**ISO/IEC JTC 1/SC 22/WG 14 N1724**

Date: yyyy-mm-dd

Reference number of document: **ISO/IEC TS 18661**

Committee identification: ISO/IEC JTC 1/SC 22/WG 14

Secretariat: ANSI

# Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 2: Decimal floating-point arithmetic

*Technologies de l'information — Langages de programmation, leurs environnements et interfaces du logiciel système — Extensions à virgule flottante pour C — Partie 2: décimal arithmétique flottante*

**Warning**

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an 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 Specification
Document subtype:
Document stage: (20) Preparation
Document language: E

## Copyright notice

# Contents

Page

**Introduction** v
  **Background** v
    **IEC 60559 floating-point standard** v
    **C support for IEC 60559** vi
  **Purpose** vii
  **Additional background on decimal floating-point arithmetic** vii

**1 Scope** 1

**2 Conformance** 1

**3 Normative references** 2

**4 Terms and definitions** 2

**5 C standard conformance** 2
**5.1 Freestanding implementations** 2
**5.2 Predefined macros** 2
**5.3 Standard headers** 3

**6 Decimal floating types** 3

**7 Characteristics of decimal floating types `<float.h>`** 4

**8 Operation binding** 8

**9 Conversions** 9
**9.1 Conversions between decimal floating and integer types** 9
**9.2 Conversions among decimal floating types, and between decimal floating and standard floating types** 10
**9.3 Conversions between decimal floating and complex types** 10
**9.4 Usual arithmetic conversions** 10
**9.5 Default argument promotion** 11

**10 Constants** 11

**11 Arithmetic operations** 12
**11.1 Operators** 12
**11.2 Functions** 12
**11.3 Conversions** 13
**11.4 Expression transformations** 13

**12 Library** 14
**12.1 Standard headers** 14
**12.2 Floating-point environment `<fenv.h>`** 14
**12.3 Decimal mathematics `<math.h>`** 16
**12.4 New `<math.h>` functions** 25
**12.4.1 Quantum and quantum exponent functions** 25
**12.4.2 Decimal re-encoding functions** 27
**12.5 Formatted input/output specifiers** 28
**12.6 `strtod32`, `strtod64`, and `strtod128` functions `<stdlib.h>`** 30
**12.7 `wcstod32`, `wcstod64`, and `wcstod128` functions `<wchar.h>`** 33
**12.8 `strfromd32`, `strfromd64`, and `strfromd128` functions `<stdlib.h>`** 35
**12.9 Type-generic macros `<tgmath.h>`** 36

**Bibliography** 39

# Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TS 18661 was prepared by Technical Committee ISO JTC 1, *Information Technology*, Subcommittee SC 22, *Programming languages, their environments, and system software interfaces*.

ISO/IEC TS 18661 consists of the following parts, under the general title *Floating-point extensions for C*:

— *Part 1: Binary floating-point arithmetic*

— *Part 2: Decimal floating-point arithmetic*

— *Part 3: Interchange and extended types*

— *Part 4: Supplemental functions*

— *Part 5: Supplemental attributes*

Part 1 updates ISO/IEC 9899:2011 (*Information technology — Programming languages, their environments and system software interfaces — Programming Language C*), Annex F in particular, to support all required features of ISO/IEC/IEEE 60559:2011 (*Information technology — Microprocessor Systems — Floating-point arithmetic*).

Part 2 supersedes ISO/IEC TR 24732:2009 (Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic).

Parts 3-5 specify extensions to ISO/IEC 9899:2011 for features recommended in ISO/IEC/IEEE 60559:2011.

# Introduction

## Background

### IEC 60559 floating-point standard

The IEEE 754-1985 standard for binary floating-point arithmetic was motivated by an expanding diversity in floating-point data representation and arithmetic, which made writing robust programs, debugging, and moving programs between systems exceedingly difficult. Now the great majority of systems provide data formats and arithmetic operations according to this standard. The IEC 60559:1989 international standard was equivalent to the IEEE 754-1985 standard. Its stated goals were:

1 Facilitate movement of existing programs from diverse computers to those that adhere to this standard.

2 Enhance the capabilities and safety available to programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs. However, we recognize that utility and safety are sometimes antagonists.

3 Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. When restricted to a declared subset of the standard, these programs should produce identical results on all conforming systems.

4 Provide direct support for

   a. Execution-time diagnosis of anomalies

   b. Smoother handling of exceptions

   c. Interval arithmetic at a reasonable cost

5 Provide for development of

   a. Standard elementary functions such as exp and cos

   b. Very high precision (multiword) arithmetic

   c. Coupling of numerical and symbolic algebraic computation

6 Enable rather than preclude further refinements and extensions.

To these ends, the standard specified a floating-point model comprising:

*formats* – for binary floating-point data, including representations for Not-a-Number (NaN) and signed infinities and zeros

*operations* – basic arithmetic operations (addition, multiplication, etc.) on the format data to compose a well-defined, closed arithmetic system (It also specified conversions between floating-point formats and decimal character sequences, and a few auxiliary operations.)

*context* – status flags for detecting exceptional conditions (invalid operation, division by zero, overflow, underflow, and inexact) and controls for choosing different rounding methods

The IEC 60559:2011 international standard is equivalent to the IEEE 754-2008 standard for floating-point arithmetic, which is a major revision to IEEE 754-1985.

The revised standard specifies more formats, including decimal as well as binary. It adds a 128-bit binary format to its basic formats. It defines extended formats for all of its basic formats. It specifies data interchange formats (which may or may not be arithmetic), including a 16-bit binary format and an unbounded tower of wider formats. To conform to the floating-point standard, an implementation must provide at least one of the basic formats, along with the required operations.

The revised standard specifies more operations. New requirements include -- among others -- arithmetic operations that round their result to a narrower format than the operands (with just one rounding), more conversions with integer types, more classifications and comparisons, and more operations for managing flags and modes. New recommendations include an extensive set of mathematical functions and seven reduction functions for sums and scaled products.

The revised standard places more emphasis on reproducible results, which is reflected in its standardization of more operations. For the most part, behaviors are completely specified. The standard requires conversions between floating-point formats and decimal character sequences to be correctly rounded for at least three more decimal digits than is required to distinguish all numbers in the widest supported binary format; it fully specifies conversions involving any number of decimal digits. It recommends that transcendental functions be correctly rounded.

The revised standard requires a way to specify a constant rounding direction for a static portion of code, with details left to programming language standards. This feature potentially allows rounding control without incurring the overhead of runtime access to a global (or thread) rounding mode.

Other features recommended by the revised standard include alternate methods for exception handling, controls for expression evaluation (allowing or disallowing various optimizations), support for fully reproducible results, and support for program debugging.

The revised standard, like its predecessor, defines it model of floating-point arithmetic 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.

IEC 60559 does not include bindings of its floating-point model for particular programming languages. However, the revised standard does include guidance for programming language standards, in recognition of the fact that features of the floating-point standard, even if well supported in the hardware, are not available to users unless the programming language provides a commensurate level of support. The implementation’s combination of both hardware and software determines conformance to the floating-point standard.

## C support for IEC 60559

The C standard specifies floating-point arithmetic using an abstract model. The representation of a floating-point number is specified in an abstract form where the constituent components (sign, exponent, significand) of the representation are defined 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, before the floating-point standard was established, there were various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would have made most of the existing implementations at the time not conforming.

Beginning with ISO/IEC 9899:1999 (C99), C has included an optional second level of specification for implementations supporting the floating-point standard. C99, in conditionally normative Annex F, introduced nearly complete support for the IEC 60559:1989 standard for binary floating-point arithmetic. Also, C99’s informative Annex G offered a specification of complex arithmetic that is compatible with IEC 60559:1989.

ISO/IEC 9899:2011 (C11) includes refinements to the C99 floating-point specification, though is still based on IEC 60559:1989. C11 upgrades Annex G from “informative” to “conditionally normative”.

ISO/IEC Technical Report 24732:2009 introduced partial C support for the decimal floating-point arithmetic in IEC 60559:2011. TR 24732, for which technical content was completed while IEEE 754-2008 was still in the later stages of development, specifies decimal types based on IEC 60559:2011 decimal formats, though it does not include all of the operations required by IEC 60559:2011.

## Purpose

The purpose of this Technical Specification is to provide a C language binding for IEC 60559:2011, based on the C11 standard, that delivers the goals of IEC 60559 to users and is feasible to implement. It is organized into five Parts.

Part 1, this document, provides changes to C11 that cover all the requirements, plus some basic recommendations, of IEC 60559:2011 for binary floating-point arithmetic. C implementations intending to support IEC 60559:2011 are expected to conform to conditionally normative Annex F as enhanced by the changes in Part 1.

Part 2 enhances TR 24732 to cover all the requirements, plus some basic recommendations, of IEC 60559:2011 for decimal floating-point arithmetic. C implementations intending to provide an extension for decimal floating-point arithmetic supporting IEC 60559-2011 are expected to conform to Part 2.

Part 3 (Interchange and extended types), Part 4 (Supplementary functions), and Part 5 (Supplementary attributes) cover recommended features of IEC 60559-2011. C implementations intending to provide extensions for these features are expected to conform to the corresponding Parts.

## Additional background on decimal floating-point arithmetic

Most of today's general-purpose computing architectures provide binary floating-point arithmetic in hardware. Binary floating point is an efficient representation that 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 (and the equivalent international ISO/IEC 60559 standard).

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, conversion between binary and decimal 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 z/Architecture (and its predecessors since System/360) is a widely used system that supports built-in decimal arithmetic. Prior to the IBM System z10 processor, however, this provided 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, REXX, 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.

The IEEE has recognized the importance of this. Decimal floating-point formats and arithmetic are major new features in the IEEE 754:2008 standard and its international equivalent IEC 60559:2011.

# Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 2: Decimal floating-point arithmetic

## 1 Scope

This document, Part 2 of ISO/IEC Technical Specification 18661, extends programming language C, as specified in IEC 9899:2011 (C11), to support decimal floating-point arithmetic conforming to ISO/IEC/IEEE 60559:2011. It covers all requirements of IEC 60559 as they pertain to C decimal floating types.

This document supersedes ISO/IEC TR 24732:2009 (Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic).

This document does not cover binary floating-point arithmetic (which is covered in Part 1 of ISO/IEC TS 18661), nor most other optional features of IEC 60559.

## 2 Conformance

An implementation conforms to Part 2 of Technical Specification 18661 if all the following are true:

a) It meets the requirements for a conforming implementation of C11 with all the changes to C11 specified in Part 2 of Technical Specification 18661.

b) It meets the requirements of the following clauses of C11 Annex F as modified by the changes specified in Parts 1 and 2 of Technical Specification 18661:

- F.2.1 Infinities and NaNs
- F.3 Operations (see clause 8 below)
- F.4 Floating to integer conversions
- F.6 The return statement
- F.7 Contracted expressions
- F.8 Floating-point environment
- F.9 Optimization
- F.10 Mathematics **`<math.h>`** (see clause 8 below)

For the purpose of specifying these conformance requirements, the macros, functions, and values mentioned in the clauses listed above are understood to refer to the corresponding macros, functions, and values defined in this document for decimal floating types. Likewise, the “rounding direction mode” is understood to refer to the rounding direction mode for decimal floating-point arithmetic.

c) It defines **`__STDC_IEC_60559_DFP__`** to **`201`**`ymm`**`L`**.

NOTE Conformance to Part 2 of Technical Specification 18661 does not include all the requirements of Part 1. An implementation may conform to either or both of Parts 1 and 2.

# 3 Normative references

The following referenced documents are indispensable for the application of this document. Only the editions cited apply.

ISO/IEC 9899:2011, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*

ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*

ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic* (with identical content to IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 2008)

ISO/IEC TS 18661-1:yyyy, *Information technology – Programming languages, their environments and system software interfaces – Floating-point extension for C – Part 1: Binary floating-point arithmetic*

# 4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2011 and ISO/IEC/IEEE 60559:2011 and the following apply.

**4.1**
**C11**
standard ISO/IEC 9899:2011, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*, including *Technical Corrigendum 1* (ISO/IEC 9899:2011/Cor. 1:2012)

# 5 C standard conformance

## 5.1 Freestanding implementations

The following change to C11 expands the conformance requirements for freestanding implements so that they might conform to this Part of Technical Specification18661

**Change to C11:**

Append to the third sentence of 4#6:

> The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines **`__STDC_IEC_60559_DFP__`** may also use features in the contents of the standard headers **`<fenv.h>`** and **`<math.h>`** and the numeric conversion functions (7.22.1) of the standard header **`<stdlib.h>`**.

## 5.2 Predefined macros

The following change to C11 replaces **`__STDC_DEC_FP__`**, the conformance macro for decimal floating-point arithmetic specified in TR 24732, with **`__STDC_IEC_60559_DFP__`**, for consistency with the conformance macro for Part 1 of Technical Specification 18661. Note that an implementation may continue to define **`__STDC_DEC_FP__`**, so that programs that use **`__STDC_DEC_FP__`** may remain valid under the changes in Part 2 of Technical Specification 18661.

**Change to C11:**

In 6.10.8.3#1, add:

> **`__STDC_IEC_60559_DFP__`** The integer constant **`201ymmL`**, intended to indicate support of decimal floating-point arithmetic according to IEC 60559.

### 5.3 Standard headers

The library functions, macros, and types defined in this Part of Technical Specification 18661 are defined by their respective headers if the macro **`__STDC_WANT_IEC_18661_EXT2__`** is defined at the point in the source file where the appropriate header is first included.

## 6 Decimal floating types

This Part of Technical Specification 18661 introduces three decimal floating types, designated as **`_Decimal32`**, **`_Decimal64`** and **`_Decimal128`**. These types support the IEC60559 decimal formats: decimal32, decimal64, and decimal128.

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

This part of Technical Specification 18661 introduces the term *standard floating types* to refer to the types **`float`**, **`double`**, and **`long double`**, which are the floating types the C Standard requires unconditionally.

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 Part of Technical Specification 18661 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`**.

**Changes to C11:**

Change the first sentence of 6.2.5#10 from:

> [10] There are three *real floating types,* designated as **`float`**, **`double`**, and **`long double`**

to:

> [10] There are three *standard 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`** 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 standard floating types and the decimal floating types comprise the *real floating types*.

In 6.2.5#10a, attach a footnote to the wording:

> The set of values of the type **`_Decimal32`**

where the footnote is:

> *) The 32-bit format is a storage-only format in IEC 60559.

Add the following to 6.4.1 Keywords:

*keyword*:
**`_Decimal32`**
**`_Decimal64`**
**`_Decimal128`**

Add the following to 6.7.2 Type specifiers:

*type-specifier*:
**`_Decimal32`**
**`_Decimal64`**
**`_Decimal128`**

Add the following bullets in 6.7.2#2 Constraints:

— **`_Decimal32`**

— **`_Decimal64`**

— **`_Decimal128`**

Add the following after 6.7.2#3:

> [3a] The type specifiers **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** shall not be used if the implementation does not support decimal floating types (see 6.10.8.3).

Add the following after 6.5#8:

> [8a] Expressions involving decimal floating types are evaluated according to the semantics of IEC 60559, including production of results with the preferred quantum exponent as specified in IEC 60559.

## 7 Characteristics of decimal floating types `<float.h>`

IEC 60559 defines a general model for floating-point data, specifies formats (both binary and decimal) for the data, and defines encodings for the formats.

The three decimal floating types correspond to decimal formats defined in IEC 60559 as follows:

— **`_Decimal32`** is a *decimal32* format, which is encoded in 32 bits

— **`_Decimal64`** is a *decimal64* format, which is encoded in 64 bits

— **`_Decimal128`** is a *decimal128* format, which is encoded in 128 bits

The value of a finite number is given by $(-1)^{\text{sign}}$ x significand x $10^{\text{exponent}}$. Refer to IEC 60559 for details of the format.

These formats are characterized by the length of the significand and the maximum exponent. Note that, for decimal IEC 60559 decimal formats, trailing zeros in the significand are significant; i.e., 1.0 is equal to but can be distinguished from 1.00. Table 1 below shows these characteristics by type:

**Table 1 – Format characteristics**

| Type | `_Decimal32` | `_Decimal64` | `_Decimal128` |
|---|---|---|---|
| Significand length in digits | 7 | 16 | 34 |
| Maximum Exponent ($E_{max}$) | 97 | 385 | 6145 |
| Minimum Exponent ($E_{min}$) | −94 | −382 | −6142 |

The maximum and minimum exponents in Table 1 are for floating-point numbers expressed with significands less than 1, as in the C11 model (5.2.4.2.2). They differ (by 1) from the maximum and minimum exponents in the IEC 60559 standard, where normalized floating-point numbers are expressed with one significant digit to the left of the radix point.

If the macro **`__STDC_WANT_IEC_18661_EXT2__`** is defined at the point in the source file where the header **`<float.h>`** is first 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 standard floating types.

**Change to C11:**

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] 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.

[3] 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 standard 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_TRUE_MIN** | **0.000001E-95DF** |
| **DEC62_TRUE_MIN** | **0.000000000000001E-383DD** |
| **DEC128_TRUE_MIN** | **0.000000000000000000000000000000001E-6143DL** |

[4] For decimal floating-point arithmetic, 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 $b = 10$.

[5] 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 $(s, c, q)$. The term *quantum* refers to the value of a unit in the last place of the coefficient. Thus, the quantum of $x$ is $b^q$.

**Table 2 – Quantum exponent ranges**

| Type | `_Decimal32` | `_Decimal64` | `_Decimal128` |
|---|---|---|---|
| Maximum Quantum Exponent ($q_{max}$) | 90 | 369 | 6111 |
| Minimum Quantum Exponent ($q_{min}$) | −101 | −398 | −6176 |

[6] For binary floating-point arithmetic following IEC 60559, 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 arithmetic, 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, operation results that are of decimal floating type have a *preferred quantum exponent*, as specified in IEC 60559, which is determined by the quantum exponents of the operands if they have decimal floating types (or by specific rules for conversions from other types). Table 3 below gives rules for determining preferred quantum exponents for results of IEC 60559 operations, and for other operations specified in this document. When exact, these operations produce a result with their preferred quantum exponent, or as close to it as possible within the limitations of the type. When inexact, these operations produce a result with the least possible quantum exponent. 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.

[7] Table 3 shows, for each operation, how the preferred quantum exponents (5.2.4.2.2a) of the operands, Q(**x**), Q(**y**), etc., determine the preferred quantum exponent of the operation result.

**Table 3 – Preferred quantum exponents**

| Operation | Preferred quantum exponent of result |
|---|---|
| **roundeven**, **round**, **trunc**, **ceil**, **floor**, **rint**, **nearbyint** | max(Q(**x**),0) |
| **nextup**, **nextdown**, **nextafter**, **nexttoward** | least possible |
| **remainder** | min(Q(**x**),Q(**y**)) |
| **fmin**, **fmax**, **fminmag**, **fmaxmag** | Q(**x**) if **x** gives the result,<br>Q(**y**) if **y** gives the result |
| **scalbn**, **scalbln**, **ldexp** | Q(**x**)+**y** |
| **logb** | 0 |
| **+**, **fadd**, **faddl**, **daddl** | min(Q(**x**),Q(**y**)) |
| **-**, **fsub**, **fsubl**, **dsubl** | min(Q(**x**),Q(**y**)) |
| *****, **fmul**, **fmull**, **dmull** | Q(**x**)+Q(**y**) |
| **/**, **fdiv**, **fdivl**, **ddivl** | Q(**x**)−Q(**y**) |
| **sqrt**, **fsqrt**, **fsqrtl**, **dsqrtl** | floor(Q(**x**)/2) |
| **fma**, **ffma**, **ffmal**, **dfmal** | min(Q(**x**)+Q(**y**),Q(**z**)) |
| conversion from integer type | 0 |
| exact conversion from non-decimal floating type | 0 |
| inexact conversion from non-decimal floating type | least possible |
| conversion between decimal floating types | Q(**x**) |
| **canonicalize** | Q(**x**) |
| **strtod**, **wcstod**, **scanf**, decimal floating constants | see 7.22.1.5 |
| **-(x)** | Q(**x**) |
| **fabs** | Q(**x**) |
| **copysign** | Q(**x**) |

| | |
|---|---|
| `quantize` | Q(`y`) |
| `quantum` | Q(`x`) |
| `encodedec`, `decodedec`, `encodebin`, `decodebin` | Q(`x`) |
| `fmod` | min(Q(`x`),Q(`y`)) |
| `fdim` | min((Q(`x`),Q(`y`)) if `x`>`y`,<br>0 if `x`≤`y` |
| `cbrt` | floor(Q(`x`)/3) |
| `hypot` | min(Q(`x`),Q(`y`)) |
| `pow` | floor(`y`×Q(`x`)) |
| `modf` | Q(`value`) |
| `*iptr` returned by `modf` | max(Q(`value`),0) |
| `frexp` | Q(`value`) if `value`=0,<br>− (length of coefficient of `value`) otherwise |
| `*res` returned by `setpayload`, `setpayloadsig` | 0 if `pl` does not represent a valid payload,<br>not applicable otherwise (NaN returned) |
| `getpayload` | 0 if `*x` is a NaN,<br>unspecified otherwise |
| transcendental functions | 0 |

## 8 Operation binding

Table 1 and subsequent text in F.3 as specified in Part 1 of Technical Specification 18661, with the further change below, show how the C decimal operations specified in this document, Part 2 of Technical Specification 18661, provide the operations required by IEC 60559 for decimal floating-point arithmetic.

**Change to C11:**

After F.3#9 (see Part 1 of Technical Specification 18661), append the following:

> [10] Decimal versions of the C `remquo` function are not provided. (The C decimal `remainder` functions provide the remainder operation defined by IEC 60559.)
>
> [11] The C `quantize` functions (7.12.11.5) provide the quantize operation defined in IEC 60559 for decimal floating-point arithmetic.
>
> [12] The binding for the convertFormat operation applies to all conversions among IEC 60559 formats. Therefore, for implementations that conform to Annex F, conversions between decimal floating types and standard floating types with IEC 60559 formats are correctly rounded and raise floating-point exceptions as specified in IEC 60559.
>
> [13] IEC 60559 specifies the convertFromHexCharacter and convertToHexCharacter operations only for binary floating-point arithmetic.
>
> [14] The C integer constant `10` provides the radix operation defined in IEC 60559 for decimal floating-point arithmetic.
>
> [15] The C `samequantum` functions (7.12.11.6) provide the sameQuantum operation defined in IEC 60559 for decimal floating-point arithmetic.
>
> [16] The C `fe_dec_getround` (7.6.3.3) and `fe_dec_setround` (7.6.3.4) functions provide the getDecimalRoundingDirection and setDecimalRoundingDirection operations defined in IEC 60559 for decimal floating-point arithmetic.

> [17] The C **`quantum`** (7.12.11.7) and **`llquantexp`** (7.12.11.8) functions compute the quantum and the (quantum) exponent *q* defined in IEC 60559 for decimal numbers viewed as having integer significands.
>
> [18] The C **`encodedec`** (7.12.11.9) and **`decodedec`** (7.12.11.10) functions provide the encodeDecimal and decodeDecimal operations defined in IEC 60559 for decimal floating-point arithmetic.
>
> [19] The C **`encodebin`** (7.12.11.11) and **`decodebin`** (7.12.11.12) functions provide the encodeBinary and decodeBinary operations defined in IEC 60559 for decimal floating-point arithmetic.

# 9 Conversions

## 9.1 Conversions between decimal floating and integer types

For conversions between real floating and integer types, C11 6.3.1.4 leaves the behavior undefined if the conversion result cannot be represented (Annex F.3 and F.4 define the behavior). To help writing portable code, this Part of Technical Specification 18661 provides defined behavior for decimal floating types.

**Changes to C11:**

Change the first sentence of 6.3.1.4#1 from:

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

to:

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

Add the follow paragraph after 6.3.1.4#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#2 from:

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

to:

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

Add the following paragraph after 6.3.1.4#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 cannot be represented exactly, the result shall be correctly rounded with exceptions raised as specified in IEC 60559.

## 9.2 Conversions among decimal floating types, and between decimal floating and standard floating types

The specification of conversions among decimal floating types is similar to the existing one for **`float`**, **`double`**, and **`long double`**, except that when the result cannot be represented exactly, correct rounding is required. Correct rounding is also required for conversions from standard to decimal floating types. Correct rounding for conversions from decimal to standard floating types is required only in Annex F for standard types conforming to IEC 60559.

**Change to C11:**

Replace 6.3.1.5#1:

> [1] When a value of real floating type is converted to a real 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 is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined. Results of some implicit conversions (6.3.1.8, 6.8.6.4) may be represented in greater range and precision than that required by the new type.

with:

> [1] When a value of real floating type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged.
>
> [2] When a value of real floating type is converted to a standard floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined.
>
> [3] When a value of real floating type is converted to a decimal floating type, if the value being converted cannot be represented exactly, the result is correctly rounded with exceptions raised as specified in IEC 60559
>
> [4] Results of some implicit conversions (6.3.1.8, 6.8.6.4) may be represented in greater range and precision than that required by the new type.

## 9.3 Conversions between decimal floating and complex types

This is covered by C11 6.3.1.7.

## 9.4 Usual arithmetic conversions

In an application that is written using decimal floating-point 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.

**Change to C11:**

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

> If one operand has decimal floating type, all other operands shall not have standard floating, complex, or imaginary type.
>
> First, if the type of either operand is **`_Decimal128`**, the other operand is converted to **`_Decimal128`**.
>
> Otherwise, if the type of either operand is **`_Decimal64`**, the other operand is converted to **`_Decimal64`**.

Otherwise, if the type of 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>

### 9.5 Default argument promotion

There is no default argument promotion specified for the decimal floating types. Default argument promotion covered in C11 6.5.2.2 [6] and [7] remains unchanged, and applies to standard floating types only.

## 10 Constants

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

This specification does not carry forward two features introduced in TR 24732: the `FLOAT_CONST_DECIMAL64` pragma and the `d` and `D` suffixes for floating constants. The pragma changed the interpretation of unsuffixed floating constants between `double` and `_Decimal46`. The suffixes provided a way to designate `double` floating constants so that the pragma would not affect them. The pragma is not included because of its potential for inadvertently reinterpreting constants. Without the pragma, the suffixes are no longer needed. Also, significant implementations use the `d` and `D` suffixes for other purposes.

**Changes to C11:**

Change *floating-suffix* in 6.4.4.2 from:

*floating-suffix*: one of
**`f l F L`**

to:

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

Add the following paragraph after 6.4.4.2#2:

[2a] Constraints

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

Add the following paragraph after 6.4.4.2#4:

[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] Decimal floating-point constants that have the same numerical value but different quantum exponents have distinguishable internal representations. The quantum exponent is specified to be the same as for the corresponding `strtod32`, `strtod64`, or `strtod128` function for the same numeric string.

# 11 Arithmetic operations

## 11.1 Operators

The operators *Add* (C11 6.5.6), *Subtract* (C11 6.5.6), *Multiply* (C11 6.5.5), *Divide* (C11 6.5.5), *Relational operators* (C11 6.5.8), *Equality operators* (C11 6.5.9), *Unary Arithmetic operators* (C11 6.5.3.3), and *Compound Assignment operators* (C11 6.5.16.2) when applied to decimal floating type operands shall follow the semantics as defined in IEC 60559.

**Changes to C11:**

Add the following after 6.5.5#2:

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

Add the following after 6.5.6#3:

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

Add the following after 6.5.8#2:

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

Add the following after 6.5.9#2:

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

Add the following bullet to 6.5.15#3:

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

Add the following after 6.5.16.2#2:

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

## 11.2 Functions

The headers and library supply a number of functions and macros that implement support for decimal floating-point arithmetic with the semantics specified in IEC 60559, including producing results with the preferred quantum exponent where appropriate. That support is provided by the following:

From C11 **`<math.h>`**, with changes in Part 1 of Technical Specification 18661, the decimal floating-point versions of:

> **`sqrt`**, **`fma`**, **`fabs`**, **`fmax`**, **`fmin`**, **`ceil`**, **`floor`**, **`trunc`**, **`round`**, **`rint`**, **`lround`**, **`llround`**, **`ldexp`**, **`frexp`**, **`ilogb`**, **`logb`**, **`scalbn`**, **`scalbln`**, **`copysign`**, **`remainder`**, **`isnan`**, **`isinf`**, **`isfinite`**, **`isnormal`**, **`signbit`**, **`fpclassify`**, **`isunordered`**, **`isgreater`**, **`isgreaterequal`**, **`isless`**, **`islessequal`** and **`islessgreater`**.

From the `<math.h>` extensions specified in Part 1 of Technical Specification 18661, the decimal floating-point versions of:

> **`roundeven`**, **`nextup`**, **`nextdown`**, **`fminmag`**, **`fmaxmag`**, **`llogb`**, **`fadd`**, **`faddl`**, **`daddl`**, **`fsub`**, **`fsubl`**, **`dsubl`**, **`fmul`**, **`fmull`**, **`dmull`**, **`fdiv`**, **`fdivl`**, **`ddivl`**, **`fsqrt`**, **`fsqrtl`**, **`dsqrtl`**, **`ffma`**, **`ffmal`**, **`dfmal`**, **`fromfp`**, **`ufromfp`**, **`fromfpx`**, **`ufromfpx`**, **`canonicalize`**, **`iseqsig`**, **`issignaling`**, **`issubnormal`**, **`iscanonical`**, **`iszero`**, **`totalorder`**, **`totalordermag`**, **`getpayload`**, **`setpayload`**, and **`setpayloadsig`**.

The `<math.h>` extensions specified below in 12.4 for the decimal-specific functions:

> **`quantize`**, **`samequantum`**, **`quantum`**, **`llquantexp`**, **`encodedec`**, **`decodedec`**, **`encodebin`**, and **`decodebin`**.

From C11 `<fenv.h>`, facilities dealing with decimal context:

> **`feraiseexcept`**, **`feclearexcept`**, **`fetestexcept`**, **`fesetexceptflag`**, **`fegetexceptflag`**, **`fesetenv`**, **`fegetenv`**, **`feupdateenv`**, and **`feholdexcept`**.

From `<fenv.h>` extensions specified in this Part of Technical Specification 18661, facilities dealing with decimal context:

> **`fe_dec_getround`** and **`fe_dec_setround`**.

From the `<fenv.h>` extensions specified in Part 1 of Technical Specification 18661, facilities dealing with decimal context:

> **`fetestexceptflag`**, **`fesetexcept`**, **`fegetmode`**, and **`fesetmode`**.

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

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

From `<stdlib.h>` and `<wchar.h>`, with changes in Part 1 of Technical Specification 18661, the decimal floating-point versions of:

> **`strtod`** and **`wcstod`**.

From the `<stdlib.h>` extensions specified in Part 1 of Technical Specification 18661, the decimal floating-point versions of:

> **`strfromd`**.

From `<wchar.h>`, decimal floating-point modified format specifiers for:

> The **`wprintf`**/**`wscanf`** family of functions.

## 11.3 Conversions

Conversions between different floating types and conversions to and from integer types are covered in clause 9.

## 11.4 Expression transformations

The following changes to C11 alert implementors that some expression transformations must be avoided in order to preserve the quantum exponent (7) of decimal floating-point numbers.

**Changes to C11:**

In F.9.2, insert at the beginning:

> [0a] Valid expression transformations must preserve values.
>
> [0b] The equivalences noted below apply to expressions of standard floating types.
>
> [1] …

In F.9.2, append:

> [2] For expressions of decimal floating types, transformations must preserve quantum exponents, as well as numerical, infinity, and NaN values (5.4.2.2a).
>
> [3] EXAMPLE: **1.** × *x* -> *x* is valid for decimal floating-point expressions *x*, but **1.0** × *x* -> *x* is not:
>
> 1. × 12.34 = (1, 1, 0) x (1, 1234, −2) = (1, 1234, −2) = 12.34
>
> 1.0 × 12.34 = (1, 10, −1) x (1, 1234, −2) = (1, 12340, −3) = 12.340
>
> The results are numerically equal, but have different quantum exponents, hence have different values.

## 12 Library

### 12.1 Standard headers

The functions, macros, and types declared or defined in Clause 12 and its subclauses are only declared or defined by their respective headers if the macro `__STDC_WANT_IEC_18661_EXT2__` is defined at the point in the source file where the appropriate header is first included.

### 12.2 Floating-point environment `<fenv.h>`

The floating-point environment specified in C11 7.6 applies to operations for both standard floating types and decimal floating types. This is to implement the *context* defined in IEC 60559. The existing general C11 specification gives flexibility to an implementation on which part of the environment is accessible to programs. Annex F requires support for all the rounding directions and exception flags (for operations for standard floating types). This document requires support for all the rounding directions and exceptions flags for operations for decimal floating types.

IEC 60559 requires separate rounding modes for binary and decimal floating-point operations. This document requires a separate rounding mode for decimal floating-point operations if the standard floating types are not decimal, and it allows the implementation to define whether the rounding modes are separate or the same if the standard floating types are decimal.

**Table 4 – Rounding mode macros**

| For decimal floating types | For standard floating types | IEC 60559 |
|---|---|---|
| **`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 |

**Changes to C11:**

Add the following after 7.6#6:

> [6a] Decimal floating-point operations and IEC 60559 binary floating-point operations (Annex F) access the same floating-point exception status flags.

Add the following after 7.6#8:

> [8a] Each of the macros
>
> **`FE_DEC_DOWNWARD`**
> **`FE_DEC_TONEAREST`**
> **`FE_DEC_TONEARESTFROMZERO`**
> **`FE_DEC_TOWARDZERO`**
> **`FE_DEC_UPWARD`**
>
> is defined for use with the **`fe_dec_getround`** and **`fe_dec_setround`** functions for getting and setting the dynamic rounding direction mode, and with the **`FENV_ROUND`** rounding control pragma (7.6.1a) for specifying a constant rounding direction, for decimal floating-point operations. The decimal rounding direction affects all (inexact) operations that produce a result of decimal floating type and all operations that produce an integer or character sequence result and have an operand of decimal floating type. The defined macros expand to integer constant expressions whose values are distinct nonnegative values.
>
> [8b] During translation, constant rounding direction modes for decimal floating-point arithmetic are in effect where specified. Elsewhere, during translation the decimal rounding direction mode is **`FE_DEC_TONEAREST`**.
>
> [8c] At program startup the dynamic rounding direction mode for decimal floating-point arithmetic is initialized to **`FE_DEC_TONEAREST`**.

Add the following after 7.6.3.2:

### 7.6.3.3 The `fe_dec_getround` function

**Synopsis**

```
[1] #define __STDC_WANT_IEC_18661_EXT2__
    #include <fenv.h>
    int fe_dec_getround(void);
```

**Description**

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

**Returns**

[3] 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**

```
[1] #define __STDC_WANT_IEC_18661_EXT2__
    #include <fenv.h>
    int fe_dec_setround(int round);
```

**Description**

[2] 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.

[3] 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 standard and decimal floating-point operations is implementation defined.

**Returns**

[4] 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).

## 12.3 Decimal mathematics `<math.h>`

The list of functions specified in the mathematics library is extended to handle decimal floating-point types. These include functions specified in C11 (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) and in Part 1 of Technical Specification 18661 (14.1, 14.2, 14.3, 14.4, 14.5, 14.8, 14.9, and 14.0). The macros `HUGE_VAL_D32`, `HUGE_VAL_D64`, `HUGE_VAL_D128`, `DEC_INFINITY`, `DEC_NAN`, `SNAND32`, `SNAND64`, and `SNAND128` are defined to help using these functions. With the exception of the decimal floating-point functions listed in 11.2, which have accuracy as specified in IEC 60559, the accuracy of decimal floating-point results is implementation-defined. The implementation may state that the accuracy is unknown. All classification macros specified in C11 (7.12.3) and in Part 1 of Technical Specification 18661 (14.7) are also extended to handle decimal floating-point types. The same applies to all comparison macros specified in C11 (7.12.14) and in Part 1 of Technical Specification 18661 (14.6).

The names of the functions are derived by adding suffixes `d32`, `d64`, and `d128` to the `double` version of the function name, except for the functions that round result to narrower type (7.12.13a).

**Changes to C11:**

Add after 7.12#2:

[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#3, the following macros:

[3] The macro

```
HUGE_VAL_D64
```

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

```
HUGE_VAL_D32
HUGE_VAL_D128
```

are respectively `_Decimal32` and `_Decimal128` analogues of `HUGE_VAL_D64`.

Add at the end of 7.12#4, the following macro:

[4] The macro

```
DEC_INFINITY
```

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

Add at the end of 7.12#5, the following macros:

[5a] The macro

```
DEC_NAN
```

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

[5b] The signaling NaN macros

```
SNAND32
SNAND64
SNAND128
```

expand into a constant expression of the respective decimal floating type representing a signaling NaN. If a signaling NaN macro is used for initializing an object of the same type that has static or thread-local storage duration, the object is initialized with a signaling NaN value.

Add at the end of 7.12#7, the following macros:

[7] The macros

```
FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128
```

are, respectively, `_Decimal32`, `_Decimal64`, and `_Decimal128` analogues of `FP_FAST_FMA`.

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);
_Decimal32 frexpd32(_Decimal32 value, int *exp);
_Decimal128 frexpd128(_Decimal128 value, int *exp);

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

long int llogbd64(_Decimal64 x);
long int llogbd32(_Decimal32 x);
long int llogbd128(_Decimal128 x);

_Decimal64 ldexpd64(_Decimal64 x, int exp);
_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);

_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);

_Decimal64 roundevend64(_Decimal64 x);
_Decimal32 roundevend32(_Decimal32 x);
_Decimal128 roundevend128(_Decimal128 x);

intmax_t fromfpd64(_Decimal64 x, int round, unsigned int width);
intmax_t fromfpd32(_Decimal32 x, int round, unsigned int width);
intmax_t fromfpd128(_Decimal128 x, int round, unsigned int width);
uintmax_t ufromfpd64(_Decimal64 x, int round, unsigned int width);
uintmax_t ufromfpd32(_Decimal32 x, int round, unsigned int width);
uintmax_t ufromfpd128(_Decimal128 x, int round, unsigned int width);

intmax_t fromfpxd64(_Decimal64 x, int round, unsigned int width);
intmax_t fromfpxd32(_Decimal32 x, int round, unsigned int width);
intmax_t fromfpxd128(_Decimal128 x, int round, unsigned int width);
uintmax_t ufromfpxd64(_Decimal64 x, int round, unsigned int width);
uintmax_t ufromfpxd32(_Decimal32 x, int round, unsigned int width);
uintmax_t ufromfpxd128(_Decimal128 x, int round, unsigned int width);
```

### 7.12.10 Remainder functions

```
_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);

_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);

_Decimal64 nextupd64(_Decimal64 x);
_Decimal32 nextupd32(_Decimal32 x);
_Decimal128 nextupd128(_Decimal128 x);

_Decimal64 nextdownd64(_Decimal64 x);
_Decimal32 nextdownd32(_Decimal32 x);
_Decimal128 nextdownd128(_Decimal128 x);

_Decimal64 canonicalized64(_Decimal64 x);
_Decimal32 canonicalized32(_Decimal32 x);
_Decimal128 canonicalized128(_Decimal128 x);
```

### 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);

_Decimal64 fmaxmagd64(_Decimal64 x, _Decimal64 y);
_Decimal32 fmaxmagd32(_Decimal32 x, _Decimal32 y);
_Decimal128 fmaxmagd128(_Decimal128 x, _Decimal128 y);

_Decimal64 fminmagd64(_Decimal64 x, _Decimal64 y);
_Decimal32 fminmagd32(_Decimal32 x, _Decimal32 y);
_Decimal128 fminmagd128(_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);
```

### 7.12.14 Functions that round result to narrower format

```
_Decimal32 d32addd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32addd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64addd128(_Decimal128 x, _Decimal128 y);

_Decimal32 d32subd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32subd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64addd128(_Decimal128 x, _Decimal128 y);

_Decimal32 d32muld64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32muld128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64muld128(_Decimal128 x, _Decimal128 y);
```

```
_Decimal32 d32divd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32divd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64divd128(_Decimal128 x, _Decimal128 y);

_Decimal32 d32fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal32 d32fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
_Decimal64 d64fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);

_Decimal32 d32sqrtd64(_Decimal64 x);
_Decimal32 d32sqrtd128(_Decimal128 x);
_Decimal64 d64sqrtd128(_Decimal128 x);
```

F.10.13 Payload functions

```
_Decimal64 getpayloadd64(const _Decimal64 *x);
_Decimal32 getpayloadd32(const _Decimal32 *x);
_Decimal128 getpayloadd128(const _Decimal128 *x);

int setpayloadd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadd128(_Decimal128 *res, _Decimal128 pl);

int setpayloadsigd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadsigd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadsigd128(_Decimal128 *res, _Decimal128 pl);
```

In 7.12.10.3, attach a footnote to the heading:

### 7.12.10.3 The `remquo` functions

where the footnote is:

*) There are no decimal floating-point versions of the `remquo` functions.

Add to the end of 7.12.14#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:

[2] The `frexp` functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer in the `int` object pointed to by `exp`.

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

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 the type of the function is a standard floating type, the exponent is an integral power of 2. If the type of the function is a decimal floating type, the exponent is an integral power of 10.

[3] If `value` is not a floating-point number or the integral power is outside the range of `int`, the results are unspecified. Otherwise, the `frexp` functions return the value `x`, such that: `x` has a

> magnitude in the interval [1/2, 1) or zero, and **value** equals $\mathbf{x} \times 2^{*\mathbf{exp}}$, when the type of the function is a standard floating type; or **x** has a magnitude in the interval [1/10, 1) or zero, and **value** equals $\mathbf{x} \times 10^{*\mathbf{exp}}$, when the type of the function is a decimal floating type. If **value** is zero, both parts of the result are zero.

Replace 7.12.6.6 paragraphs 2 and 3:

> [2] The **ldexp** functions multiply a floating-point number by an integral power of 2. A range error may occur.
>
> [3] The **ldexp** functions return $\mathbf{x} \times 2^{\mathbf{exp}}$.

with the following:

> [2] The **ldexp** functions multiply a floating-point number by an integral power of 2 when the type of the function is a standard floating type, or by an integral power of 10 when the type of the function is a decimal floating type. A range error may occur.
>
> [3] The **ldexp** functions return $\mathbf{x} \times 2^{\mathbf{exp}}$ when the type of the function is a standard floating type, or return $\mathbf{x} \times 10^{\mathbf{exp}}$ when the type of the function is a decimal floating type.

Replace 7.12.6.11#2:

> [2] 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,**
>
> $$1 \le \mathbf{x} \times \mathtt{FLT\_RADIX}^{-\mathrm{logb}(\mathbf{x})} < \mathtt{FLT\_RADIX}$$
>
> A domain error or pole error may occur if the argument is zero.

with the following:

> [2] 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**,
>
> $$1 \le \mathbf{x} \times b^{-\mathrm{logb}(\mathbf{x})} < b$$
>
> where $b$ = **FLT_RADIX** if the type of the function is a standard floating type, or $b = 10$ if the type of the function is a decimal floating type. A domain error or range error may occur if the argument is zero.

Replace 7.12.6.13 paragraphs 2 and 3:

> [2] The **scalbn** and **scalbln** functions compute $\mathbf{x} \times \mathtt{FLT\_RADIX}^{\mathrm{n}}$ efficiently, not normally by computing $\mathtt{FLT\_RADIX}^{\mathrm{n}}$ explicitly. A range error may occur.
>
> [3] The **scalbn** and **scalbln** functions return $\mathbf{x} \times \mathtt{FLT\_RADIX}^{\mathrm{n}}$.

with the following:

> [2] The **scalbn** and **scalbln** functions compute $\mathbf{x} \times b^{\mathrm{n}}$, where $b$ = **FLT_RADIX** if the type of the function is a standard floating type, or $b = 10$ if the type of the function is a decimal floating type. A range error may occur.
>
> [3] The **scalbn** and **scalbln** functions return $\mathbf{x} \times b^{\mathrm{n}}$.

## 12.4 New `<math.h>` functions

This clause adds new functions to **`<math.h>`**.

### 12.4.1 Quantum and quantum exponent functions

This specification does not carry forward the **`quantexp`** functions from TR 24732, which return the quantum exponent of their argument as an **`int`**. Instead it introduces the **`quantum`** functions, which return the quantum rather than the quantum exponent, and the **`llquantexp`** functions, which return the quantum exponent as a **`long long`**, instead of **`int`**. The new interfaces offer natural extensions for support of wider IEC 60559 decimal formats in Part 3 of Technical Specification 18661.

**Change to C11:**

After 7.12.11.4, add:

#### 7.12.11.5 The `quantize` functions

**Synopsis**

[1]

```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
_Decimal32 quantized32 (_Decimal32 x, _Decimal32 y);
_Decimal64 quantized64 (_Decimal64 x, _Decimal64 y);
_Decimal128 quantized128 (_Decimal128 x, _Decimal128 y);
```

**Description**

[2] The **`quantize`** functions set the quantum exponent of argument **`x`** to the quantum exponent of argument **`y`**, while attempting to keep the value the same. If the quantum 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 quantum 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 raise the “underflow” floating-point exception.

**Returns**

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

#### 7.12.11.6 The `samequantum` functions

**Synopsis**

[1]

```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
_Bool samequantumd32(_Decimal32 x, _Decimal32 y);
_Bool samequantumd64(_Decimal64 x, _Decimal64 y);
_Bool samequantumd128(_Decimal128 x, _Decimal128 y);
```

**Description**

[2] The **`samequantum`** functions determine if the quantum exponents of **`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**

[3] The **samequantum** functions return nonzero (true) when **x** and **y** have the same quantum exponents, zero (false) otherwise.

### 7.12.11.7 The `quantum` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
_Decimal32 quantumd32(_Decimal32 x);
_Decimal64 quantumd64(_Decimal64 x);
_Decimal128 quantumd128(_Decimal128 x);
```

**Description**

[2] The **quantum** functions compute the quantum (5.2.4.2.2a) of a finite argument. If **x** is infinite, the result is +∞. If x is NaN, the result is NaN.

**Returns**

[3] The **quantum** functions return the quantum of **x**.

### 7.12.11.8 The `llquantexp` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
long long llquantexpd32(_Decimal32 x);
long long llquantexpd64(_Decimal64 x);
long long llquantexpd128(_Decimal128 x);
```

**Description**

[2] The **llquantexp** functions compute the quantum exponent (5.2.4.2.2a) of a finite argument. If **x** is infinite or NaN, they compute **LLONG_MIN** and a domain error occurs.

**Returns**

[3] The **llquantexp** functions return the quantum exponent of **x**.

### 12.4.2 Decimal re-encoding functions

**Change to C11:**

After 7.12.11.8, add:

#### 7.12.11.9 The `encodedec` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
void encodedecd32(unsigned char * restrict encptr, const _Decimal32 *
   restrict xptr);
void encodedecd64(unsigned char * restrict encptr, const _Decimal64 *
   restrict xptr);
void encodedecd128(unsigned char * restrict encptr, const _Decimal128 *
   restrict xptr);
```

**Description**

[2] The **encodedecd***N* functions convert `*xptr` into an IEC 60559 decimal*N* encoding in the encoding scheme based on decimal encoding of the significand and store the resulting encoding as an *N*/8 element array, with 8 bits per array element, in the object pointed to by `encptr`. The order of bytes in the array is implementation defined. These functions preserve the value of `*xptr` and raise no floating-point exceptions. If `*xptr` is non-canonical, these functions may or may not produce a canonical encoding.

**Returns**

[3] The `encodedec` functions return no value.

#### 7.12.11.10 The `decodedec` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
void decodedecd32(_Decimal32 * restrict xptr, const unsigned char *
   restrict encptr);
void decodedecd64(_Decimal64 * restrict xptr, const unsigned char *
   restrict encptr);
void decodedecd128(_Decimal128 * restrict xptr, const unsigned char *
   restrict encptr);
```

**Description**

[2] The **decodedecd***N* functions interpret the *N*/8 element array pointed to by `encptr` as an IEC 60559 decimal*N* encoding, with 8 bits per array element, in the encoding scheme based on decimal encoding of the significand. The order of bytes in the array is implementation defined. These functions convert the given encoding into a representation in the type **_Decimal***N*, and store the result in the object pointed to by `xptr`. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

**Returns**

[3] The `decodedec` functions return no value.

### 7.12.11.11 The `encodebin` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
void encodebind32(unsigned char * restrict encptr, const _Decimal32 *
   restrict xptr);
void encodebind64(unsigned char * restrict encptr, const _Decimal64 *
   restrict xptr);
void encodebind128(unsigned char * restrict encptr, const _Decimal128 *
   restrict xptr);
```

**Description**

[2] The **`encodebind`***N* functions convert **`*xptr`** into an IEC 60559 decimal*N* encoding in the encoding scheme based on binary encoding of the significand and store the resulting encoding as an *N*/8 element array, with 8 bits per array element, in the object pointed to by **`encptr`**. The order of bytes in the array is implementation defined. These functions preserve the value of **`*xptr`** and raise no floating-point exceptions. If **`*xptr`** is non-canonical, these functions may or may not produce a canonical encoding.

**Returns**

[3] The **`encodebin`** functions return no value.

### 7.12.11.12 The `decodebin` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#include <math.h>
void decodebind32(_Decimal32 * restrict xptr, const unsigned char *
   restrict encptr);
void decodebind64(_Decimal64 * restrict xptr, const unsigned char *
   restrict encptr);
void decodebind128(_Decimal128 * restrict xptr, const unsigned char *
   restrict encptr);
```

**Description**

[2] The **`decodebind`***N* functions interpret the *N*/8 element array pointed to by **`encptr`** as an IEC 60559 decimal*N* encoding, with 8 bits per array element, in the encoding scheme based on binary encoding of the significand. The order of bytes in the array is implementation defined. These functions convert the given encoding into a representation in the type **`_Decimal`***N*, and store the result in the object pointed to by **`xptr`**. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

**Returns**

[3] The **`decodebin`** functions return no value.

## 12.5 Formatted input/output specifiers

With the following decimal forms of the **`a`**,**`A`** format specifiers, the **`printf`** family of functions provide conversions to decimal character sequences that preserve quantum exponents, as required by IEC 60559.

**Changes to C11:**

Add the following to 7.21.6.1#7, 7.21.6.2#11, 7.29.2.1#7, and 7.29.2.2#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.

Add the following to 7.21.6.1#8 and 7.29.2.1#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 is present (in the conversion specification) and is zero or 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 were missing. If the precision is present (and nonzero) and less than the precision *p* of the decimal floating type, the conversion first obtains an intermediate result by rounding 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, then converts the intermediate result as if the precision were missing. The length of the coefficient of the intermediate result is the smallest number, at least as large as the formatting precision, for which the quantum exponent is within the quantum exponent range of the type (see Table 2). The intermediate rounding may overflow.

EXAMPLE 1 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, 120, −8) | `0.00000120` |
| ( 1, 120, −9) | `1.20e-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` |

EXAMPLE 2 To illustrate the effects of a precision specification, 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);
printf(“%.0Ha\n”, x);
```

assuming default rounding, results in:

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

EXAMPLE 3 To illustrate the effects of the exponent range, the sequence:

```
_Decimal32 x = 9543210e87DF; // represented by the triple (1, 9543210, 87)
_Decimal32 y = 9500000e90DF; // represented by the triple (1, 9500000, 90)
printf(“%.6Ha\n”, x);
printf(“%.5Ha\n”, x);
printf(“%.4Ha\n”, x);
printf(“%.3Ha\n”, x);
printf(“%.2Ha\n”, x);
printf(“%.1Ha\n”, x);
printf(“%.1Ha\n”, y);
```

assuming default rounding, results in:

```
9.54321e+93
9.5432e+93
9.543e+93
9.540e+93
9.500e+93
1.0000e+94
inf
```

## 12.6 `strtod32`, `strtod64`, and `strtod128` functions `<stdlib.h>`

The specifications of these functions are similar to those of **`strtod`**, **`strtof`**, and **`strtold`** as defined in C11 7.22.1.3. These functions are declared in **`<stdlib.h>`**.

**Changes to C11:**

After 7.22.1.4, add:

### 7.22.1.5 The `strtod32`, `strtod64`, and `strtod128` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#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 (before rounding). 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. 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 IEC 60559.

[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.

EXAMPLE 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 rounding mode |
| "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) |
| "0x1.8p+4" | (1,0,0), and “x1.8p+4" is stored in the object pointed to by **`endptr`**, provided **`endptr`** is not a null pointer |

[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 of the triple (1,0,0) is returned. If the correct value overflows and default rounding is in effect (7.12.1), 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.

In 7.22.1.5#4, attach a footnote to the wording:

the meaning of the d-char sequences is implementation-defined.

where the footnote is:

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

## 12.7 `wcstod32`, `wcstod64`, and `wcstod128` functions `<wchar.h>`

The specifications of these functions are similar to those of **`wcstod`**, **`wcstof`**, and **`wcstold`** as defined in C11 7.29.4.1.1. They are declared in **`<wchar.h>`**.

**Change to C11:**

After 7.29.4.1.2, add:

### 7.29.4.1.3 The `wcstod32`, `wcstod64`, and `wcstod128` functions

**Synopsis**

[1]
```
#define __STDC_WANT_IEC_18661_EXT2__
#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 (before rounding). 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. 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 IEC 60559.

[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 of the triple (1,0,0) is returned. If the correct value overflows and default rounding is in effect (7.12.1),

> 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.

In 7.29.4.1.3#4, attach a footnote to the wording:

> the meaning of the d-wchar sequences is implementation-defined.

where the footnote is:

> *) An implementation may use the d-wchar sequence to determine extra information to be represented in the NaN's significand.

## 12.8 `strfromd32`, `strfromd64`, and `strfromd128` functions `<stdlib.h>`

The specifications of these functions are similar to those of **`strfromd`**, **`strfromf`**, and **`strfromld`** (7.22.1.2a) as defined in Part 1 (10.2) of Technical Specification 18661. These functions are declared in **`<stdlib.h>`**.

**Change to C11:**

After 7.22.1.5, add:

> **7.22.1.6 The `strfromd32`, `strfromd64`, and `strfromd128` functions**
>
> **Synopsis**
>
> [1]
> ```
> #define __STDC_WANT_IEC_18661_EXT2__
> #include <stdlib.h>
> int strfromd32(char * restrict s, size_t n, const char * restrict
>     format, _Decimal32 fp);
> int strfromd64(char * restrict s, size_t n, const char * restrict
>     format, _Decimal64 fp);
> int strfromd128(char * restrict s, size_t n, const char * restrict
>     format, _Decimal128 fp);
> ```
>
> **Description**
>
> [1] The **`strfromd32`**, **`strfromd64`**, and **`strfromd128`** functions are equivalent to **`snprintf(s, n, format, fp)`** (7.21.6.5), except the **`format`** string contains only an optional precision and one of the conversion specifiers **`a`**, **`A`**, **`e`**, **`E`**, **`f`**, **`F`**, **`g`**, or **`G`**, which applies to the type (**`_Decimal32`**, **`_Decimal64`**, or **`_Decimal128`**) indicated by the function suffix (rather than by a length modifier). Use of these functions with any other **`format`** string results in undefined behavior.
>
> **Returns**
>
> [1] The **`strfromd32`**, **`strfromd64`**, and **`strfromd128`** functions return the number of characters that would have been written had **`n`** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **`n`**.

## 12.9 Type-generic macros `<tgmath.h>`

The following changes to C11 enhance the specification of type-generic macros in **`<tgmath.h>`** to apply to decimal floating types, as well as standard floating types.

**Changes to C11:**

In 7.25, replace paragraphs 2 and 3:

> [2] Of the **`<math.h>`** and **`<complex.h>`** functions without an **`f`** (**`float`**) or **l** (**`long double`**) suffix, several have one or more parameters whose corresponding real type is **`double`.** For each such function, except **`modf`,** there is a corresponding *type-generic macro*.313) The parameters whose corresponding real type is **`double`** in the function synopsis are *generic parameters.* Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.314)
>
> [3] Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:
>
> — First, if any argument for generic parameters has type **long double,** the type determined is **long double.**
>
> — Otherwise, if any argument for generic parameters has type **double** or is of integer type, the type determined is **double.**
>
> — Otherwise, the type determined is **float.**

with:

> [2] This clause specifies a many-to-one correspondence of functions in **<math.h>** and **<complex.h>** with a *type-generic macro*.313) Use of the type-generic macro invokes a corresponding function whose type is determined by the types of the arguments for particular parameters called the *generic parameters*.314)
>
> [3] Of the **<math.h>** and **<complex.h>** functions without an **f** (float) or **l** (long **double)** suffix, several have one or more parameters whose corresponding real type is **double.** For each such function, except **modf**, there is a corresponding type-generic macro.*313)* The parameters whose corresponding real type is **double** in the function synopsis are generic parameters.
>
> [3a] Some of the **<math.h>** functions for decimal floating types have no unsuffixed counterpart. Of these functions with a **d64** suffix, some have one or more parameters whose type is **_Decimal64**. For each such function, except **`decodedecd64`**, **`encodedecd64`**, **`decodebind64`**, and **`encodebind64`**, there is a corresponding type-generic macro. The parameters whose real type is **_Decimal64** in the function synopsis are generic parameters.
>
> [3b] If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type, the behavior is undefined.
>
> [3c] Use of a type-generic macro invokes a function whose generic parameters have the corresponding real type determined by the corresponding real types of the arguments as follows:
>
> — First, if any argument for generic parameters has type **_Decimal128**, the type determined is **_Decimal128**.
>
> — Otherwise, if any argument for generic parameters has type **_Decimal64**, or if any argument for generic parameters is of integer type and another argument for generic parameters has type **_Decimal32**, the type determined is **_Decimal64**.

- Otherwise, if any argument for generic parameters has type **_Decimal32**, the type determined is **_Decimal32**.
- Otherwise, if the corresponding real type of any argument for generic parameters is **long double**, the type determined is **long double**.
- Otherwise, if the corresponding real type any argument for generic parameters is **double** or is of integer type, the type determined is **double**.
- Otherwise, if any argument for generic parameters is of integer type, the type determined is **double**.
- Otherwise, the type determined is **`float`**.

If neither **`<math.h>`** nor **`<complex.h>`** define a function whose generic parameters have the determined corresponding real type, the behavior is undefined.

In 7.25#5, replace the last sentence:

> If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro results in undefined behavior.

with:

> If all arguments for generic parameters are real, then use of the macro invokes a real function (provided **`<math.h>`** defines a function of the determined type); otherwise, use of the macro results in undefined behavior.

In 7.25#6, replace the last sentence:

> Use of the macro with any real or complex argument invokes a complex function.

with:

> Use of the macro with any argument of standard floating or complex type invokes a complex function. Use of the macro with an argument of a decimal floating type results in undefined behavior.

After 7.25#6, add the paragraph:

> [6a] For each **`d64`**-suffixed function in **`<math.h>`**, except **`decodedecd64`**, **`encodedecd64`**, **`decodebind64`**, and **`encodebind64`**, that does not have an unsuffixed counterpart, the corresponding type-generic macro has the name of the function, but without the suffix. These type-generic macros are:

| **`<math.h>`** function | type-generic macro |
|---|---|
| **`quantized`***N* | **`quantize`** |
| **`samequantumd`***N* | **`samequantum`** |
| **`quantumd`***N* | **`quantum`** |
| **`llquantexpd`***N* | **`llquantexp`** |

> Use of the macro with an argument of standard floating or complex type or with only integer type arguments results in undefined behavior.

[6b] A type-generic macro **`cbrt`** that conforms to the specification in this clause and that is affected by constant rounding modes as specified in Part 1 of Technical Specification 18661 could be implemented as follows:

```
#ifdef __STDC_WANT_IEC_18661_EXT2
   #define cbrt(X)  _Generic((X),                  \
                       _Decimal128: cbrtd128(X),    \
                       _Decimal64: cbrtd64(X),      \
                       _Decimal32: cbrtd32(X),      \
                       long double: cbrtl(X),       \
                       default: _Roundwise_cbrt(X), \
                       float: cbrtf(X)              \
                       )
#else
   #define cbrt(X)  _Generic((X),                  \
                       long double: cbrtl(X),       \
                       default: _Roundwise_cbrt(X), \
                       float: cbrtf(X)              \
                       )
#endif
```

where **`_Roundwise_cbrt()`** is equivalent to **`cbrt()`** invoked without macro-replacement suppression.

In 7.25#7, insert at the beginning of the example:

```
#define __STDC_WANT_IEC_18661_EXT2__
```

In 7.25#7, append to the declarations:

```
#if __STDC_IEC_60559_DFP__ >= 201ymmL
_Decimal32 d32;
_Decimal64 d64;
_Decimal128 d128;
#endif
```

In 7.25#7, append to the table:

```
exp(d64)              expd64(d64);
sqrt(d32)             sqrtd32(d32);
fmax(d64, d128)       fmaxd128(d64, d128);
pow(d32, n)           powd64(d32, n);
remainder(d64, d)     undefined behavior
creal(d64)            undefined behavior
remquo(d32, d32, &n)  undefined behavior
llquantexp(d)         undefined behavior
quantize(dc)          undefined behavior
samequantum(n, n)     undefined behavior
```

# Bibliography

[1] ISO/IEC 9899:2011, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*

[2] ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*

[3] ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic*

[4] ISO/IEC TR 24732:2009, *Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic*

[5] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems, second edition*

[6] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*

[7] IEEE 754−1985, *IEEE Standard for Binary Floating-Point Arithmetic*

[8] IEEE 854−1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*