ISO/IEC JTC1 SC22 WG14 N1312

Date: 2008-05-16

Reference number of document: **ISO/IEC TR 24732**

Committee identification: ISO/IEC JTC1 SC22 WG14

SC22 Secretariat: ANSI

Information Technology —

Programming languages, their environments and system software interfaces —

Extension for the programming language C to support decimal floating-point arithmetic —

Warning

This document is an ISO/IEC draft Technical Report. It is not an ISO/IEC International Technical Report. It is distributed for review and comment. It is subject to change without notice and shall not be referred to as an International Technical Report or International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Report Type 2
Document subtype: n/a
Document stage: (3) Proposed Draft Technical Report
Document language: E

Contents

1 Introduction

1.1 Background

Most of today's general purpose computing architectures provide binary floating-point arithmetic in hardware. Binary floating-point is an efficient representation which minimizes memory use, and is simpler to implement than floating-point arithmetic using other bases. It has therefore become the norm for scientific computations, with almost all implementations following the IEEE 754 standard for binary floating-point arithmetic.

However, human computation and communication of numeric values almost always uses decimal arithmetic and decimal notations. Laboratory notes, scientific papers, legal documents, business reports and financial statements all record numeric values in decimal form. When numeric data are given to a program or are displayed to a user, binary to-and-from decimal conversion is required. There are inherent rounding errors involved in such conversions; decimal fractions cannot, in general, be represented exactly by binary floating-point values. These errors often cause usability and efficiency problems, depending on the application.

These problems are minor when the application domain accepts, or requires results to have, associated error estimates (as is the case with scientific applications). However, in business and financial applications, computations are either required to be exact (with no rounding errors) unless explicitly rounded, or be supported by detailed analyses that are auditable to be correct. Such applications therefore have to take special care in handling any rounding errors introduced by the computations.

The most efficient way to avoid conversion error is to use decimal arithmetic. Currently, the IBM zArchitecture (and its predecessors since System/360) is a widely used system that supports built-in decimal arithmetic. This, however, provides integer arithmetic only, meaning that every number and computation has to have separate scale information preserved and computed in order to maintain the required precision and value range. Such scaling is difficult to code and is error-prone; it affects execution time significantly, and the resulting program is often difficult to maintain and enhance.

Even though the hardware may not provide decimal arithmetic operations, the support can still be emulated by software. Programming languages used for business applications either have native decimal types (such as PL/I, COBOL, C#, or Visual Basic) or provide decimal arithmetic libraries (such as the BigDecimal class in Java). The arithmetic used in business applications, nowadays, is almost invariably decimal floating-point; the COBOL 2002 ISO standard, for example, requires that all standard decimal arithmetic calculations use 32-digit decimal floating-point.

Arguably, the C language hits a sweet spot within the wide range of programming languages available today – it strikes an optimal balance between usability and performance. Its simple and expressive syntax makes it easy to program; and its close-to-the-hardware semantics makes it efficient. Despite the advent of newer programming languages, C is still often used together with other languages to code the computationally intensive part of an application. In many cases, entire

business applications are written in C/C++. To maintain the vitality of C, the need for decimal arithmetic by the business and financial community cannot be ignored.

The importance of this has been recognized by the IEEE. The IEEE 754 standard is currently being revised, and the major change in that revision is the addition of decimal floating-point formats and arithmetic.

Historically there has been a close tie between IEEE 754 and C with respect to floating-point specification. This Technical Report proposes to add decimal floating types and arithmetic to the C programming language specification.

1.2 The Arithmetic Model

This Technical Report proposes to add support for the decimal formats for floating-point data specified in IEEE 754-2008, with operations and behaviors consistent with that specification. IEEE 754-2008 provides a unified specification for floating-point arithmetic using both binary radix and decimal radix representations. For binary radix, it specifies upwardly-compatible extensions to the previous version, IEEE 754-1985 (equivalently IEC 60559:1989, which is already supported by C99 implementations that define the macro **__STDC_IEC_559__**). Those extensions are not considered in this proposal. Instead, this proposal confines itself to supporting the decimal radix formats, which are new in this revision of IEEE 754.

The model of floating-point arithmetic used in IEEE 754-2008 has three components:

- *data* - numbers and NaNs, which can be manipulated by, or be the results of, the operations it specifies
- *operations* - (addition, multiplication, conversions, etc) which can be carried out on data
- *context* - the status of operations (namely, exceptions flags), and controls to govern the results of operations (for example, rounding modes). (IEEE 754-2008 does not use a single term to refer to these collectively.)

The model defines these components in the abstract. It neither defines the way in which operations are expressed (which might vary depending on the computer language or other interface being used), nor does it define the concrete representation (specific layout in storage, or in a processor's register, for example) of data or context, except that it does define specific encodings that are to be used for data that may be exchanged between different implementations that conform to the specification.

From the perspective of the C language, *data* are represented by data types, *operations* are defined within expressions, and *context* is the floating environment specified in **<fenv.h>**. This Technical Report specifies how the C language implements these components.

1.3 The Formats

IEEE 754-2008 specifies *formats*, in terms of their radix, exponent range, and precision (significand length), to support general purpose decimal floating-point arithmetic. It specifies operation semantics in terms of values and abstract representations of data (format members). It also specifies bit-level encodings for formats intended for data interchange.

C99 specifies floating-point arithmetic using a two-layer organization. The first layer provides a specification using an abstract model. The representation of a floating-point number is specified in an abstract form where the constituent components of the representation are defined (sign, exponent, significand) but not the internals of these components. In particular, the exponent range, significand size, and the base (or radix) are implementation defined. This allows flexibility for an implementation to take advantage of its underlying hardware architecture. Furthermore, certain behaviors of operations are also implementation defined, for example in the area of handling of special numbers and in exceptions.

The reason for this approach is historical. At the time when C was first standardized, there were already various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would make most of the existing implementations at the time not conforming.

C99 provides a binding to IEEE 754 by specifying an Annex F, *IEC 60559 floating point arithmetic*, and adopting that standard by reference. An implementation may choose not to conform to IEEE 754 and indicates that by not defining the macro __STDC_IEC_559__.This means not all implementations need to support IEEE 754, and the floating-point arithmetic need not be binary.

This Technical Report specifies decimal floating-point arithmetic according to IEEE 754-2008, with the constituent components of the representation defined. This is more stringent than the existing C99 approach for the floating types. Since it is expected that all decimal floating-point hardware implementations will conform to the revised IEEE 754, binding to this standard directly benefits both implementers and programmers.

2 General

2.1 Scope

This Technical Report specifies an extension to the programming language C, specified by the international standard ISO/IEC 9899:1999. The extension provides support for decimal floating-point arithmetic that is intended to be consistent with the specification in IEEE 754-2008. However, as of the October 4, 2006 IEEE draft, the referenced standard is still in draft review stage. Any conflict between the requirements described here and the referenced standard is unintentional. This Technical Report defers to IEEE 754-2008.

The binary floating-point arithmetic as specified in IEEE 754-2008 is not considered in this Technical Report.

2.2 References

The following standards contain provisions which, through reference in this text, constitute provisions of this Technical Report. For dated references, subsequent amendment to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred applies. Members of IEC and ISO maintain registers of current valid International Standards.

ISO/IEC 9899:1999, *Information technology - Programming languages, their environments and system software interfaces - Programming Language C*.

ISO/IEC 9899:1999/Cor 1:2001, *Information technology - Programming languages, their environments and system software interfaces - Programming Language C – Technical Corrigendum 1*.

ISO/IEC 9899:1999/Cor 2:2004, *Information technology - Programming languages, their environments and system software interfaces - Programming Language C – Technical Corrigendum 2*.

ISO/IEC TR 18037, *Information technology - Programming languages, their environments and system software interfaces – Extensions for the programming language C to support embedded processors.*

ISO/IEC 1989:2002, *Information technology - Programming languages - COBOL*.

IEC 60559:1989, *Binary floating-point arithmetic for microprocessors systems* (previously designated IEC 559:1989).

ANSI X3.274, *Information Technology - Programming Language REXX*.

ANSI/IEEE 754-1985 - *IEEE Standard for Binary Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 1985.

ANSI/IEEE 854-1987 - *IEEE Standard for Radix-Independent Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 1987.

The IEEE 754 revision working group is currently revising the specification for floating-point arithmetic:

ANSI/IEEE 754-2008 - *IEEE Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc. Draft.

A Decimal Floating-Point Specification, Schwarz, Cowlishaw, Smith, and Webb, in the *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (Arith 15)*, IEEE, June 2001.

Note: Reference materials relating to IEEE 754-2008 can be found in http://grouper.ieee.org/groups/754/ and http://www.validlab.com/754R/.

3 Predefined macro name

The following macro name is conditionally defined by the implementation:

__STDC_DEC_FP__ The integer constant **200805L**, intended to indicate conformance to this technical report.

4 Decimal floating types

This Technical Report introduces three decimal floating types, designated as **_Decimal32**, **_Decimal64** and **_Decimal128**. The set of values of type **_Decimal32** is a subset of the set of values of the type **_Decimal64**; the set of values of the type **_Decimal64** is a subset of the set of values of the type **_Decimal128**.

Within the type hierarchy, decimal floating types are base types, real types and arithmetic types.

The types **float**, **double**, and **long double** are also called generic floating types for the purpose of this Technical Report.

Note: C does not specify a radix for **float**, **double** and **long double**. An implementation can choose the representation of **float**, **double** and **long double** to be the same as the decimal floating types. In any case, the decimal floating types are distinct from **float**, **double** and **long double** regardless of the representation.

Note: This Technical Report does not define decimal complex types or decimal imaginary types. The three complex types remain as **float _Complex**, **double _Complex** and **long double _Complex**, and the three imaginary types remain as **float _Imaginary**, **double _Imaginary** and **long double _Imaginary**.

Suggested changes to C99:

Change the first sentence of 6.2.5#10:

[10] There are three *generic floating types*, designated as **float**, **double** and **long double**.

Add the following paragraphs after 6.2.5#10:

[10a] There are three *decimal floating types*, designated as **_Decimal32**, **_Decimal64** and **_Decimal128**. The set of values of the type **_Decimal32**[1] is a subset of the set of values of the type **_Decimal64**; the set of values of the type **_Decimal64** is a subset of the set of values of the type **_Decimal128**. Decimal floating types are real floating types.

[10b] Together, the generic floating types and the decimal floating types comprise the *real floating types*.

Add the following to 6.7.2 Type specifiers:

type-specifier:
_Decimal32
_Decimal64
_Decimal128

Add the following paragraph after 6.5#8:

[8a] Expressions involving decimal floating-point operands are evaluated according to the semantics of IEEE 754-2008, including production of results with the preferred exponent as specified in IEEE 754-2008.

5 Characteristics of decimal floating types <float.h>

The characteristics of decimal floating types are defined in terms of a model specifying general decimal arithmetic (1.2). The formats are specified in IEEE 754-2008 (1.3).

The three decimal formats defined in IEEE 754-2008 correspond to the three decimal floating types as follows:

- **_Decimal32** is a *decimal32* number, which is encoded in four consecutive octets (32 bits)
- **_Decimal64** is a *decimal64* number, which is encoded in eight consecutive octets (64 bits)
- **_Decimal128** is a *decimal128* number, which is encoded in 16 consecutive octets (128 bits)

The value of a finite number is given by $(-1)^{sign}$ x coefficient x $10^{exponent}$. Refer to IEEE 754-2008 for details of the format.

These formats are characterized by the length of the coefficient, and the maximum and minimum exponent. The coefficient is not normalized, so trailing zeros are significant; i.e., 1.0 is equal to but can be distinguished from 1.00. The table below shows these characteristics by format:

[1] The 32-bit format is a storage only format in IEEE 754-2008.

Format	**_Decimal32**	**_Decimal64**	**_Decimal128**
Coefficient length in digits	7	16	34
Maximum Exponent (E_{max})	97	385	6145
Minimum Exponent (E_{min})	-94	-382	-6142

If the macro **__STDC_WANT_DEC_FP__** is defined at the point in the source file where the header **<float.h>** is included, the header **<float.h>** shall define several macros that expand to various limits and parameters of the decimal floating types. The names and meaning of these macros are similar to the corresponding macros for generic floating types.

Suggested change to C99:

Add the following after 5.2.4.2.2:

5.2.4.2.2a Characteristics of decimal floating types <float.h>

[1] Macros in **<float.h>** provide characteristics of floating types in terms of the model presented in 5.2.4.2.2. The prefixes **DEC32_**, **DEC64_**, and **DEC128_** denote the types **_Decimal32**, **_Decimal64**, and **_Decimal128** respectively.

[2] For decimal floating-point, it is often convenient to consider an alternate equivalent model where the significand is represented with integer rather than fraction digits: a floating-point number (x) is defined by the model

$$x = sb^{(e-p)} \sum_{k=1}^{p} f_k b^{(p-k)}$$

where s, b, e, p, and f_k are as defined in 5.2.4.2.2, and $b = 10$.

[3] The term *quantum exponent* refers to $q = e - p$ and *coefficient* to $c = f_1 f_2 ... f_p$, an integer between 0 and $b^p - 1$ inclusive. Thus, $x = s * c * b^q$ is represented by the triple of integers *(s, c, q)*.

[4] For binary floating-point following IEC 60559 (and IEEE 754-2008), representations in the model described in 5.2.4.2.2 that have the same numerical value are indistinguishable in the arithmetic. However, for decimal floating-point, representations that have the same numerical value but different quantum exponents, e.g., (1, 10, -1) representing 1.0 and (1, 100, -2) representing 1.00, are distinguishable. To facilitate exact fixed-point calculation, standard decimal floating-point operations and functions have a *preferred quantum exponent*, as specified in IEEE 754-2008, which is determined by the quantum exponents of the operands if they have decimal floating-point types (or by specific rules for conversions from other types), and they produce a result with that preferred quantum exponent, or as close to it as possible within the limitations of the type. For example, the preferred quantum exponent for addition is the minimum of the

quantum exponents of the operands. Hence (1, 123, - 2) + (1, 4000, -3) = (1, 5230, -3) or 1.23 + 4.000 = 5.230.

[5] Except for assignment and casts, the values of operations with decimal floating operands and values subject to the usual arithmetic conversions and of decimal floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of **DEC_EVAL_METHOD**:

-1 indeterminable;
0 evaluate all operations and constants just to the range and precision of the type;
1 evaluate operations and constants of type **_Decimal32** and **_Decimal64** to the range and precision of the **_Decimal64** type, evaluate **_Decimal128** operations and constants to the range and precision of the **_Decimal128** type;
2 evaluate all operations and constants to the range and precision of the **_Decimal128** type.

[6] The integer values given in the following lists shall be replaced by constant expressions suitable for use in **#if** preprocessing directives:

- radix of exponent representation, *b(=10)*

  For the generic floating-point types, this value is implementation-defined and is specified by the macro **FLT_RADIX**. For the decimal floating-point types there is no corresponding macro, since the value 10 is an inherent property of the types. Wherever **FLT_RADIX** appears in a description of a function that has versions that operate on decimal floating-point types, it is noted that for the decimal floating-point versions the value used is implicitly 10, rather than **FLT_RADIX**.

- number of digits in the coefficient

  DEC32_MANT_DIG 7
  DEC64_MANT_DIG 16
  DEC128_MANT_DIG 34

- minimum exponent

  DEC32_MIN_EXP -94
  DEC64_MIN_EXP -382
  DEC128_MIN_EXP -6142

- maximum exponent

  DEC32_MAX_EXP 97
  DEC64_MAX_EXP 385
  DEC128_MAX_EXP 6145

- maximum representable finite decimal floating number (there are 6, 15 and 33 9's after the decimal points respectively)

```
DEC32_MAX      9.999999E96DF
DEC64_MAX      9.999999999999999E384DD
DEC128_MAX     9.999999999999999999999999999999999E6144DL
```

- the difference between 1 and the least value greater than 1 that is representable in the given floating point type

```
DEC32_EPSILON  1E-6DF
DEC64_EPSILON  1E-15DD
DEC128_EPSILON 1E-33DL
```

- minimum normalized positive decimal floating number

```
DEC32_MIN      1E-95DF
DEC64_MIN      1E-383DD
DEC128_MIN     1E-6143DL
```

- minimum positive subnormal decimal floating number

```
DEC32_SUBNORMAL_MIN      0.000001E-95DF
DEC62_SUBNORMAL_MIN      0.000000000000001E-383DD
DEC128_SUBNORMAL_MIN
     0.00000000000000000000000000000001E-6143DL
```

6 Conversions

6.1 Conversions between decimal floating and integer

For conversions between real floating and integer types, C99 6.3.1.4 leaves the behavior undefined if the conversion result cannot be represented (Annex F.4 tightened up the behavior.) To help writing portable code, this Technical Report provides defined behavior for decimal floating type. Furthermore, it is useful to allow program execution to continue without interruption unless the program needs to check the condition.

Suggested changes to C99:

Change the first sentence of 6.3.1.4 paragraph 1:

[1] When a finite value of generic floating type is converted to an integer type …

Add the follow paragraph after 6.3.1.4 paragraph 1:

[1a] When a finite value of decimal floating type is converted to an integer type other than **_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the "invalid" floating-point exception shall be raised and the result of the conversion is unspecified.

Change the first sentence of 6.3.1.4 paragraph 2:

[2] When a value of integer type is converted to a generic floating type, …

Add the following paragraph after 6.3.1.4 paragraph 2:

[2a] When a value of integer type is converted to a decimal floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result shall be correctly rounded with exceptions raised as specified in IEEE 754-2008.

6.2 Conversions among decimal floating types, and between decimal floating types and generic floating types

The specification is similar to the existing ones for **float**, **double** and **long double**, except that when the result cannot be represented exactly, the behavior is tightened to become correctly rounded.

Suggested change to C99:

Add after 6.3.1.5#2.

[3] When a **_Decimal32** is promoted to **_Decimal64** or **_Decimal128**, or a **_Decimal64** is promoted to **_Decimal128**, the value is converted to the type being promoted to. All extra precision and/or range (for the converted to type) are removed.

[4] When a **_Decimal64** is demoted to **_Decimal32**, a **_Decimal128** is demoted to **_Decimal64** or **_Decimal32**, or conversion is performed among decimal and generic floating types other than the above, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is correctly rounded with exceptions raised as specified in IEEE 754-2008.

6.3 Conversions between decimal floating and complex

This is covered by C99 6.3.1.7.

6.4 Usual arithmetic conversions

In an application that is written using decimal arithmetic, mixed operations between decimal and other real types are likely to occur only when interfacing with other languages, calling existing libraries written for binary floating point arithmetic, or accessing existing data. Determining the common type for mixed operations is difficult because ranges overlap; therefore, mixed mode operations are not allowed and the programmer must use explicit casts. Implicit conversions are allowed only for simple assignment, **`return`** statement, and in argument passing involving prototyped functions.

Following are suggested changes to C99:

Insert the following to 6.3.1.8#1, after "This pattern is called the *usual arithmetic conversions*:"

6.3.1.8[1]

... This pattern is called the *usual arithmetic conversions*:

If one operand is a decimal floating type, all other operands shall not be generic floating type, complex type, or imaginary type:

> First if either operand is **`_Decimal128`**, the other operand is converted to **`_Decimal128`**.
>
> Otherwise, if either operand is **`_Decimal64`**, the other operand is converted to **`_Decimal64`**.
>
> Otherwise, if either operand is **`_Decimal32`**, the other operand is converted to **`_Decimal32`**.

If there are no decimal floating types in the operands:

> First, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without ... <the rest of 6.3.1.8#1 remains the same>

6.5 Default argument promotion

There is no default argument promotion specified for the decimal floating types. Default argument promotion covered in C99 6.5.2.2 [6] and [7] remains unchanged, and applies to generic floating types only.

7 Constants

New suffixes are added to denote decimal floating constants: **DF** for **_Decimal32**, **DD** for **_Decimal64**, and **DL** for **_Decimal128**.

Suggested changes to C99:

Change *floating-suffix* in 6.4.4.2 to:

floating-suffix: one of
f d l F D L df dd dl DF DD DL

Add the following paragraph after 6.4.4.2#2:

6.4.4.2
...
[2a] **Constraints**

The *floating-suffix* **df**, **dd**, **dl**, **DF**, **DD** and **DL** shall not be used in a *hexadecimal-floating-constant*.

Change 6.4.4.2#4 to:

[4] An unsuffixed floating constant has type **double**, unless modified by the standard pragma **FLOAT_CONST_DECIMAL64**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **d** or **D**, it has type **double**. If suffixed by the letter **l** or **L**, it has type **long double**.

Add the following paragraph after 6.4.4.2#4:

6.4.4.2
...
[4a] If a floating constant is suffixed by **df** or **DF**, it has type **_Decimal32**. If suffixed by **dd** or **DD**, it has type **_Decimal64**. If suffixed by **dl** or **DL**, it has type **_Decimal128**.

Add the following paragraph after 6.4.4.2#5:

[5a] For decimal floating-point constants, representations that have the same numerical value but different quantum exponents have distinguishable internal formats. The quantum exponent is specified to be the same as strtodxx for the same representation string.

Add the following paragraph after 6.4.4.2#7:

Forward references: the **FLOAT_CONST_DECIMAL64** pragma (6.4.4.2a).

7.1 Unsuffixed floating constant

The above introduces new suffixes for the decimal floating constants. It would help usability if unsuffixed floating constant could be used. The issue can be illustrated by the following example:

```
_Decimal64 rate = 0.1;
```

The constant 0.1 has type **double**. In an implementation where binary representation is used for the floating types, the internal representation of 0.1 cannot be exact. The variable *rate* will get a value slightly different from 0.1. This defeats the purpose of decimal floating types. On the other hand, requiring programmers to write:

```
_Decimal64 rate = 0.1dd;
```

can be inconvenient and affect readability of the program.

7.1.1 The FLOAT_CONST_DECIMAL64 pragma

Source code that uses both generic and decimal floating point values in close proximity ought to use the suffixed forms of floating-point constants for clarity. However, it may be expected that a typical usage pattern would be that within significant portions of source code, all of the floating-point usage would be purely generic or purely decimal. The **FLOAT_CONST_DECIMAL64** pragma allows programmers to establish a context in which unsuffixed floating-point constants would be uniformly interpreted as having either type **double** (as they do in C99) or type **_Decimal64**.

Note that as a practical matter, especially with early implementations, this pragma could have unintended effects on floating-point constants defined in header files, as C99 does not have a suffix to specify that a constant is of type **double** explicitly; all such constants will be unsuffixed in pre-existing C99 header files. If this pragma is in the “on” state when such a header is included, unsuffixed constants it uses outside of macro definitions (e.g. in initializers or inline functions) will be given type **_Decimal64**. Similarly, if the pragma is in the “on” state when a macro defined in such a header is expanded, the constant in the macro expansion will be given type **_Decimal64**. Existing C header files containing constants of type **double** can be made safe from this kind of misinterpretation by adding an explicit **d** or **D** suffix to each such constant conditionally, when the preprocessor expression:

```
(__STDC_DEC_FP__ >= 200805L)
```

is true. But unless and until it is known that all the headers used by a program have been modified to use an explicit **d** or **D** suffix on constants of type **double**, setting this pragma to the “on” state requires extreme caution.

Suggested changes to C99:

Add the following paragraphs after 6.4.4.2:

6.4.4.2a The `FLOAT_CONST_DECIMAL64` pragma

The type given to an unsuffixed floating-point constant is normally type **`double`**. However, the following pragma may be used to change this behavior:

#pragma **`STDC FLOAT_CONST_DECIMAL64`** *on-off-switch*

This pragma directs the implementation to treat unsuffixed floating-point constants as having type **`double`** (where the state is "off") or type **`_Decimal64`** (where the state is "on"). The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **`FLOAT_CONST_DECIMAL64`** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **`FLOAT_CONST_DECIMAL64`** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is "off".

Add the following to the list of **`STDC`** pragmas in 6.10.6:

#pragma **`STDC FLOAT_CONST_DECIMAL64`** *on-off-switch*

8 Arithmetic Operations

8.1 Operators

The operators *Add* (C99 6.5.6), *Subtract* (C99 6.5.6), *Multiply* (C99 6.5.5), *Divide* (C99 6.5.5), *Relational operators* (C99 6.5.8), *Equality operators* (C99 6.5.9), *Unary Arithmetic operators* (C99 6.5.3.3), and *Compound Assignment operators* (C99 6.5.16.2) when applied to decimal floating type operands shall follow the semantics as defined in IEEE 754-2008.

Suggested changes to C99:

Add the following after 6.5.5 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall not have generic floating type, complex type, nor imaginary type.

Add the following after 6.5.6 paragraph 3:

[3a] If either operand has decimal floating type, the other operand shall not have generic floating type, complex type, nor imaginary type.

Add the following after 6.5.8 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall not have generic floating type.

Add the following after 6.5.9 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall not have generic floating type, complex type, nor imaginary type.

Add the following bullet to 6.5.15 paragraph 3:

- one operand has decimal floating type, and the other has arithmetic type other than generic floating type, complex type, or imaginary type;

Add the following after 6.5.16.2 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall not have generic floating type, complex type, nor imaginary type.

8.2 Functions

The headers and library supply a number of functions and macros that implement support for decimal floating point data with the semantics specified in IEEE 754-2008, including producing results with the preferred exponent where appropriate. That support is provided by the following:

From **<math.h>**, the decimal floating-point type versions of:
sqrt, **fma**, **fabs**, **fmax**, **fmin**, **ceil**, **floor**, **trunc**, **round**, **rint**, **lround**, **llround**, **ldexp**, **frexp**, **ilogb**, **logb**, **scalbn**, **scalbln**, **copysign**, **nextafter**, **remainder**, **isnan**, **isinf**, **isfinite**, **isnormal**, **signbit**, **fpclassify**, **isunordered**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, **quantize**, and **samequantum.**

From **<fenv.h>**, facilities dealing with decimal context:
feraiseexcept, **feclearexcept**, **fetestexcept**, **fesetexceptflag**, **fegetexceptflag**, **fe_dec_getround**, **fe_dec_setround**, **fesetenv**, **fegetenv**, **feupdateenv**, and **feholdexcept**.

From **<stdio.h>**, decimal floating-point modified format specifiers for:

The **printf**/**scanf** family of functions.

From **<stdlib.h>** and **<wchar.h>**, the decimal floating-point type versions of:
strtod and **wcstod.**

From **<wchar.h>**, decimal floating-point modified format specifiers for:
The wide **printf**/**scanf** family of functions.

8.3 Conversions

Conversions between different formats and to/from integer formats are covered in section 6.

9 Library

9.1 Standard headers

The functions, macros, and types declared or defined in Clause 9 and its subclauses are only declared or defined by their respective headers if the macro **__STDC_WANT_DEC_FP__** is defined at the point in the source file where the appropriate header is included.

9.2 Floating-point environment <fenv.h>

The floating point environment specified in C99 7.6 applies to both generic floating types and decimal floating types. This is to implement the *context* defined in IEEE 754-2008. The existing C99 specification gives flexibility to an implementation on which part of the environment is accessible to programs. The decimal floating-point arithmetic specifies a more stringent requirement. All the rounding directions and flags are supported.

DEC Macros	Existing C99 macros for generic floating types	IEEE 754
FE_DEC_TOWARDZERO	**FE_TOWARDZERO**	Toward zero
FE_DEC_TONEAREST	**FE_TONEAREST**	To nearest, ties even
FE_DEC_UPWARD	**FE_UPWARD**	Toward plus infinity
FE_DEC_DOWNWARD	**FE_DOWNWARD**	Toward minus infinity
FE_DEC_TONEARESTFROMZERO	n/a	To nearest, ties away from zero

Suggested changes to C99:

Add the following after 7.6 paragraph 7:

7.6
...
[7a] Each of the macros

FE_DEC_DOWNWARD
FE_DEC_TONEAREST
FE_DEC_TONEARESTFROMZERO
FE_DEC_TOWARDZERO
FE_DEC_UPWARD

is defined and used by **fe_dec_getround** and **fe_dec_setround** functions for getting and setting the rounding direction of decimal floating-point operations. The default rounding direction for decimal floating-point operations shall be **FE_DEC_TONEAREST**.

Add the following after 7.6.3.2:

7.6.3.3 The fe_dec_getround function

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <fenv.h>
int fe_dec_getround(void);
```

Description

The **fe_dec_getround** function gets the current rounding direction for decimal floating-point operations.

Returns

The **fe_dec_getround** function returns the value of the rounding direction macro representing the current rounding direction for decimal floating-point operations, or a negative value if there is no such rounding macro or the current rounding direction is not determinable.

7.6.3.4 The fe_dec_setround function

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <fenv.h>
int fe_dec_setround(int round);
```

Description

The **fe_dec_setround** function establishes the rounding direction for decimal floating-point operations represented by its argument round. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

If **FLT_RADIX** is not 10, the rounding direction altered by the **fesetround** function is independent of the rounding direction altered by the **fe_dec_setround** function; otherwise if **FLT_RADIX** is 10, whether the **fesetround** and **fe_dec_setround** functions alter the rounding direction of both generic floating type and decimal floating type operations is implementation defined.

Returns

The **fe_dec_setround** function returns a zero value if and only if the argument is equal to a rounding direction macro (that is, if and only if the requested rounding direction was established).

9.3 Decimal mathematics <math.h>

The list of elementary functions specified in the mathematics library is extended to handle decimal floating-point types. These include functions specified in 7.12.4, 7.12.5, 7.12.6, 7.12.7, 7.12.8, 7.12.9, 7.12.10, 7.12.11, 7.12.12, and 7.12.13. The macros **HUGE_VAL_D32**, **HUGE_VAL_D64**, **HUGE_VAL_D128**, **DEC_INFINITY** and **DEC_NAN** are defined to help using these functions. With the exception of the decimal floating-point functions listed in 8.2, which have accuracy as specified in IEEE 754-2008, the accuracy of decimal floating-point results is implementation-defined. The implementation may state that the accuracy is unknown. All classification macros specified in C99 7.12.3 are also extended to handle decimal floating-point types. The same applies to all comparison macros specified in 7.12.14.

The names of the functions are derived by adding suffixes d32, d64 and d128 to the **double** version of the function name.

Suggested changes to C99:

Add after 7.12 paragraph 2.

7.12

[2a] The types

```
_Decimal32_t
_Decimal64_t
```

are decimal floating types at least as wide as **_Decimal32** and **_Decimal64**, respectively, and such that **_Decimal64_t** is at least as wide as **_Decimal32_t**. If **DEC_EVAL_METHOD**

equals 0, **_Decimal32_t** and **_Decimal64_t** are **_Decimal32** and **_Decimal64**, respectively; if **DEC_EVAL_METHOD** equals 1, they are both **_Decimal64**; if **DEC_EVAL_METHOD** equals 2, they are both **_Decimal128**; and for other values of **DEC_EVAL_METHOD**, they are otherwise implementation-defined.

Add at the end of 7.12 paragraph 3 the following macros.

7.12

[3] The macro

HUGE_VAL_D64

expands to a constant expression of type **_Decimal64** representing infinity. The macros

HUGE_VAL_D32
HUGE_VAL_D128

are respectively **_Decimal32** and **_Decimal128** analogs of **HUGE_VAL_D64**.

Add at the end of 7.12 paragraph 4 the following macro.

7.12

[4] The macro

DEC_INFINITY

expands to a constant expression of type **_Decimal32** representing positive infinity.

Add at the end of 7.12 paragraph 5 the following macro.

7.12

[5] The macro

DEC_NAN

expands to a constant expression of type **_Decimal32** representing a quiet NaN.

Add at the end of 7.12 paragraph 7 the following macro.

7.12

[7] The macro

FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128

are, respectively, **_Decimal32**, **_Decimal64** and **_Decimal128** analogs of **FP_FAST_FMA**.

Suggested changes to C99:

Add the following list of function prototypes to the synopsis of the respective subclauses:

7.12.4 Trigonometric functions

```
_Decimal64 acosd64(_Decimal64 x);
_Decimal32 acosd32(_Decimal32 x);
_Decimal128 acosd128(_Decimal128 x);

_Decimal64 asind64(_Decimal64 x);
_Decimal32 asind32(_Decimal32 x);
_Decimal128 asind128(_Decimal128 x);

_Decimal64 atand64(_Decimal64 x);
_Decimal32 atand32(_Decimal32 x);
_Decimal128 atand128(_Decimal128 x);

_Decimal64 atan2d64(_Decimal64 y, _Decimal64 x);
_Decimal32 atan2d32(_Decimal32 y, _Decimal32 x);
_Decimal128 atan2d128(_Decimal128 y, _Decimal128 x);

_Decimal64 cosd64(_Decimal64 x);
_Decimal32 cosd32(_Decimal32 x);
_Decimal128 cosd128(_Decimal128 x);

_Decimal64 sind64(_Decimal64 x);
_Decimal32 sind32(_Decimal32 x);
_Decimal128 sind128(_Decimal128 x);

_Decimal64 tand64(_Decimal64 x);
_Decimal32 tand32(_Decimal32 x);
_Decimal128 tand128(_Decimal128 x);
```

7.12.5 Hyperbolic functions

```
_Decimal64 acoshd64(_Decimal64 x);
_Decimal32 acoshd32(_Decimal32 x);
```

```
_Decimal128 acoshd128(_Decimal128 x);

_Decimal64 asinhd64(_Decimal64 x);
_Decimal32 asinhd32(_Decimal32 x);
_Decimal128 asinhd128(_Decimal128 x);

_Decimal64 atanhd64(_Decimal64 x);
_Decimal32 atanhd32(_Decimal32 x);
_Decimal128 atanhd128(_Decimal128 x);

_Decimal64 coshd64(_Decimal64 x);
_Decimal32 coshd32(_Decimal32 x);
_Decimal128 coshd128(_Decimal128 x);

_Decimal64 sinhd64(_Decimal64 x);
_Decimal32 sinhd32(_Decimal32 x);
_Decimal128 sinhd128(_Decimal128 x);

_Decimal64 tanhd64(_Decimal64 x);
_Decimal32 tanhd32(_Decimal32 x);
_Decimal128 tanhd128(_Decimal128 x);
```

7.12.6 Exponential and logarithmic functions

```
_Decimal64 expd64(_Decimal64 x);
_Decimal32 expd32(_Decimal32 x);
_Decimal128 expd128(_Decimal128 x);

_Decimal64 exp2d64(_Decimal64 x);
_Decimal32 exp2d32(_Decimal32 x);
_Decimal128 exp2d128(_Decimal128 x);

_Decimal64 expm1d64(_Decimal64 x);
_Decimal32 expm1d32(_Decimal32 x);
_Decimal128 expm1d128(_Decimal128 x);

_Decimal64 frexpd64(_Decimal64 value, int *exp);[2]
_Decimal32 frexpd32(_Decimal32 value, int *exp);
_Decimal128 frexpd128(_Decimal128 value, int *exp);

int ilogbd64(_Decimal64 x);
int ilogbd32(_Decimal32 x);
int ilogbd128(_Decimal128 x);

_Decimal64 ldexpd64(_Decimal64 x, int exp);[3]
```

[2] See suggested changes to the frexp function description below.

```
_Decimal32 ldexpd32(_Decimal32 x, int exp);
_Decimal128 ldexpd128(_Decimal128 x, int exp);

_Decimal64 logd64(_Decimal64 x);
_Decimal32 logd32(_Decimal32 x);
_Decimal128 logd128(_Decimal128 x);

_Decimal64 log10d64(_Decimal64 x);
_Decimal32 log10d32(_Decimal32 x);
_Decimal128 log10d128(_Decimal128 x);

_Decimal64 log1pd64(_Decimal64 x);
_Decimal32 log1pd32(_Decimal32 x);
_Decimal128 log1pd128(_Decimal128 x);

_Decimal64 log2d64(_Decimal64 x);
_Decimal32 log2d32(_Decimal32 x);
_Decimal128 log2d128(_Decimal128 x);

_Decimal64 logbd64(_Decimal64 x);
_Decimal32 logbd32(_Decimal32 x);
_Decimal128 logbd128(_Decimal128 x);

_Decimal64 modfd64(_Decimal64 value, _Decimal64 *iptr);
_Decimal32 modfd32(_Decimal32 value, _Decimal32 *iptr);
_Decimal128 modfd128(_Decimal128 value, _Decimal128 *iptr);

_Decimal64 scalbnd64(_Decimal64 x, int n);
_Decimal32 scalbnd32(_Decimal32 x, int n);
_Decimal128 scalbnd128(_Decimal128 x, int n);

_Decimal64 scalblnd64(_Decimal64 x, long int n);
_Decimal32 scalblnd32(_Decimal32 x, long int n);
_Decimal128 scalblnd128(_Decimal128 x, long int n);
```

7.12.7 Power and absolute-value functions

```
_Decimal64 cbrtd64(_Decimal64 x);
_Decimal32 cbrtd32(_Decimal32 x);
_Decimal128 cbrtd128(_Decimal128 x);

_Decimal64 fabsd64(_Decimal64 x);
_Decimal32 fabsd32(_Decimal32 x);
_Decimal128 fabsd128(_Decimal128 x);
```

[3] See suggested changes to the ldexp function description below.

```
_Decimal64 hypotd64(_Decimal64 x, _Decimal64 y);
_Decimal32 hypotd32(_Decimal32 x, _Decimal32 y);
_Decimal128 hypotd128(_Decimal128 x, _Decimal128 y);

_Decimal64 powd64(_Decimal64 x, _Decimal64 y);
_Decimal32 powd32(_Decimal32 x, _Decimal32 y);
_Decimal128 powd128(_Decimal128 x, _Decimal128 y);

_Decimal64 sqrtd64(_Decimal64 x);
_Decimal32 sqrtd32(_Decimal32 x);
_Decimal128 sqrtd128(_Decimal128 x);
```

7.12.8 Error and gamma functions

```
_Decimal64 erfd64(_Decimal64 x);
_Decimal32 erfd32(_Decimal32 x);
_Decimal128 erfd128(_Decimal128 x);

_Decimal64 erfcd64(_Decimal64 x);
_Decimal32 erfcd32(_Decimal32 x);
_Decimal128 erfcd128(_Decimal128 x);

_Decimal64 lgammad64(_Decimal64 x);
_Decimal32 lgammad32(_Decimal32 x);
_Decimal128 lgammad128(_Decimal128 x);

_Decimal64 tgammad64(_Decimal64 x);
_Decimal32 tgammad32(_Decimal32 x);
_Decimal128 tgammad128(_Decimal128 x);
```

7.12.9 Nearest integer functions

```
_Decimal64 ceild64(_Decimal64 x);
_Decimal32 ceild32(_Decimal32 x);
_Decimal128 ceild128(_Decimal128 x);

_Decimal64 floord64(_Decimal64 x);
_Decimal32 floord32(_Decimal32 x);
_Decimal128 floord128(_Decimal128 x);

_Decimal64 nearbyintd64(_Decimal64 x);
_Decimal32 nearbyintd32(_Decimal32 x);
_Decimal128 nearbyintd128(_Decimal128 x);

_Decimal64 rintd64(_Decimal64 x);
_Decimal32 rintd32(_Decimal32 x);
```

```
_Decimal128 rintd128(_Decimal128 x);

long int lrintd64(_Decimal64 x);
long int lrintd32(_Decimal32 x);
long int lrintd128(_Decimal128 x);

long long int llrintd64(_Decimal64 x);
long long int llrintd32(_Decimal32 x);
long long int llrintd128(_Decimal128 x);

_Decimal64 roundd64(_Decimal64 x);
_Decimal32 roundd32(_Decimal32 x);
_Decimal128 roundd128(_Decimal128 x);

long int lroundd64(_Decimal64 x);
long int lroundd32(_Decimal32 x);
long int lroundd128(_Decimal128 x);

long long int llroundd64(_Decimal64 x);
long long int llroundd32(_Decimal32 x);
long long int llroundd128(_Decimal128 x);

_Decimal64 truncd64(_Decimal64 x);
_Decimal32 truncd32(_Decimal32 x);
_Decimal128 truncd128(_Decimal128 x);
```

7.12.10 Remainder functions[4]

```
_Decimal64 fmodd64(_Decimal64 x, _Decimal64 y);
_Decimal32 fmodd32(_Decimal32 x, _Decimal32 y);
_Decimal128 fmodd128(_Decimal128 x, _Decimal128 y);

_Decimal64 remainderd64(_Decimal64 x, _Decimal64 y);
_Decimal32 remainderd32(_Decimal32 x, _Decimal32 y);
_Decimal128 remainderd128(_Decimal128 x, _Decimal128 y);
```

7.12.11 Manipulation functions

```
_Decimal64 copysignd64(_Decimal64 x, _Decimal64 y);
_Decimal32 copysignd32(_Decimal32 x, _Decimal32 y);
_Decimal128 copysignd128(_Decimal128 x, _Decimal128 y);

_Decimal64 nand64(const char *tagp);
_Decimal32 nand32(const char *tagp);
_Decimal128 nand128(const char *tagp);
```

[4] There is no decimal floating-point type versions of the remquo function.

```
_Decimal64 nextafterd64(_Decimal64 x, _Decimal64 y);
_Decimal32 nextafterd32(_Decimal32 x, _Decimal32 y);
_Decimal128 nextafterd128(_Decimal128 x, _Decimal128 y);

_Decimal64 nexttowardd64(_Decimal64 x, _Decimal128 y);
_Decimal32 nexttowardd32(_Decimal32 x, _Decimal128 y);
_Decimal128 nexttowardd128(_Decimal128 x, _Decimal128 y);
```

7.12.12 Maximum, minimum, and positive difference functions

```
_Decimal64 fdimd64(_Decimal64 x, _Decimal64 y);
_Decimal32 fdimd32(_Decimal32 x, _Decimal32 y);
_Decimal128 fdimd128(_Decimal128 x, _Decimal128 y);

_Decimal64 fmaxd64(_Decimal64 x, _Decimal64 y);
_Decimal32 fmaxd32(_Decimal32 x, _Decimal32 y);
_Decimal128 fmaxd128(_Decimal128 x, _Decimal128 y);

_Decimal64 fmind64(_Decimal64 x, _Decimal64 y);
_Decimal32 fmind32(_Decimal32 x, _Decimal32 y);
_Decimal128 fmind128(_Decimal128 x, _Decimal128 y);
```

7.12.13 Floating multiply-add

```
_Decimal64 fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal32 fmad32(_Decimal32 x, _Decimal32 y, _Decimal32 z);
_Decimal128 fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
```

Add to the end of 7.12.14 paragraph 1:

[1] … If either argument has decimal floating type, the other argument shall have decimal floating type as well.

Replace 7.12.6.4 paragraphs 2 and 3 with the following:

[2] The **`frexp`** functions break a floating-point number into a normalized fraction and an integer exponent. They store the integer in the **`int`** object pointed to by **`exp`**. If **`value`** is a decimal floating-point number, the exponent is an integral power of 10; otherwise it is an integral power of 2.

[3] If **`value`** is not a floating-point number, the results are unspecified. Otherwise, the **`frexp`** functions return the value **`x`**, such that **`x`** has a magnitude in the interval [1/10, 1) or zero, and **`value`** equals **`x`** * $10^{*\mathbf{exp}}$ when **`value`** is a decimal floating-point number, or **`x`** has a magnitude in the interval [1/2, 1) or zero, and **`value`** equals **`x`** * $2^{*\mathbf{exp}}$ when **`value`** is a generic floating-point number. If **`value`** is zero, both parts of the result are zero.

Replace 7.12.6.6 paragraphs 2 and 3 with the following:

[2] The **ldexp** functions multiply a decimal floating-point number by an integral power of 10, or a generic floating-point number by an integral power of 2. A range error may occur.

[3] If **x** is a decimal floating-point number, the **ldexp** functions return **x** * $10^{\mathbf{exp}}$; otherwise they return **x** * $2^{\mathbf{exp}}$.

Replace 7.12.6.11 paragraph 2 with the following:

The **logb** functions extract the exponent of **x**, as a signed integer value in floating-point format. If **x** is subnormal it is treated as though it were normalized; thus, for positive finite **x**,

$$1 \leq \mathbf{x} * b^{-\mathbf{logb}(\mathbf{x})} < b$$

where b = 10 if **x** is a decimal floating-point number; otherwise b = **FLT_RADIX**.

A domain error or range error may occur if the argument is zero.

Replace 7.12.6.13 paragraphs 2 and 3 with the following:

[2] The **scalbn** and **scalbln** functions compute **x** * $b^{\mathbf{n}}$ (where b = 10 if **x** is a decimal floating-point number; otherwise b = **FLT_RADIX**) efficiently, not normally by computing $b^{\mathbf{n}}$ explicitly. A range error may occur.

[3] The **scalbn** and **scalbln** functions return **x** * $b^{\mathbf{n}}$.

9.4 New <math.h> functions

The following are new functions added to **<math.h>**.

Suggested addition to C99:

7.12.11.5 The quantize functions

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <math.h>
_Decimal32  quantized32 (_Decimal32 x,  _Decimal32 y);
_Decimal64  quantized64 (_Decimal64 x,  _Decimal64 y);
_Decimal128 quantized128(_Decimal128 x, _Decimal128 y);
```

Description

The **quantize** functions set the exponent of argument **x** to the exponent of argument **y**, while attempting to keep the value the same. If the exponent is being increased, the value shall be correctly rounded according to the current rounding mode; if the result does not have the same value as **x**, the "inexact" floating-point exception shall be raised. If the exponent is being decreased and the significand of the result has more digits than the type would allow, the result is NaN and the "invalid" floating-point exception shall be raised. If one or both operands are NaN the result is NaN. Otherwise if only one operand is infinity, the result is NaN and the "invalid" floating-point exception shall be raised. If both operands are infinity, the result is **DEC_INFINITY** with the sign as **x**, converted to the type of the function. The **quantize** functions do not signal underflow.

Returns

The **quantize** functions return the number which is equal in value (except for any rounding) and sign to **x**, and which has an exponent set to be equal to the exponent of **y**.

7.12.11.6 The samequantum functions

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <math.h>
_Bool  samequantumd32  (_Decimal32 x,  _Decimal32 y);
_Bool  samequantumd64  (_Decimal64 x,  _Decimal64 y);
_Bool  samequantumd128 (_Decimal128 x, _Decimal128 y);
```

Description

The **samequantum** functions determine if the quantum exponents of the **x** and **y** are the same. If both **x** and **y** are NaN, or infinity, they have the same quantum exponents; if exactly one operand is infinity or exactly one operand is NaN, they do not have the same quantum exponents. The **samequantum** functions raise no exception.

Returns

The **samequantum** functions return **true** when **x** and **y** have the same quantum exponents, **false** otherwise.

7.12.11.7 The quantexp functions

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <math.h>
```

```
int quantexpd32  (_Decimal32 x);
int quantexpd64  (_Decimal64 x);
int quantexpd128 (_Decimal128 x);
```

Description

The **quantexp** functions compute the quantum exponent of a finite argument. If **x** is infinite or NaN, they compute **INT_MIN** and a domain error occurs.

Returns

The **quantexp** functions return the quantum exponent of **x**.

9.5 Formatted input/output specifiers

Suggested changes to C99:

Add the following to 7.19.6.1 paragraph 7, to 7.19.6.2 paragraph 11, to 7.24.2.1 paragraph 7, and to 7.24.2.2 paragraph 11:

H Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **_Decimal32** argument.

D Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **_Decimal64** argument.

DD Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **_Decimal128** argument.

Change all occurrences of:

A **double** argument representing …

in the descriptions of the **e**, **E**, **f**, **F**, **g**, and **G** conversion specifiers in 7.19.6.1 paragraph 8 and 7.24.2.1 paragraph 8 to:

A **double** or decimal floating type argument representing …

Change the second paragraph in the description of the **a**,**A** conversion specifier in 7.19.6.1 paragraph 8 and 7.24.2.1. paragraph 8 to:

A **double** or decimal floating type argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

Add the following to 7.19.6.1 paragraph 8 and 7.24.2.1 paragraph 8, under **a**,**A** conversion specifiers:

If an ***H***, ***D***, or ***DD*** modifier is present and the precision is missing, then for a decimal floating type argument represented by a triple of integers (*s*, *c*, *q*), where *n* is the number of digits in the coefficient *c*,

- if 0 >= *q* >= -(*n*+5), use style **f** formatting with formatting precision equal to -***q***,
- otherwise, use style **e** formatting with formatting precision equal to *n* - 1, with the exceptions that if *c* = 0 then the *digit-sequence* in the *exponent-part* shall have the value *q* (rather than 0), and that the exponent is always expressed with the minimum number of digits required to represent its value (the exponent never contains a leading zero).

If the precision modifier is present and at least as large as the precision *p* (5.2.4.2.2) of the decimal floating type, the conversion is as if the precision modifier were missing. If the precision modifier is present and less than the precision *p* of the decimal floating type, the conversion first rounds the input, in the type, according to the current rounding direction for decimal floating-point operations, to the number of digits specified by the precision modifier, then converts the result as if the precision modifier were missing.

Examples:

Following are representations of **_Decimal64** arguments as triples (*s*, *c*, *q*) and the corresponding character sequences **printf** produces with **%Da**:

(1, 123, 0)	123
(-1, 123, 0)	-123
(1, 123, -2)	1.23
(1, 123, 1)	1.23e+3
(-1, 123, 1)	-1.23e+3
(1, 123, -8)	0.00000123
(1, 123, -9)	1.23e-7
(1, 1234567890123456, 0)	1234567890123456
(1, 1234567890123456, 1)	1.234567890123456e+16
(1, 1234567890123456, -1)	123456789012345.6
(1, 1234567890123456, -21)	0.000001234567890123456
(1, 1234567890123456, -22)	1.234567890123456e-7
(1, 0, 0)	0
(-1, 0, 0)	-0
(1, 0, -6)	0.000000
(1, 0, -7)	0e-7
(1, 0, 2)	0e+2
(1, 5, -6)	0.000005
(1, 50, -7)	0.0000050
(1, 5, -7)	5e-7

To illustrate the effects of a precision modifier, the sequence:

```
_Decimal32 x = 6543.00DF;  // represented by the triple (1, 654300, -2)
printf("%Ha\n", x);
printf("%.6Ha\n", x);
printf("%.5Ha\n", x);
printf("%.4Ha\n", x);
printf("%.3Ha\n", x);
printf("%.2Ha\n", x);
printf("%.1Ha\n", x);
```

results in:

```
6543.00
6543.00
6543.0
6543
6.54e+3
6.5e+3
7e+3
```

9.6 strtod32, strtod64, and strtod128 functions <stdlib.h>

The specifications of these functions are similar to those of **`strtod`**, **`strtof`**, and **`strtold`** as defined in C99 7.20.1.3. These functions are declared in **`<stdlib.h>`**.

Suggested addition to C99:

7.20.1.5 The strtod32, strtod64, and strtod128 functions

Synopsis

[1]
```
#define __STDC_WANT_DEC_FP__
#include <stdlib.h>
_Decimal32  strtod32 (const char * restrict nptr, char ** restrict endptr);
_Decimal64  strtod64 (const char * restrict nptr, char ** restrict endptr);
_Decimal128 strtod128(const char * restrict nptr, char ** restrict endptr);
```

Description

[2] The **`strtod32`**, **`strtod64`**, and **`strtod128`** functions convert the initial portion of the string pointed to by **`nptr`** to **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **`isspace`** function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or

more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

[3] The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2;
- **`INF`** or **`INFINITY`**, ignoring case
- **`NAN`** or **`NAN`**(*d-char-sequence*$_{opt}$), ignoring case in the **`NAN`** part, where:

d-char-sequence:
digit
d-char-sequence digit

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

[4] If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that it is not a hexadecimal floating number, that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A character sequence **`INF`** or **`INFINITY`** is interpreted as an infinity. A character sequence **`NAN`** or **`NAN`**(*d-char-sequence*$_{opt}$), is interpreted as a quiet NaN; the meaning of the d-char sequences is implementation-defined.[5] A pointer to the final string is stored in the object pointed to by **`endptr`**, provided that **`endptr`** is not a null pointer.

[5] If the sequence is negated, the sign *s* is set to -1, else *s* is set to 1.

[6] If the subject sequence has the expected form for a floating-point number, then the result shall be correctly rounded as specified in IEEE 754-2008.

[7] The coefficient *c* and the quantum exponent *q* of a finite converted floating-point number are determined from the subject sequence as follows:

- The *fractional-constant* or *digit-sequence* and the *exponent-part* (if any) are extracted from the subject sequence. If there is an *exponent-part*, then *q* is set to the value of *sign*$_{opt}$ *digit-sequence* in the *exponent-part*. If there is no *exponent-part*, *q* is set to 0.

[5] An implementation may use the d-char sequence to determine extra information to be represented in the NaN's significand.

- If there is a *fractional-constant*, *q* is decreased by the number of digits to the right of the decimal point and the decimal point is removed to form a *digit-sequence*.
- *c* is set to the value of the *digit-sequence* (after any decimal point has been removed).
- Rounding required because of insufficient precision or range in the type of the result will round *c* to the full precision available in the type, and will adjust *q* accordingly within the limits of the type, provided the rounding does not yield an infinity (in which case an appropriately signed internal representation of infinity is returned). If the full precision of the type would require *q* to be smaller than the minimum for the type, then *q* is pinned at the minimum and *c* is adjusted through the subnormal range accordingly, perhaps to zero.

Examples:

Following are subject sequences of the decimal form and the resulting triples (*s*, *c*, *q*) produced by **`strtod64`**. Note that for **`_Decimal64`**, the precision (maximum coefficient length) is 16 and the quantum exponent range is -398 <= *q* <= 369.

"0" (1,0,0)
"0.00" (1,0,-2)
"123" (1,123,0)
"-123" (-1,123,0)
"1.23E3" (1,123,1)
"1.23E+3" (1,123,1)
"12.3E+7" (1,123,6)
"12.0" (1,120,-1)
"12.3" (1,123,-1)
"0.00123" (1,123,-5)
"-1.23E-12" (-1,123,-14)
"1234.5E-4" (1,12345,-5)
"-0" (-1,0,0)
"-0.00" (-1,0,-2)
"0E+7" (1,0,7)
"-0E-7" (-1,0,-7)
"12345678901234567890" (1, 1234567890123457, 4) or (1, 1234567890123456, 4) depending on rounding mode
"1234E-400" (1, 12, -398) or (1, 13, -398) depending on rounding mode
"1234E-402" (1, 0, -398) or (1, 1, -398) depending on roundingmode
"1000." (1,1000,0)
".0001" (1,1,-4)
"1000.e0" (1,1000,0)
".0001e0" (1,1,-4)
"1000.0" (1,10000,-1)
"0.0001" (1,1,-4)
"1000.00" (1,100000,-2)
"00.0001" (1,1,-4)
"001000." (1,1000,0)
"001000.0" (1,10000,-1)

"001000.00"	(1,100000,-2)
"00.00"	(1,0,-2)
"00."	(1,0,0)
".00"	(1,0,-2)
"00.00e-5"	(1,0,-7)
"00.e-5"	(1,0,-5)
".00e-5"	(1,0,-7)

[8] In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

[9] If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

[10] The functions return the converted value, if any. If no conversion could be performed, the value +0.E0dd is returned. If the correct value is outside the range of representable values, plus or minus **HUGE_VAL_D64**, **HUGE_VAL_D32**, or **HUGE_VAL_D128** is returned (according to the return type and sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno** acquires the value **ERANGE** is implementation-defined.

9.7 wcstod32, wcstod64, and wcstod128 functions <wchar.h>

The specifications of these functions are similar to those of **wcstod**, **wcstof**, and **wcstold** as defined in C99 7.24.4.1.1. They are declared in **<wchar.h>**.

Suggested addition to C99:

7.24.4.1.3 The wcstod32, wcstod64, and wcstod128 functions

Synopsis

[1]
```
#define __STDC_WANT_DEC_FP__
#include <wchar.h>
_Decimal32  wcstod32 (const wchar_t * restrict nptr, wchar_t ** restrict endptr);
_Decimal64  wcstod64 (const wchar_t * restrict nptr, wchar_t ** restrict endptr);
_Decimal128 wcstod128(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Description

[2] The **wcstod32**, **wcstod64**, and **wcstod128** functions convert the initial portion of the wide string pointed to by **nptr** to **_Decimal32**, **_Decimal64**, and **_Decimal128** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the **iswspace** function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

[3] The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined in 6.4.4.2;
- **INF** or **INFINITY**, ignoring case
- **NAN** or **NAN**(*d-wchar-sequence*$_{opt}$), ignoring case in the **NAN** part, where:

d-wchar-sequence:
digit
d-wchar-sequence digit

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

[4] If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that it is not a hexadecimal floating number, that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide character sequence **INF** or **INFINITY** is interpreted as an infinity. A wide character sequence **NAN** or **NAN**(*d-wchar-sequence*$_{opt}$), is interpreted as a quiet NaN; the meaning of the d-wchar sequences is implementation-defined.[6] A pointer to the final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

[5] If the sequence is negated, the sign *s* is set to -1, else *s* is set to 1.

[6] If the subject sequence has the expected form for a floating-point number, then the result shall be correctly rounded as specified in IEEE 754-2008.

[6] An implementation may use the d-char sequence to determine extra information to be represented in the NaN's significand.

[7] The coefficient *c* and the quantum exponent *q* of a finite converted floating-point number are determined from the subject sequence as follows:

- The *fractional-constant* or *digit-sequence* and the *exponent-part* (if any) are extracted from the subject sequence. If there is an *exponent-part*, then *q* is set to the value of $sign_{opt}$ *digit-sequence* in the *exponent-part*. If there is no *exponent-part*, *q* is set to 0.
- If there is a *fractional-constant*, *q* is decreased by the number of digits to the right of the decimal point and the decimal point is removed to form a *digit-sequence*.
- *c* is set to the value of the *digit-sequence* (after any decimal point has been removed).
- Rounding required because of insufficient precision or range in the type of the result will round *c* to the full precision available in the type, and will adjust *q* accordingly within the limits of the type, provided the rounding does not yield an infinity (in which case an appropriately signed internal representation of infinity is returned). If the full precision of the type would require *q* to be smaller than the minimum for the type, then *q* is pinned at the minimum and *c* is adjusted through the subnormal range accordingly, perhaps to zero.

[8] In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

[9] If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

[10] The functions return the converted value, if any. If no conversion could be performed, the value +0.E0dd is returned. If the correct value is outside the range of representable values, plus or minus **HUGE_VAL_D64**, **HUGE_VAL_D32**, or **HUGE_VAL_D128** is returned (according to the return type and sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno** acquires the value **ERANGE** is implementation-defined.

9.8 Type-generic macros <tgmath.h>

All new functions added to **<math.h>** are subjected to the same requirement as specified in C99 7.22 to provide support for *type-generic* macro expansion. When one of the type-generic arguments has a decimal floating type, use of the type-generic macro invokes a function whose parameters have the types determined as follows:

If there is more than one real floating type arguments, usual arithmetic conversions are applied to the real floating type arguments so that they have compatible types. Then,

- If any argument has type **_Decimal128**, the type determined is **_Decimal128**.
- Otherwise, if any argument has type **_Decimal64**, or if one argument has an integer type and another argument has type **_Decimal32**, the type determined is **_Decimal64**.
- Otherwise, if any argument has type **_Decimal32**, the type determined is **_Decimal32**.
- Otherwise, the specification in C99 7.22 paragraph 3 applies.

EXAMPLE

```
pow(2,3.0)      // expands to the double version of pow:
                //      pow((double)2, (double)3.0)
pow(2,3.DD)     // expands to the _Decimal64 version of pow:
                //      powd64((_Decimal64)2, (_Decimal64)3.DD)
```

Index