following types, and declares the functions specified below. Note: all the functions declared here are accessible to elemental and nodal functions as described in §3.7.1.

The exact definition of **physical** is implementation specific. The following definition is a placeholder for that specific definition, which must be fully specified.

```
shape []physical;
```

The types **dpce_layout_specs_t** and **dpce_layout_t** are defined as follows:

```
typedef enum { DPCE_BLOCK, DPCE_SCALE } dpce_layout_specs_t;

typedef struct {
                  size_t size;
                  dpce_layout_specs_t spec;
                } dpce_layout_t;
```

4.14.1 General purpose DPCE utilities [NEW]

There are seven functions from **<stdlib.h>** that are redeclared here as elemental functions. The intent is that the following functions will behave as they are described in §7.10 of [2] when operating on nonparallel operands, and will behave elementally when any operand is parallel.

```
int abs(int j) elemental;
int atoi(const char *nptr) elemental;
long int atol(const char *nptr) elemental;
```

```
void qsort(void *base, size_t nmemb, size_t size,
        int (*compar)(const void*, const void*) elemental);
void *calloc(size_t nmemb, size_t size) elemental;
void *malloc(size_t size) elemental;
void *realloc(void *ptr, size_t size) elemental;
```

When executing elementally, the functions **malloc**, **calloc**, and **realloc** return a parallel pointer to parallel. However, each element of this pointer value has only position-oriented semantics and may only be used to reference an object in its same position. In particular, if a single element of the value is converted (for example, via parallel indexing), to a nonparallel pointer to parallel and the result is dereferenced, then the behavior is undefined. If it is necessary to create a pointer value with both position-oriented and "global" semantics, then the functions **palloc** (§4.14.1.6) and **pfree** (§4.14.1.8) should be used.

Examples

```
shape [10]S;
int:S *:S many_ptr_to_many;
int:S * one_ptr_to_many;
int:S many;
int:S many2;

one_ptr_to_many = &many;

*one_ptr_to_many = 19;          /* All positions of many receive 19
                                   */

many_ptr_to_many = &many;       /* All positions point to many */

[5]many_ptr_to_many = &many2; /* Position 5 points to many2 */

*many_ptr_to_many = 199;        /* Positions 0-4, 6-9 of many get
                                   199, position 5 gets 199 */

one_ptr_to_many = [5]many_ptr_to_many;
                                /* Assign pointer to many2 */

*one_ptr_to_many = 1999;        /* All positions of many2 get 1999
                                   */

many_ptr_to_many = (int:S *:S)malloc((size_t:S)sizeof(int:S));
                                /* Each position points to an int in
                                   its own position */

one_ptr_to_many = [5]many_ptr_to_many;
                                /* Converts position-oriented
                                   parallel pointer to nonparallel
                                   pointer */

*one_ptr_to_many = 19999;       /* Undefined behavior */
```

There are twelve new general purpose utility functions declared as follows.

4.14.1.1 The dimof Function [NEW]

Synopsis

```
#include <dpce.h>
int dimof(shape s, int a);
```

Description

The **dimof** function extracts the dimension of the shape **s** in the specified axis **a**. If the axis specified by **a** is outside the range of the rank specified for shape **s**, or if shape **s** is partially assigned or fully unassigned, the behavior is undefined.

Returns

The **dimof** function returns the extracted dimension.

Examples

```
shape [10][20][30]S;
shape []T, U;
int dim0, dim1, dim2;

dim0 = dimof(S,0);          /* Assigns dim0 to be 10 */
dim1 = dimof(S,1);          /* Assigns dim1 to be 20 */
dim2 = dimof(S,2);          /* Assigns dim2 to be 30 */

dimof(S,5);                 /* Undefined behavior */
dimof(T,0);                 /* Undefined behavior */
dimof(U,0);                 /* Undefined behavior */
```

4.14.1.2 The `layoutof` Function [NEW]

Synopsis

```
#include <dpce.h>
dpce_layout_t layoutof(shape s, axis a);
```

Description

The **`layoutof`** function extracts the layout specification for shape **`s`** along the designated axis **`a`**. The return value's spec member indicates whether the layout specification is a **`block`** or **`scale`** specifier, and its size member indicates the corresponding specification. If shape **`s`** is fully assigned, the returned layout specification is what was specified in the declaration of the shape type; otherwise it will be a **`block`** specification of 0. If the axis specified in **`a`** is outside the range of the rank of shape **`s`**, or if the shape **`s`** is not fully assigned, the behavior is undefined. This function gives access to the user-defined layout specification. (See §3.5.4.4.)

Returns

The **`layoutof`** function returns the extracted layout specification.

Examples

```
shape [1024 block (256)]S;
shape T;
dpce_layout_t S_layout;

layoutof(S,1);                  /* Undefined behavior */
layoutof(T,1);                  /* Undefined behavior */

S_layout = layoutof(S,0);       /* Extract layout */

/* Use extracted layout */
{ shape [2048 block (layoutof(S,0))]T;
  ...
}

T = S;                          /* T becomes fully assigned, gets
                                   layout specification from S, so
                                   the next expression evaluates
                                   to 1 */
layoutof(S,0) == layoutof(T,0);
```

4.14.1.3 The newshape Function [NEW]

Synopsis

```
#include <dpce.h>
shape newshape(int rank, int *dimensions,
               dpce_layout_t *specs);
```

Description

The **newshape** function returns a shape value of the specified **rank** and **dimensions**, initializes the context so that all positions are active, and initializes the layout for the shape value as specified by the **specs** argument. The **rank** argument must be an integer greater than 0. The **dimensions** must be an integer array of *n* values, where *n* is the specified rank; each dimension shall be an integer greater than 0. The **specs** argument is an array of *n* **dpce_layout_t** specifications for the shape value to be returned; each layout specification shall be a valid **dpce_layout_t**; otherwise, the behavior is undefined.

Returns

The **newshape** function returns the initialized shape value.

Examples

```
shape [10]S;
shape []T;
shape U;
int ten = 10;
int thousand = 1000;
int dims[3] = {50,50,20};

dpce_layout_t specs_1[] = { {0, DPCE_BLOCK} };
dpce_layout_t specs_2[] = { {10, DPCE_BLOCK} };
dpce_layout_t specs_3[] = { {50, DPCE_BLOCK},
                            {50, DPCE_BLOCK},
                            {20, DPCE_BLOCK} };

S = newshape(0, 0, 0);  /* Undefined */

S = newshape(1, &ten, specs_1);
                        /* Returns 1-dimensional shape value with
                           default distribution */

T = newshape(1, &thousand, specs_2);
                        /* Returns 1-dimensional shape value with
                           blocks of 10 elements per partition;
                           makes T a fully assigned shape */

U = newshape(3, dims, specs_3);
                        /* Returns 3-dimensional shape value with
                           no distribution; makes U a fully
                           assigned shape */
```

4.14.1.4 The nodeof Function [NEW]

Synopsis

```
#include <dpce.h>
int:void nodeof(shape s);
```

Description

The **nodeof** function extracts the mapping of positions to nodes from the layout component of the given shape **s**, where each node in the implementation has a unique integral designation. If the shape **s** is not fully assigned, the behavior is undefined. This function gives access to the implementation-defined layout for the given shape.

When this function is executed in a nodal environment the node designation in each position will be that of the node on which it is executing.

Returns

The **nodeof** function returns a parallel **int** of shape **s** whose value at each position is the unique node designation on which each position of the given shape is mapped by the shape's layout.

Examples

```
/* Assume for the following example that <dpce.h> defines:

    shape [2]physical;

   and

    nodeof(physical) => { 0, 1 }

   That is, the unique node designation for the two nodes
   of this implementation are 0 and 1.
*/

shape [10]S, [10 block 3]T, U;

nodeof(S);              /* Denotes parallel int with values:
                           { 0, 0, 0, 0, 0, 1, 1, 1, 1, 1 } */

nodeof(T);              /* Denotes parallel int with values:
                           { 0, 0, 0, 1, 1, 1, 0, 0, 0, 1 } */

nodeof(U);              /* Undefined behavior */
```

4.14.1.5 The nodepositionsof Function [NEW]

Synopsis

```
#include <dpce.h>
int:physical nodepositionsof(shape s);
```

Description

The **nodepositionsof** function computes the total number of positions of the given shape **s** that are mapped to each node of the implementation environment. If the shape **s** is not fully assigned, the behavior is undefined.

Returns

The **nodepositionsof** function returns a parallel **int** of **physical** shape whose value at each position is the total number of positions of shape **s** mapped to the node that contains that element of shape **physical**.

Examples

```
/* Assume for the following example that <dpce.h> defines:

    shape [2]physical;

   and

    nodeof(physical) => { 0, 1 }

   That is, the unique node designation for the two nodes
   of this implementation are 0 and 1.
*/

shape [10]S, [10 block 3]T, U;

nodepositionsof(S);          /* Denotes parallel int {5, 5} */
nodepositionsof(T);          /* Denotes parallel int {6, 4} */
nodepositionsof(U);          /* Undefined behavior */
```

4.14.1.6 The palloc Function [NEW]

Synopsis

```
#include <dpce.h>
void:void *palloc(shape *s, int bsize);
```

Description

The **palloc** function allocates space for a parallel object of the shape pointed to by **s** and having element size of **bsize** bytes, and associates the given parallel object with the shape object pointed to by **s**.

The shape pointed to by **s** must be fully assigned when **palloc** is called; otherwise, the behavior is undefined.

Returns

The **palloc** function returns a null pointer or a pointer to the allocated space.

The return value usually needs to be cast to a particular parallel type.

Examples

```
shape [20][20]S, [N][M]T, U;
int:S *x;

palloc(&S, sizeof(*x));                  /* Allocates space for a 20x20
                                            parallel int */
palloc(&T, sizeof(*x));                  /* Allocates space for an NxM
                                            parallel int */

x = (int:S *)palloc(&S,sizeof(*x)); /* Casting to pointer to
                                            specific parallel type */

palloc(&U, sizeof(*x));                  /* Undefined */
```

4.14.1.7 The pcoord Function [NEW]

Synopsis

```
#include <dpce.h>
int:void pcoord(shape s, axis a);
```

Description

The **pcoord** function is a parallel axis coordinate value constructor. For a given axis specifier **a** and a given shape **s**, this function returns a parallel **int** of shape **s** whose value at each position is initialized to the coordinate at the position in the given axis. If **a** is an axis number that is outside the range of the rank of shape **s**, or if **s** is not fully assigned, the behavior is undefined.

Returns

The **pcoord** function returns a parallel **int** of the same shape as **s**.

Examples

```
shape [10]S, [2][2]T;
int:S x;
int:T y;

x = pcoord(S,0);          /* Parallel int of shape S having value 0
                             in position [0], 1 in position [1],
                             ..., and 9 in position [9]. */

y = pcoord(T,1);          /* Parallel int of shape T having value 0
                             in positions [0,0] and [1,0], and 1
                             in positions [0,1] and [1,1]. */

pcoord(T,3);              /* Undefined behavior */
```

4.14.1.8 The pfree Function [NEW]

Synopsis

```
#include <dpce.h>
void pfree(void:void *pptr);
```

Description

The **pfree** function causes the space pointed to by **pptr** to be deallocated. If **pptr** is a null pointer, no action occurs. If **pptr** does not match a pointer returned earlier by the **palloc** function, or if the space has already been deallocated by an earlier call to **pfree**, the behavior is undefined.

Returns

The **pfree** function does not return a value.

Examples

```
shape [20][20]S;
int:S *x, y;

x = palloc(&S, sizeof(*x));          /* Allocates space */

pfree(x);                            /* Deallocates space */

pfree(&y);                           /* Undefined behavior */
pfree(x);                            /* Undefined behavior */
```

4.14.1.9 The positionsof Function [NEW]

Synopsis

```
#include <dpce.h>
int positionsof(shape s);
```

Description

The **positionsof** function computes the number of positions in any parallel object of shape **s**, which is the product of all the dimensions of the shape. If the shape **s** is not fully assigned, the behavior is undefined.

Returns

The **positionsof** function returns the computed number of positions.

Examples

```
shape [10][30]S, T;
int n;

n = positionsof(S);          /* Returns 300 */

positionsof(T);              /* Undefined behavior */
```

4.14.1.10 The **prand** Function [NEW]

Synopsis

```
#include <dpce.h>
int:void prand(shape s);
```

Description

The **prand** function provides a parallel version of the **rand** function. It constructs a parallel **int** of the same shape as **s**. The sequence of values generated in the active positions of the return value is implementation-defined. If the shape **s** is not fully assigned, the behavior is undefined.

Returns

The **prand** function returns a parallel **int** of the same shape as **s**.

4.14.1.11 The **psrand** Function [NEW]

Synopsis

```
#include <dpce.h>
void psrand(unsigned seed);
```

Description

The **psrand** function provides the same functionality with respect to the **prand** function as the **srand** function does with respect to the **rand** function. A call to **psrand** affects all subsequent calls to **prand** regardless of the shape argument to **prand**.

Returns

The **psrand** function does not return a value.

4.14.1.12 Therankof Function [NEW]

Synopsis

```
#include <dpce.h>
int rankof(shape s);
```

Description

The **rankof** function extracts the rank component value of the shape **s**. If the shape **s** is not fully assigned or partially assigned, the behavior is undefined.

Returns

The **rankof** function returns the extracted rank value.

Examples

```
shape [10][50]S, T;
int rank;

rank = rankof(S);          /* Returns rank of shape S == 2 */

rankof(T);                 /* Undefined behavior */
```

4.14.2 DPCE mathematics [NEW]

The following functions from **<math.h>** are redeclared here as elemental functions. The intent is that they will behave as they are described in §7.10 of [2] when operating on nonparallel operands, and will behave elementally when any operand is parallel.

```
double acos(double x) elemental;
double asin(double x) elemental;
double atan(double x) elemental;
double atan2(double y, double x) elemental;
double cos(double x) elemental;
double sin(double x) elemental;
double tan(double x) elemental;
double cosh(double x) elemental;
double sinh(double x) elemental;
double tanh(double x) elemental;
double exp(double x) elemental;
double frexp(double value, int *exp) elemental;
double ldexp(double x, int exp) elemental;
double log(double x) elemental;
double log10(double x) elemental;
double modf(double value, double *iptr) elemental;
double pow(double x, double y) elemental;
double sqrt(double x) elemental;
double ceil(double x) elemental;
double fabs(double x) elemental;
double floor(double x) elemental;
double fmod(double x, double y) elemental;
```

4.14.3 DPCE string handling [NEW]

The following functions from **<string.h>** are redeclared here as elemental functions. The intent is that they will behave as they are described in §7.10 of [2] when operating on nonparallel operands, and will behave elementally when any operand is parallel.

```
void *memcpy(void *s1, const void *s2, size_t n) elemental;
void *memmove(void *s1, const void *s2, size_t n) elemental;
int memcmp(const void *s1, const void *s2, size_t n) elemental;
void *memset(void *s, int c, size_t n) elemental;
```

APPENDICES

A. OTHER PROPOSED EXTENSIONS

There were numerous other extensions that the committee considered and, for the reasons cited, decided not to include at the current time. References to the text of the actual proposals, if any, are given for each category of extension in the subsections of this appendix.

A.1 PARALLEL CONTROL FLOW CONSTRUCTS

Although proposed in [13], [20], and [23] and generally thought to be useful, the group did not reach consensus on the desired semantics. Implementors may choose to revisit this issue. Meanwhile, the **where** statement provides adequate functionality.

A.2 FORALL STATEMENT

This was proposed in [26], but was not adopted because it added no new expressive power to DPCE and because it lacked performance transparency. These limitations were noted in [26] itself.

A.3 ITERATORS

Iterators are being separately pursued by another X3J11 subgroup whose working document is [30, 32]. The idea was not added to DPCE for the same reasons as the FORALL statement.

A.4 BIT DATA TYPES

Although not a formal X3J11/NCEG paper [31], a formal proposal was presented to DPCE on SRC's bit oriented extensions at its meeting in September, 1993. The proposal was not incorporated into the current DPCE proposal because of time and resource constraints.

A.5 I/O LIBRARY

Although considered desirable, a parallel I/O library was left for future extensions.

A.6 DYNAMIC LAYOUT

Dynamic layout would provide a facility for reorganizing data, e.g., between stages of a calculation. Such a feature is likely to be communication-intensive, and therefore would lack performance transparency. The user is always free to explicitly move data with the existing functionality to redistribute it.

A.7 ARRAYS AS FIRST CLASS OBJECTS

The DPCE proposal could be implemented more elegantly and robustly if C arrays were simply made first class objects. This would break enormous amounts of existing C code, though, so parallel objects were introduced as a replacement for first class arrays.

A.8 OVERLOADING

Adding a general overloading mechanism would provide a way to manage the redeclared library functions proposed here, as well as allowing user-written overloaded libraries. However, this was deemed to place an excessive burden on the implementation, and was one of the C* features that was not included in DPCE [10]. The use of elemental functions for providing both parallel and nonparallel functionality for library functions and user-written functions alleviates the immediate need for this extension.

A.9 CURRENT SHAPE

The concept of a current shape can be useful in code intended to be adaptable to various shapes at run time. However, it is more restrictive than the language needs to be. Most of its functionality was subsumed by shape `void`. Current shape is discussed in [19].

A.10 INTERMEDIATE SHAPE EQUIVALENCE TEST

The intermediate shape equivalence test in C* requires that the shapes of operands in binary or ternary operations must be derived from the same shape variable name. This name equivalence test provides a compile time means for detecting when two shapes may not be the same at run time. The test is only performed when all operands are derived from shape variables; the compiler is not required to track all possible shape assignments to determine the equivalence of shapes and, hence, may report errors where none exist. Implementations are encouraged to provide a compile-time option that would enable this test, issuing warnings where C* reports errors.

A.11 C++ COMPATIBLE PARALLEL INDEXING SYNTAX

The committee adopted numerous syntax changes to ANSI/ISO C. In many cases these syntax changes will also conflict with C++ syntax. The decision to add syntax was not capricious, but rather in each case the addition was viewed as necessary to best support programmers in the writing of DPCE programs.

For example, the **where** statement was added because the committee felt the difference between conditionalization (altering the control flow) and contextualization (deactivating positions of a shape) is significant. Therefore, providing contextualization by overloading the `if` statement was rejected as being potentially confusing to programmers. Also, the option of providing contextualization via a set of library functions to explicitly manipulate the context of a shape was rejected as being too low-level and error-prone. In particular, the use of statement syntax allows the compiler to automatically reestablish the prior context upon exit from the **where** statement.

There was much discussion about the choice of syntax for the parallel index operator. Initially the discussion centered on whether parallel indexing needed to be separated from array subscripting, and to some extent mirrored the debate on whether arrays and parallel objects needed to be supported separately. Eventually the committee concluded that the potentially different memory layout requirements of arrays and parallel objects justified separating the concepts. Therefore, the committee also separated array subscripting and parallel indexing, which aided the transparency of programs by in effect highlighting the potentially expensive parallel index operation.

However, even after these decisions were reached, questions remained as to which syntax to use for the parallel index operator. The choice of using square brackets as a prefix operator, as opposed to the use of square brackets as a postfix operator for array subscripting, reflects the committee's desire to build upon the similarity of parallel indexing and array subscripting while emphasizing the fundamental differences.

A proposal was made to better support compatibility with C++ by providing parallel indexing via the overloading of the function call syntax. The committee felt that the square bracket indexing notation would be more intuitive to the C programmer. In addition, while overloading the function call operator might seem to better support C++ implementations of the DPCE concepts for initial experimentation and evaluation, the committee noted that a C++ emulation of parallel indexing would be unable to support parallel indexing as both lvalues and rvalues. To get the correct semantics requires knowledge of how the result will be used, which is not available to the implementation of an overloaded C++ operator. Furthermore, use of function call syntax for parallel indexing would change the precedence of this operation. Finally, the committee felt that, since parallel indexing was just one of several desired syntax changes (such as the **where** statement discussed above), there was no overriding reason to take extraordinary steps to obtain C++ compatibility in this one case.

A.12 SLICING OF ARRAYS

Extending the slicing proposal to be allowed in array subscripting as well as parallel indexing was discussed, and while deemed a desirable and straight-forward extension, it was determined to be beyond the scope of the DPCE proposal.

B. EXISTING IMPLEMENTATIONS

- Pacific-Sierra Research – VAST-DPC (DPCE and C*)
- Thinking Machines Corporation – CM-1/CM-2 C* (SIMD) and CM-5 C* (MIMD)
- University of New Hampshire – UNH C* (distributed-memory MIMD and shared-memory MIMD; DPCE extensions)

INDEX

<dpce.h> 6, 7, 9, 92
<math.h> 92, 104
<stdlib.h> 92
<string.h> 92, 104
abs 92
acos 104
active position 4, 14, 19, 20, 22, 78, 83, 84, 85, 102
additive operators 48
address operator 30
arithmetic conversions 11
array declarators 70
array of parallel 8, 12, 16, 30, 64, 70
array subscripting 16, 70, 106, 107
  nonparallel subscript 16
  parallel subscript 16, 30
arrays as first class objects 105
asin 104
assignment operators 59
  compound assignment 61
  simple assignment 60
atan 104
atan2 104
atoi 92
atol 92
bit data type 105
bitwise AND operator 55
bitwise exclusive OR operator 55
bitwise inclusive OR operator 56
bitwise shift operators 51
block 7, 71, 95
C++ compatibility 106
calloc 93
case label 77
cast operators 45
ceil 104
comma operator 63
compatible types 9
composite types 9
conditional operator 57
conditionalization 106
constant expressions 63, 64, 71, 76
context 4
contextualization 6, 14, 19, 23, 40, 56, 57, 82, 106
contextualization statement 78
conversions 11
cos 104
cosh 104
current shape 106
data parallel model 6
declarators 67
dimension 4
dimof 13, 36, 38, 42, 94
DPCE_BLOCK 92
dpce_layout_specs_t 92
dpce_layout_t 92, 95, 96
DPCE_SCALE 92
element 4
elemental 7, 67
elemental execution 4
elemental function 18, 20, 68, 70, 78, 85, 92, 104, 106
elemental pointer 69
elementally assignable 18
equality operators 54
everywhere 7, 82
everywhere statement 84
exp 104
fabs 104
floor 104
fmod 104
for statement 77
frexp 104
function calls 18
function declarators 70
function definitions 85
function types
  compatible function types 18
  composite function types 9, 92, 104
  elemental (see elemental function)
  nodal (see nodal function)
I/O library 105
if statement 77, 106
inactive position 14, 22, 23, 83
indirection operator 30
initialization 76
integral promotions 11
intermediate shape equivalence test 106
iteration statements 77

iterators 105
jump statements 78
layout 4
  dynamic 105
  physical 6
layoutof 13, 95
ldexp 104
log 104
log10 104
logical AND operator 56
logical OR operator 57
malloc 35, 86, 93
memcmp 104
memcpy 104
memmove 104
memset 10, 104
modf 104
modulus operator 47
multiplicative operators 47
multithreaded 6, 22, 80
newshape 96
nodal 7, 67
nodal execution environment 22
nodal function 18, 22, 70, 71, 80, 85, 88
node 4, 6, 22, 97, 98
nodeof 97, 98
nodepositionsof 91, 98
null pointer constant 12, 54
null pointer to parallel 12
null pointer-to-parallel constant 12, 54
overloading 106
palloc 36, 93, 99, 101
parallel arguments 18, 19, 22, 85
parallel constants 12, 63
parallel control flow 105, 106
parallel indexing 4, 30, 36, 85, 93, 106, 107
  nonparallel index 37
  parallel index 38
parallel lvalue 12
parallel object 4, 7
parallel operand 4, 13
parallel parameters 19, 70
parallel pointer 4, 8, 12, 30, 32, 45, 48, 49, 52, 54, 68
parallel reduction assignment 62
parallel return types 19
parallel types 8
  arrays (see array of parallel)
  compatible parallel types 9
  generic parallel type 64
  parallel arithmetic types 8
  parallel floating types 8
  parallel incomplete types 7
  parallel integral types 8
  parallel object types 7
  parallel structures and unions 8
  pointers (see parallel pointer and pointer to parallel)
parallel value 4
parallel-indexed expression 37, 38, 40
pcoord 15, 39, 40, 63, 87, 90, 100
pfree 93, 101
physical 64, 92
physical shape 4, 6, 7, 18, 22, 23, 80, 92, 98
pointer declarators 68
pointer to parallel 4, 8, 12, 16, 30, 32, 45, 48, 49, 52, 54, 68
position 5
position-oriented 16, 23, 32, 85, 89, 93
positionsof 22, 101
postfix decrement operator 27
postfix increment operator 27
postfix operators 16
pow 104
prand 102
prefix decrement operator 29
prefix increment operator 29
primary expressions 15
promotions 11, 14, 22, 38
psrand 102
qsort 93
rank 5
rankof 13, 42, 103
realloc 93
reduction 5
reduction operators 44, 61, 85
relational operators 52
replication 14, 22, 30, 38, 45, 60, 63, 76
return statement 78
same shape 38
scalar reduction assignment 62

scale 7, 71, 95
shape 7, 66
shape declarators 71
shape objects 7, 72
    fully assigned 13, 60, 72, 74, 95, 97, 98, 99, 100, 101, 102, 103
    fully unassigned 60, 72, 94
    partially assigned 60, 72, 94, 103
shape types 5, 7
    compatible shape types 9, 13, 36
    composite shape types 9
    fully specified 7, 9, 72
    fully unspecified 7, 9, 72
    generic 7, 9, 19, 60, 64, 70, 106
    partially specified 7, 9, 72
    void 7, 9, 19, 60, 64, 70, 106
shapeof 7, 13, 19, 28, 36, 54, 65
sin 104
single-node environment 22, 88
single-threaded 80
sinh 104
sizeof 35
sliced expression 15, 36, 37, 41
slicing of arrays 107
SPMD 22
sqrt 104
statements 77
stride expression 41
structure members 26
switch statement 77
tan 104
tanh 104
type names 75
type specifiers 66
unary arithmetic operators 34
unary operators 28
unary reduction operators 44
union members 26
usual arithmetic conversions 11
void pointer 12
void:void pointer 8, 12, 64
where 7, 19, 82
where statement 83, 105, 106

Chapter 8

Extended Integers for C

John W Kwan
Hewlett-Packard Company
Cupertino, California
+1 408 447 6442
email : jkwan@cup.hp.com

Acknowledgement

The idea of `<inttypes.h>` grew out of discussions in an industry group originally formed to address the problem of mapping the `int` data type in 64-bit machines. Many people participated in that group and contributed ideas for such a header, including representatives from Amdahl, Apple, DEC, Hal Computer, HP, IBM, SGI/Mips, Sun and SPARC International. Additional ideas came from members of the Numerical C Extension Group (NCEG) of X3J11 and other members of X3J11. In particular, Ron Guilmette, Tom MacDonald, Frank Farance, Rex Jaeschke, David Keaton, Jim Thomas, Fred Tydeman, Jeffery Zeeb and Jonathan Ziebell reviewed earlier versions of this paper and provided valuable suggestions. Special thanks to Randy Meyers of Digital Equipment Corporation who provided the macros for the formatted I/O specification and who collaborated with me on an earlier version of this paper.

1. Introduction

The C Standard specifies that the language should support four, signed and unsigned, integer data types, `char`, `short`, `int` and `long`. However, the Standard places very little requirement on their size (number of bits) other than that `int` and `short` be at least 16-bits and `long` be at least as long as `int` and not smaller than 32-bits. Traditionally (i.e. under Kernighan and Ritchie), C had always assumed that `int` is the most efficient (i.e. the fastest) integer data type on a machine, and Standard C, with its integral promotion rule, tacitly continues this assumption. For 16-bit systems, most implementations assign 8, 16, 16 and 32 bits to `char`, `short`, `int`, and `long`, respectively. For 32-bit systems, the common practice is to assign 8, 16, 32 and 32 bits to these types. This difference in `int` size can create some interesting problems for users who migrate from one system to another which assigns different sizes to integral types, because Standard C's integral promotion rule can produce silent changes unexpectedly.

Consider the following example :

```
main()
{
      long L = -1;
      unsigned int i = 1;
      if (L > i)
            printf ("L greater than i\n") ;
      else
            printf ("L not greater than i\n") ;
}
```

Under the Standard's promotion rule, this program will print "L greater than i" if size of `int` equals size of `long`; but it will print "L not greater than i" if size of `int` is less than size of `long`. Both results are conforming and correct. Hence the size of the `int` data type is significant in any C implementation. To complicate matters further, the need for an integer type larger than 32 bits arises for those 32-bit systems that support large files. Implementors that feel such a need have introduced a larger integer type commonly referred to as `long long`.

The need for a general solution for the "extended integer" problem has increased with the introduction of 64-bit based systems in the industry. Efforts to find a common `int` size on 64-bit systems have turned out to be much more difficult than expected.

First of all, any change in the size of `int` from the current definition will produce incompatibility; and no mapping of the base integer types to a particular range of values produces satisfactory performance in all systems. A data model that is optimal for one architecture is usually sub-optimal for another. After much discussion, the industry remains divided. However, the current system of different `int` sizes on different platforms makes life difficult for software developers who, as a result, must maintain different source for different machines (usually by using `#ifdef`). This is not very desirable. Providing the means for users to write portable code is a must if Standard C is to become "the programming language of choice."

This specification addresses this very issue. To help software developers write portable code, implementations should provide a set of integer types whose definitions are consistent across machines and independent of operating systems and other implementation idiosyncrasies. This can be provided through a new header called `<inttypes.h>`. This header defines, via `typedef`, integer types of various sizes; implementations are free to `typedef` them to integer types, including well defined extensions, that they support.

2. <inttypes.h>

```
/********************** Basic integer types ****************************
**
** The following defines the basic fixed-size integer types.
**
** Implementations are free to typedef them to Standard C integer types or
** extensions that they support. If an implementation does not support one
** of the particular integer data types below, then it should not define the
** typedefs and macros corresponding to that data type.
**
** intmax_t and uintmax_t are to be the longest (in number of bits) signed
** and unsigned integer types supported by the implementation.
**
** intptr_t and uintptr_t are signed and unsigned integer types large enough
** to hold any data pointer; that is, data pointers can be assigned into or
** from these integer types without losing precision.
**
** intfast_t and uintfast_t are the most efficient signed and unsigned
** integer types of the implementation. These shall be at least 16-bits
** long.
**
*/

typedef ? int8_t;        /* 8-bit signed integer */
typedef ? int16_t;       /* 16-bit signed integer */
typedef ? int32_t;       /* 32-bit signed integer */
typedef ? int64_t;       /* 64-bit signed integer */

typedef ? uint8_t;       /* 8-bit unsigned integer */
typedef ? uint16_t;      /* 16-bit unsigned integer */
typedef ? uint32_t;      /* 32-bit unsigned integer */
typedef ? uint64_t;      /* 64-bit unsigned integer */

typedef ? intmax_t;      /* largest signed integer supported */
typedef ? uintmax_t;     /* largest unsigned integer supported */

typedef ? intptr_t;      /* signed integer type capable of holding a void * */
typedef ? uintptr_t      /* unsigned integer type capable of holding a void * */

typedef ? intfast_t;     /* most efficient signed integer type */
typedef ? uintfast_t     /* most efficient unsigned integer type */
```

```
/********************** Extended integer types *************************
**
** The following define the smallest integer types that can hold the
** specified number of bits.
**
** Implementations are free to typedef them to Standard C integer types or
** any supported extensions.
*/

/* smallest signed integer of at least 8 bits */
typedef ? int_least8_t;

/* the most efficient signed integer of at least 8 bits */
typedef ? int_fast8_t;

/* smallest signed integer of at least 16 bits */
typedef ? int_least16_t;

/* the most efficient signed integer of at least 16 bits */
typedef ? int_fast16_t;

/* smallest signed integer of at least 32 bits */
typedef ? int_least32_t;

/* the most efficient signed integer of at least 32 bits */
typedef ? int_fast32_t;

/* smallest signed integer of at least 64 bits */
typedef ? int_least64_t;

/* the most efficient signed integer of at least 64 bits */
typedef ? int_fast64_t;

/* smallest unsigned integer of at least 8 bits */
typedef ? uint_least8_t;

/* the most efficient unsigned integer of at least 8 bits */
typedef ? uint_fast8_t;

/* smallest unsigned integer of at least 16 bits */
typedef ? uint_least16_t;

/* the most efficient unsigned integer of at least 16 bits */
typedef ? uint_fast16_t;

/* smallest unsigned integer of at least 32 bits */
typedef ? uint_least32_t;

/* the most efficient unsigned integer of at least 32 bits */
typedef ? uint_fast32_t;

/* smallest unsigned integer of at least 64 bits */
typedef ? uint_least64_t;
```

```
/* the most efficient unsigned integer of at least 64 bits */
typedef ? uint_fast64_t;


/* ************************** limits ********************************
**
** The following defines the limits for the above types.
**
** INTMAX_MIN (minimum value of the largest supported signed integer type),
** INTMAX_MAX (maximum value of the largest supported signed integer type),
** and UINTMAX_MAX (maximum value of the largest supported unsigned integer
** type) can be set to implementation defined limits.
**
** NOTE : A programmer can test to see whether an implementation supports
** a particular size of integer by testing if the macro that gives the
** maximum for that datatype is defined. For example, if #ifdef UINT64_MAX
** tests false, the implementation does not support unsigned 64 bit integers.
**
** The values used below are merely examples. Actual limits for
** an implementation may vary.
**
** The type of these macros is intentionally unspecified.
**
*/

#define INT8_MIN  (-128)
#define INT16_MIN (-32767-1)
#define INT32_MIN (-2147483647-1)
#define INT64_MIN (-9223372036854775807-1)

#define INT_LEAST8_MIN ?   /* implementation defined */
#define INT_LEAST16_MIN ?  /* implementation defined */
#define INT_LEAST32_MIN ?  /* implementation defined */
#define INT_LEAST64_MIN ?  /* implementation defined */

#define INT_FAST8_MIN ?    /* implmentation defined */
#define INT_FAST16_MIN ?   /* implmentation defined */
#define INT_FAST32_MIN ?   /* implmentation defined */
#define INT_FAST64_MIN ?   /* implmentation defined */

#define INT8_MAX  (127)
#define INT16_MAX (32767)
#define INT32_MAX (2147483647)
#define INT64_MAX (9223372036854775807)

#define UINT8_MAX  (255U)
#define UINT16_MAX (65535U)
#define UINT32_MAX (4294967295U)
#define UINT64_MAX (18446744073709551615U)

#define INTMAX_MIN ?    /* implementation defined */
#define INTMAX_MAX ?    /* implementation defined */
#define UINTMAX_MAX ?   /* implementation defined */
```

```
#define INTFAST_MIN ?   /* implementation defined */
#define INTFAST_MAX ?   /* implementation defined */
#define UINTFAST_MAX ?  /* implementation defined */


#define INT_LEAST8_MAX ?    /* implementation defined */
#define INT_LEAST16_MAX ?   /* implementation defined */
#define INT_LEAST32_MAX ?   /* implementation defined */
#define INT_LEAST64_MAX ?   /* implementation defined */

#define INT_FAST8_MAX ?     /* implementation defined */
#define INT_FAST16_MAX ?    /* implementation defined */
#define INT_FAST32_MAX ?    /* implementation defined */
#define INT_FAST64_MAX ?    /* implementation defined */


/* The following 2 macros are provided for testing whether the types
** intptr_t and uintptr_t (integers large enough to hold a void *) are
** defined in this header. They are needed in case the architecture can't
** represent a pointer in any standard integral type.
*/
#define INTPTR_MAX
#define UINTPTR_MAX



/* ************************ CONSTANTS ********************************
**
** The following macros create constants of the above types. The intent is
** that:
**    Constants defined using these macros have a specific size and
**    signedness. The suffix used for int64_t and uint64_t (ll and ull)
**    are for examples only. Implementations are permitted to use other
**    suffixes.
*/

#define __CONCAT__(A,B) A ## B

#define INT8_C(c)         (c)
#define INT16_C(c)        (c)
#define INT32_C(c)        (c)
#define INT64_C(c)        __CONCAT__(c,ll)

#define UINT8_C(c)        __CONCAT__(c,u)
#define UINT16_C(c)       __CONCAT__(c,u)
#define UINT32_C(c)       __CONCAT__(c,u)
#define UINT64_C(c)       __CONCAT__(c,ull)

#define INTMAX_C(c)       __CONCAT__(c,ll)
#define UINTMAX_C(c)      __CONCAT__(c,ull)
```

```
/************************ FORMATTED I/O *******************************
**
** The following macros can be used even when an implementation has not
** extended the printf/scanf family of functions.
**
** The form of the names of the macros is either "PRI" for printf specifiers
** or "SCN" for scanf specifiers, followed by the conversion specifier letter
** followed by the datatype size. For example, PRId32 is the macro for
** the printf d conversion specifier with the flags for 32 bit datatype.
**
** Separate macros are given for printf and scanf because typically different
** size flags must prefix the conversion specifier letter.
**
*** An example using one of these macros:
**
**    uint64_t u;
**    printf("u = %016" PRIx64 "\n", u);
**
** For the purpose of example, the definitions of the printf/scanf macros
** below have the values appropriate for a machine with 16 bit shorts,
** 32 bit ints, and 64 bit longs.
**
*/

/* printf macros for signed integers */
#define PRId8           "d"
#define PRId16          "d"
#define PRId32          "d"
#define PRId64          "ld"

#define PRIdLEAST8      "d"
#define PRIdLEAST16     "d"
#define PRIdLEAST32     "d"
#define PRIdLEAST64     "ld"

#define PRIdFAST8       "d"
#define PRIdFAST16      "d"
#define PRIdFAST32      "d"
#define PRIdFAST64      "ld"

#define PRIi8           "i"
#define PRIi16          "i"
#define PRIi32          "i"
#define PRIi64          "li"

#define PRIiLEAST8      "i"
#define PRIiLEAST16     "i"
#define PRIiLEAST32     "i"
#define PRIiLEAST64     "li"

#define PRIiFAST8       "i"
#define PRIiFAST16      "i"
#define PRIiFAST32      "i"
```

```
#define PRIiFAST64      "li"

#
#define PRIo8           "o"
#define PRIo16          "o"
#define PRIo32          "o"
#define PRIo64          "lo"

#define PRIoLEAST8      "o"
#define PRIoLEAST16     "o"
#define PRIoLEAST32     "o"
#define PRIoLEAST64     "lo"

#define PRIoFAST8       "o"
#define PRIoFAST16      "o"
#define PRIoFAST32      "o"
#define PRIoFAST64      "lo"

#define PRIx8           "x"
#define PRIx16          "x"
#define PRIx32          "x"
#define PRIx64          "lx"

#define PRIxLEAST8      "x"
#define PRIxLEAST16     "x"
#define PRIxLEAST32     "x"
#define PRIxLEAST64     "lx"

#define PRIxFAST8       "x"
#define PRIxFAST16      "x"
#define PRIxFAST32      "x"
#define PRIxFAST64      "lx"

#define PRIX8           "X"
#define PRIX16          "X"
#define PRIX32          "X"
#define PRIX64          "lX"

#define PRIXLEAST8      "X"
#define PRIXLEST16      "X"
#define PRIXLEST32      "X"
#define PRIXLEST64      "lX"

#define PRIXFAST8       "X"
#define PRIXFAST16      "X"
#define PRIXFAST32      "X"
#define PRIXFAST64      "lX"

/* printf macros for unsigned integers */
#define PRIu8           "u"
#define PRIu16          "u"
#define PRIu32          "u"
#define PRIu64          "lu"
```

```
#define PRIuLEAST8      "u"
#define PRIuLEAST16     "u"
#define PRIuLEAST32     "u"
#define PRIuLEAST64     "lu"

#define PRIuFAST8       "u"
#define PRIuFAST16      "u"
#define PRIuFAST32      "u"
#define PRIuFAST64      "lu"

/* scanf macros */
#define SCNd16          "hd"
#define SCNd32          "d"
#define SCNd64          "ld"

#define SCNi16          "hi"
#define SCNi32          "i"
#define SCNi64          "li"

#define SCNo16          "ho"
#define SCNo32          "o"
#define SCNo64          "lo"

#define SCNu16          "hu"
#define SCNu32          "u"
#define SCNu64          "lu"

#define SCNx16          "hx"
#define SCNx32          "x"
#define SCNx64          "lx"

/* The following macros define I/O formats for intmax_t and uintmax_t. */

#define PRIdMAX  ?      /* implementation defined */
#define PRIoMAX  ?      /* implementation defined */
#define PRIxMAX  ?      /* implementation defined */
#define PRIuMAX  ?      /* implementation defined */

#define SCNiMAX  ?      /* implementation defined */
#define SCNdMAX  ?      /* implementation defined */
#define SCNoMAX  ?      /* implementation defined */
#define SCNxMAX  ?      /* implementation defined */
```

```
/* The following macros define I/O formats for intfast_t and uintfast_t. */

#define PRIdFAST  ?       /* implementation defined */
#define PRIoFAST  ?       /* implementation defined */
#define PRIxFAST  ?       /* implementation defined */
#define PRIuFAST  ?       /* implementation defined */

#define SCNiFAST  ?       /* implementation defined */
#define SCNdFAST  ?       /* implementation defined */
#define SCNoFAST  ?       /* implementation defined */
#define SCNxFAST  ?       /* implementation defined */


/*************** conversion functions ********************************
**
** The following routines convert from strings to the largest supported
** integer types. They parallel the strtol and strtoul functions in Standard
** C. Implementations are free to equate them to any existing functions
** they may have.
*/

extern intmax_t strtoimax (const char *, char**, int);
extern uintmax_t strtoumax (const char *, char**, int);


/* end of inttypes.h */
```

3. Notes

The intent of this paper is to take a "minimalist" approach to solving the extended integer problem in C; one that accommodates, but does not necessarily requires, extensions to the language. Internal representation of a data type (e.g. endianess, bit and byte ordering) is outside the scope of this specification.

Besides defining integer data types of 8, 16, 32 and 64 bits, this header also defines integer types of at least 8, 16, 32 and 64 bits. This is mainly for systems whose word size does not fit the 16-bit or 32-bit word model. Implementations do not have to support all data types defined in this specification.

The various MIN/MAX macros define the limits of each data type. By using the `#ifdef` directive, users can test to see if certain data types are supported in an implementation. For example, if `#ifdef` INT64_MAX tests false, this header does not define 64-bit integers.

The __CONCAT__ macro provides a means to construct constants of a particular type.

The types `intfast_t` and `uintfast_t` define the most efficient signed and unsigned integer type of an implementation. They shall be at least 16-bits long. For those systems that have more than one "most efficient" integer types, the largest one should be used.

This paper also presents a method to handle formatted I/O that requires little or no extension to the language by providing macros for existing formats like `d`, `i`, `o` and `x`. There are no macros corresponding to the `c` conversion specifier. Most implementations do not support the `c` conversion specifier for anything besides `int`. Likewise, there are no `scanf` macros for the 8-bit datatypes as most implementations do not support reading a `char` of any size.