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: Arithmétique décimal en virgule flottante
Foreword ............................................................................................................................ vi
Introduction .......................................................................................................................... viii
1 Scope ................................................................................................................................... 1
2 Conformance ........................................................................................................................ 1
3 Normative references ........................................................................................................... 1
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 ......................................................................................................... 9
7 Characteristics of decimal floating types <float.h> ............................................................. 10
8 Operation binding ................................................................................................................ 16
9 Conversions .......................................................................................................................... 17
  9.1 Conversions between decimal floating and integer types ............................................... 17
  9.2 Conversions among decimal floating types, and between decimal floating and standard
      floating types .................................................................................................................. 17
  9.3 Conversions between decimal floating and complex types ........................................... 18
  9.4 Usual arithmetic conversions ......................................................................................... 18
  9.5 Default argument promotion ......................................................................................... 19
10 Constants ................................................................................................................................ 19
11 Arithmetic operations ......................................................................................................... 20
  11.1 Operators ...................................................................................................................... 20
  11.2 Functions ...................................................................................................................... 20
  11.3 Conversions ................................................................................................................... 22
  11.4 Expression transformations ......................................................................................... 22
12 Library ................................................................................................................................... 22
  12.1 Standard headers .......................................................................................................... 22
  12.2 Decimal floating-point environment in <fenv.h> ............................................................. 22
  12.3 Decimal mathematics in <math.h> ................................................................................ 27
  12.4 Decimal-only functions in <math.h> .......................................................................... 37
    12.4.1 Quantum and quantum exponent functions .......................................................... 37
    12.4.2 Decimal re-encoding functions ............................................................................ 39
  12.5 Formatted input/output specifiers ................................................................................ 41
  12.6 stdtolV functions in <stdlib.h> ................................................................................... 44
  12.7 wcstolV functions in <wchar.h> ................................................................................... 47
  12.8 strfomdV functions in <stdlib.h> ................................................................................ 49
  12.9 Type-generic math for decimal in <tgmath.h> ............................................................... 50
Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO’s adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC 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 Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C:

— Part 1: Binary floating-point arithmetic
— Part 2: Decimal floating-point arithmetic

The following parts are under preparation:

— Part 3: Interchange and extended types
— Part 4: Supplementary functions
— Part 5: Supplementary attributes


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 the following:

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 the following:

- 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; 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 ISO/IEC/IEEE 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 its 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 the exchange of floating-point data 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.


**Purpose**

The purpose of ISO/IEC TS 18661 is to provide a C language binding for ISO/IEC/IEEE 60559:2011, based on the C11 standard, that delivers the goals of ISO/IEC/IEEE 60559 to users and is feasible to implement. It is organized into five parts.

ISO/IEC TS 18661-1 provides changes to C11 that cover all the requirements, plus some basic recommendations, of ISO/IEC/IEEE 60559:2011 for binary floating-point arithmetic. C implementations intending to support ISO/IEC/IEEE 60559:2011 are expected to conform to conditionally normative annex F as enhanced by the changes in ISO/IEC TS 18661-1.


ISO/IEC TS 18661-3 (Interchange and extended types), ISO/IEC TS 18661-4 (Supplementary functions), and ISO/IEC TS 18661-5 (Supplementary attributes) cover recommended features of ISO/IEC/IEEE 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/IEEE 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 might 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 this importance. Decimal floating-point formats and arithmetic are major new features in the IEEE 754-2008 standard and its international equivalent ISO/IEC/IEEE 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 part of ISO/IEC TS 18661 does not cover binary floating-point arithmetic (which is covered in ISO/IEC TS 18661-1), nor does it cover most optional features of IEC 60559.

2 Conformance

An implementation conforms to this part of ISO/IEC TS 18661 if

a) it meets the requirements for a conforming implementation of C11 with all the changes to C11 specified in ISO/IEC TS 18661-1 and in this part of ISO/IEC TS 18661; and

b) it defines __STDC_IEC_60559_DFP__ to 201504L.

NOTE Conformance to this part of ISO/IEC TS 18661 does not include all the requirements of ISO/IEC TS 18661-1. An implementation may conform to either or both of ISO/IEC TS 18661-1 and ISO/IEC TS 18661-2.

3 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.


4 Terms and definitions


5 C standard conformance

5.1 Freestanding implementations

The following change to C11 + TS18661-1 expands the conformance requirements for freestanding implementations so that they might conform to this part of ISO/IEC TS 18661.

Change to C2X + TS18661-1 (2019-02-11):

Replace the first sentence of 4#7:

The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines __STDC_IEC_60559_BFP__ 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>.

with:

The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines __STDC_IEC_60559_BFP__ or __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 + TS18661-1 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 ISO/IEC TS 18661-1. 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 ISO/IEC TS 18661-2.

Change to C2X + TS18661-1 (2019-02-11):

In 6.10.8.3#1, add:

__STDC_IEC_60559_DFP__ The integer constant 201504L intended to indicate support of decimal floating types and conformance with annex F for IEC 60559 decimal floating-point arithmetic.

The following change to C11 + TS18661-1 specifies the applications of annex F to binary and decimal floating-point arithmetic.
Change to C2X + TS18661-1 (2019-02-11):

Replace F.1#3:

[3] An implementation that defines \texttt{__STDC\_IEC\_60559\_BFP__} to 201404L shall conform to the specifications in this annex. Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

with:

[3] An implementation that defines \texttt{__STDC\_IEC\_60559\_BFP__} to 201404L shall conform to the specifications in this annex for binary floating-point arithmetic.

[4] An implementation that defines \texttt{__STDC\_IEC\_60559\_DFP__} to 201504L shall conform to the specifications for decimal floating-point arithmetic in the following subclauses of this annex:

— F.2.1 Infinities and NaNs
— F.3 Operations
— 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 \texttt{<math.h>}

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

[5] Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

5.3 Standard headers

The new identifiers added to C11 library headers by this part of ISO/IEC TS 18661 are defined or declared by their respective headers only if \texttt{__STDC\_WANT\_IEC\_60559\_DFP\_EXT__} is defined as a macro at the point in the source file where the appropriate header is first included. The macro \texttt{__STDC\_WANT\_IEC\_60559\_DFP\_EXT__} replaces the macro \texttt{__STDC\_WANT\_DEC\_FP__} specified in TR 24732 for the same purpose. The following changes to C11 + TS18661-1 list these identifiers in each applicable library subclause.
Changes to C2X + TS18661-1 (2019-02-11):

In 5.2.4.2.1#1, change:

[1] The following identifiers are defined only if \_\_STDC\_WANT\_IEC\_60559\_BFP\_EXT\_ is defined as a macro at the point in the source file where <limits.h> is first included:

to:

[1] The following identifiers are defined only if \_\_STDC\_WANT\_IEC\_60559\_BFP\_EXT\_ or \_\_STDC\_WANT\_IEC\_60559\_DFP\_EXT\_ is defined as a macro at the point in the source file where <limits.h> is first included:

After 5.2.4.2.2#8, insert the paragraph:

[8a] The following identifiers are defined only if \_\_STDC\_WANT\_IEC\_60559\_DFP\_EXT\_ is defined as a macro at the point in the source file where <float.h> is first included:

for \(N = 32, 64, \) and 128:

<table>
<thead>
<tr>
<th>DEC(_N) MANT DIG</th>
<th>DEC(_N) MAX</th>
<th>DEC(_N) TRUE MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC(_N) MIN EXP</td>
<td>DEC(_N) EPSILON</td>
<td></td>
</tr>
<tr>
<td>DEC(_N) MAX EXP</td>
<td>DEC(_N) MIN</td>
<td></td>
</tr>
</tbody>
</table>

After 7.6#5, insert the paragraph:

[5a] The following identifiers are declared only if \_\_STDC\_WANT\_IEC\_60559\_DFP\_EXT\_ is defined as a macro at the point in the source file where <fenv.h> is first included:

\texttt{fe\_dec\_getround} \hspace{1cm} \texttt{fe\_dec\_setround}

Change 7.12#2 from:

[2] The following identifiers are defined or declared only if \_\_STDC\_WANT\_IEC\_60559\_BFP\_EXT\_ is defined as a macro at the point in the source file where <math.h> is first included:

\hspace{1cm} \texttt{FP\_INT\_UPWARD} \hspace{1cm} \texttt{FP\_FAST\_FSUBL}
\hspace{1cm} \texttt{FP\_INT\_DOWNWARD} \hspace{1cm} \texttt{FP\_FAST\_FSUBL}
\hspace{1cm} \texttt{FP\_INT\_TOWARDZERO} \hspace{1cm} \texttt{FP\_FAST\_DSUBL}
\hspace{1cm} \texttt{FP\_INT\_TONEARESTFROMZERO} \hspace{1cm} \texttt{FP\_FAST\_FMUL}
\hspace{1cm} \texttt{FP\_INT\_TONEAREST} \hspace{1cm} \texttt{FP\_FAST\_FMULL}
\hspace{1cm} \texttt{FP\_LLOGBO} \hspace{1cm} \texttt{FP\_FAST\_DMULL}
\hspace{1cm} \texttt{FP\_LLOGBNAN} \hspace{1cm} \texttt{FP\_FAST\_FDIV}
\hspace{1cm} \texttt{SNANF} \hspace{1cm} \texttt{FP\_FAST\_FDIVL}
\hspace{1cm} \texttt{SNAN} \hspace{1cm} \texttt{FP\_FAST\_DDIVL}
\hspace{1cm} \texttt{SNANL} \hspace{1cm} \texttt{FP\_FAST\_FSQRT}
\hspace{1cm} \texttt{FP\_FAST\_FADD} \hspace{1cm} \texttt{FP\_FAST\_FSQRTL}
\hspace{1cm} \texttt{FP\_FAST\_FADDL} \hspace{1cm} \texttt{FP\_FAST\_DSQRTL}
\hspace{1cm} \texttt{FP\_FAST\_DADDL}
iseqsig                  fmaxmagf                  ffma
iscanonical             fmaxmagl                  dfma
issignaling             fminmag                  fsqrt
issubnormal             fminmagf                  fsqrtl
iszero                  fminmagl                  dsqrtl
fromfp                  nextup                   totalorder
fromfpf                 nextupf                  totalorderf
fromfpl                  nextupl                  totalorderl
ufromfp                  nextdown                 totalordermag
ufromfpf                nextdownf                totalordermagf
ufromfpl                 nextdownl                totalordermagl
fromfp                   fadd                     canonicalize
fromfpfx                 faddl                    canonicalizef
fromfpxl                 daddl                    canonicalizel
ufromfp                   fsub                     getpayload
ufromfpfx                fsubl                    getpayloadf
ufromfpxl                dsUBL                    getpayloadl
roundeven                fmul                     setpayload
roundevenf               fmul                     setpayloadf
roundevenl               dmull                    setpayloadl
llogb                    fdiv                     setpayloadsig
llogbf                   fdivl                    setpayloadsigf
llogbl                   ddvl                     setpayloadsigl
fmaxmag                  ffma

to:

[2] The following identifiers are defined only if __STDC_WANT_IEC_60559_BFP_EXT__ or __STDC_WANT_IEC_60559_DFP_EXT__ is defined as a macro at the point in the source file where <math.h> is first included:

FP_INT_UPWARD            FP_LLOGBNAN
FP_INT_DOWNWARD          iseqsig
FP_INT_TOWARDZERO        iscanonical
FP_INT_TONEARESTFROMZERO issignaling
FP_INT_TONEAREST          issubnormal
FP_LLOGB0                iszero
[2a] The following identifiers are defined or declared only if

```
__STDC_WANT_IEC_60559_BFP_EXT__ is defined as a macro at the point in the source file
where `<math.h>` is first included:
```

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNANF</td>
<td>ufromfpxf</td>
<td>dmul</td>
</tr>
<tr>
<td>SNAN</td>
<td>ufromfpxl</td>
<td>fdiv</td>
</tr>
<tr>
<td>SNANL</td>
<td>roundeven</td>
<td>fdivl</td>
</tr>
<tr>
<td>FP FAST FADD</td>
<td>roundevenf</td>
<td>ddivl</td>
</tr>
<tr>
<td>FP FAST FADDL</td>
<td>llogb</td>
<td>fdivl</td>
</tr>
<tr>
<td>FP FAST DADDL</td>
<td>llogbf</td>
<td>dfmal</td>
</tr>
<tr>
<td>FP FAST FSUBL</td>
<td>llogbl</td>
<td>fsqrt</td>
</tr>
<tr>
<td>FP FAST DSUBL</td>
<td>fmaxmag</td>
<td>fsqlr</td>
</tr>
<tr>
<td>FP FAST FMUL</td>
<td>fmaxmagl</td>
<td>totalorder</td>
</tr>
<tr>
<td>FP FAST FMULL</td>
<td>fminmag</td>
<td>totalorderf</td>
</tr>
<tr>
<td>FP FAST DMULL</td>
<td>fminmagl</td>
<td>totalorderl</td>
</tr>
<tr>
<td>FP FAST FDIV</td>
<td>fminmagf</td>
<td>totalordermag</td>
</tr>
<tr>
<td>FP FAST FDIVL</td>
<td>nextup</td>
<td>totalordermagf</td>
</tr>
<tr>
<td>FP FAST FSQRT</td>
<td>nextupf</td>
<td>totalordermagf</td>
</tr>
<tr>
<td>FP FAST FSQRTL</td>
<td>nextupl</td>
<td>canonicalize</td>
</tr>
<tr>
<td>FP FAST DSQRTL</td>
<td>nextdown</td>
<td>canonicalizef</td>
</tr>
<tr>
<td>fromfp</td>
<td>nextdownf</td>
<td>canonicalizel</td>
</tr>
<tr>
<td>fromfpf</td>
<td>nextdownfl</td>
<td>getpayload</td>
</tr>
<tr>
<td>fromfp1</td>
<td>fadd</td>
<td>getpayloadf</td>
</tr>
<tr>
<td>ufromfp</td>
<td>faddl</td>
<td>getpayloadl</td>
</tr>
<tr>
<td>ufromfpf</td>
<td>daddl</td>
<td>setpayload</td>
</tr>
<tr>
<td>ufromfpl</td>
<td>fsub</td>
<td>setpayloadf</td>
</tr>
<tr>
<td>fromfp</td>
<td>fsubl</td>
<td>setpayloadl</td>
</tr>
<tr>
<td>fromfpf</td>
<td>dsubl</td>
<td>setpayloadsig</td>
</tr>
<tr>
<td>fromfpfl</td>
<td>fmul</td>
<td>setpayloadsigf</td>
</tr>
<tr>
<td>ufromfpfl</td>
<td>fmul</td>
<td>setpayloadsigl</td>
</tr>
</tbody>
</table>

[2b] The following identifiers are defined or declared only if

```
__STDC_WANT_IEC_60559_DFP_EXT__ is defined as a macro at the point in the source file
where `<math.h>` is first included:
```

<table>
<thead>
<tr>
<th>Identifier</th>
<th>DEC_32</th>
<th>DEC_INFINITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal32 t</td>
<td></td>
<td>DEC_INFINITY</td>
</tr>
<tr>
<td>Decimal64 t</td>
<td></td>
<td>DEC NAN</td>
</tr>
</tbody>
</table>
and for $N = 32, 64, 128$:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE VAL DN</td>
<td>modfdN</td>
</tr>
<tr>
<td>SNANDN</td>
<td>remainderdN</td>
</tr>
<tr>
<td>FP FAST FMADDN</td>
<td>fabsdN</td>
</tr>
<tr>
<td>acosdN</td>
<td>nandN</td>
</tr>
<tr>
<td>asindN</td>
<td>nextafterdN</td>
</tr>
<tr>
<td>atandN</td>
<td>nextrtowarddN</td>
</tr>
<tr>
<td>atan2dN</td>
<td>nextupdN</td>
</tr>
<tr>
<td>cosdN</td>
<td>canonicalizedN</td>
</tr>
<tr>
<td>sindN</td>
<td>fdimdN</td>
</tr>
<tr>
<td>tandN</td>
<td>erfcdN</td>
</tr>
<tr>
<td>acoshdN</td>
<td>fmaxdN</td>
</tr>
<tr>
<td>asinhdN</td>
<td>fmindN</td>
</tr>
<tr>
<td>atanhdN</td>
<td>fminmagdN</td>
</tr>
<tr>
<td>coshdN</td>
<td>floordN</td>
</tr>
<tr>
<td>sinhN</td>
<td>fmadN</td>
</tr>
<tr>
<td>tanhdN</td>
<td>nearbyintdN</td>
</tr>
<tr>
<td>expdN</td>
<td>totalorderdN</td>
</tr>
<tr>
<td>exp2dN</td>
<td>totalordermagdN</td>
</tr>
<tr>
<td>expm1dN</td>
<td>lrintdN</td>
</tr>
<tr>
<td>frexpdN</td>
<td>getpayloaddN</td>
</tr>
<tr>
<td>ilogbdN</td>
<td>quantizedN</td>
</tr>
<tr>
<td>llrounddN</td>
<td>samequantumdN</td>
</tr>
<tr>
<td>lllogbdN</td>
<td>quantumdN</td>
</tr>
<tr>
<td>lldexpdN</td>
<td>llquantexpdN</td>
</tr>
<tr>
<td>logdN</td>
<td>fromfpdN</td>
</tr>
<tr>
<td>log10dN</td>
<td>encodedecn</td>
</tr>
<tr>
<td>log1pdN</td>
<td>decodedecn</td>
</tr>
<tr>
<td>log2dN</td>
<td>encodebindN</td>
</tr>
<tr>
<td>logbdN</td>
<td>fmoddN</td>
</tr>
</tbody>
</table>

and for $(M, N) = (32,64), (32,128), (64,128)$:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP FAST DMADDN</td>
<td>FP FAST DMFMADN</td>
</tr>
<tr>
<td>FP FAST DSMSUBD</td>
<td>dMmuldN</td>
</tr>
<tr>
<td>FP FAST DMMLDN</td>
<td>FP FAST DMSQRTD</td>
</tr>
<tr>
<td>FP FAST DMDIVDN</td>
<td>dMdivdN</td>
</tr>
<tr>
<td>FP FAST DMMLDN</td>
<td>dMadddN</td>
</tr>
<tr>
<td>FP FAST DMSQRTDN</td>
<td>dMfmadN</td>
</tr>
</tbody>
</table>

In 7.20#5, change:

[5] The following identifiers are defined only if __STDC_WANT_IEC_60559_BFP_EXT__ is defined as a macro at the point in the source file where `<stdint.h>` is first included:

to:

[5] The following identifiers are defined only if __STDC_WANT_IEC_60559_BFP_EXT__ or __STDC_WANT_IEC_60559_DFP_EXT__ is defined as a macro at the point in the source file where `<stdint.h>` is first included:
After 7.22#2, insert the paragraph:

[2a] The following identifiers are declared only if  \_\_STDC\_\_WANT\_\_IEC\_\_60559\_\_DFP\_\_EXT\_\_ is defined as a macro at the point in the source file where <stdlib.h> is first included:

\begin{verbatim}
strfromd32  strfromd128  strtod64
strfromd64  strtod32  strtod128
\end{verbatim}

Change 7.25#2 from:

[2] The following identifiers are defined as type-generic macros only if \_\_STDC\_\_WANT\_\_IEC\_\_60559\_\_BFP\_\_EXT\_\_ is defined as a macro at the point in the source file where <tgmath.h> is first included:

\begin{verbatim}
roundeven  fromfpx  fmul
llogb      ufro\texttt{m}fpx  dmul
fmaxmag    totalorder  fdiv
fminmag    totalordermag  ddiv
nextup     fadd  ffma
nextdown   dadd  dfma
fromfp     fsub  fsqrt
ufro\texttt{m}fp     dsub  dsqrt
\end{verbatim}

[2a] The following identifiers are defined as type-generic macros only if \_\_STDC\_\_WANT\_\_IEC\_\_60559\_\_BFP\_\_EXT\_\_ or \_\_STDC\_\_WANT\_\_IEC\_\_60559\_\_DFP\_\_EXT\_\_ is defined as a macro at the point in the source file where <tgmath.h> is first included:

\begin{verbatim}
roundeven  nextup  fromfpx
llogb      nextdown  ufro\texttt{m}fpx
fmaxmag    fromfp  totalorder
fminmag    ufro\texttt{m}fp  totalordermag
\end{verbatim}

[2a] The following identifiers are defined as type-generic macros only if \_\_STDC\_\_WANT\_\_IEC\_\_60559\_\_BFP\_\_EXT\_\_ is defined as a macro at the point in the source file where <tgmath.h> is first included:

\begin{verbatim}
fadd  fmul  ffma
dadd  dmul  dfma
fsub  fdiv  fsqrt
dsub  ddiv  dsqrt
\end{verbatim}

[2b] The following identifiers are defined as type-generic macros only if \_\_STDC\_\_WANT\_\_IEC\_\_60559\_\_DFP\_\_EXT\_\_ is defined as a macro at the point in the source file where <tgmath.h> is first included:

\begin{verbatim}
d32add  d64add  quantize
d32sub  d64sub  samequantum
d32mul  d64mul  quantum
d32div  d64div  llquantexp
d32fma  d64fma
\end{verbatim}

\begin{verbatim}
d32sqrt  d64sqrt
\end{verbatim}
6 Decimal floating types

This part of ISO/IEC TS 18661 introduces three decimal floating types, designated as _Decimal32, _Decimal64 and _Decimal128. These types support the IEC 60559 decimal formats: decimal32, decimal64, and decimal128.

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

This part of ISO/IEC TS 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. Regardless of the representation, the decimal floating types are distinct from the types float, double, and long double.

NOTE This part of ISO/IEC TS 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 C2X + TS18661-1 (2019-02-11):

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. Respectively, they have the IEC 60559 formats: decimal32, decimal64, and decimal128. Decimal floating types are real floating types.

[10b] The standard floating types and the decimal floating types are collectively called the real floating types.

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

they have the IEC 60559 formats: decimal32

where the footnote is:

*) IEC 60559 specifies decimal32 as a data-interchange format that does not require arithmetic support; however, _Decimal32 is a fully supported arithmetic type.
Add the following to 6.4.1 Keywords:

\[\text{keyword:} \]

\[\text{Decimal32} \]
\[\text{Decimal64} \]
\[\text{Decimal128} \]

Add the following to 6.7.2 Type specifiers:

\[\text{type-specifier:} \]

\[\text{Decimal32} \]
\[\text{Decimal64} \]
\[\text{Decimal128} \]

Add the following bullets in 6.7.2#2 Constraints:

\[\text{— Decimal32} \]
\[\text{— Decimal64} \]
\[\text{— Decimal128} \]

Add the following after 6.7.2#3:

\[\text{[3a] The type specifiers } \text{Decimal32, Decimal64, and Decimal128} \text{ shall not be used if the implementation does not support decimal floating types (see 6.10.8.3).} \]

Add the following after 6.5#8:

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

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:

\[\text{— _Decimal32} \text{ is a decimal32 format, which is encoded in 32 bits} \]
\[\text{— _Decimal64} \text{ is a decimal64 format, which is encoded in 64 bits} \]
\[\text{— _Decimal128} \text{ is a decimal128 format, which is encoded in 128 bits} \]

The value of a finite number is given by \((-1)\text{sign} \times \text{significand} \times 10^{\text{exponent}}\). 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. The table below shows these characteristics by type:
### Format characteristics

<table>
<thead>
<tr>
<th>Type</th>
<th>Decimal32</th>
<th>Decimal64</th>
<th>Decimal128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Significand length in digits</td>
<td>7</td>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>Maximum Exponent ($E_{\text{max}}$)</td>
<td>97</td>
<td>385</td>
<td>6145</td>
</tr>
<tr>
<td>Minimum Exponent ($E_{\text{min}}$)</td>
<td>−94</td>
<td>−382</td>
<td>−6142</td>
</tr>
</tbody>
</table>

The maximum and minimum exponents in the table 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_60559_DFP_EXT__` 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.

**Changes to C2X + TS18661-1 (2019-02-11):**

In 5.2.4.2.2#8, append the sentence:

Decimal floating-point operations have stricter requirements.

In 5.2.4.2.2#9, change:

All except `CR_DECIMAL_DIG` (F.5), `DECIMAL_DIG`, `FLT_EVAL_METHOD`, `FLT_RADIX`, and `FLT_ROUNDS` have separate names for all three floating-point types. The floating-point model representation is provided for all values except `FLT_EVAL_METHOD` and `FLT_ROUNDS`.

to:

All except `CR_DECIMAL_DIG` (F.5), `DECIMAL_DIG`, `DEC_EVAL_METHOD`, `FLT_EVAL_METHOD`, `FLT_RADIX`, and `FLT_ROUNDS` have separate names for all real floating types. The floating-point model representation is provided for all values except `DEC_EVAL_METHOD`, `FLT_EVAL_METHOD`, and `FLT_ROUNDS`.

After 5.2.4.2.2#9, insert the paragraph:

[9a] The remainder of this subclause specifies characteristics of standard floating types.

In 5.2.4.2.2#10, change:

[10] The rounding mode for floating-point addition is characterized by the implementation-defined value of `FLT_ROUNDS`.

to:

[10] The rounding mode for floating-point addition for standard floating types is characterized by the implementation-defined value of `FLT_ROUNDS`.
Add the following after 5.2.4.2.2:

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

[1] This subclause specifies macros in `<float.h>` that provide characteristics of decimal 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] `DEC_EVAL_METHOD` is the decimal floating-point analogue of `FLT_EVAL_METHOD` (5.2.4.2.2). Its implementation-defined value characterizes the use of evaluation formats for decimal floating types:

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 types, this value is implementation-defined and is specified by the macro `FLT_RADIX`. For the decimal floating 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 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_DEC32_MANT_DIG</td>
<td>7</td>
</tr>
<tr>
<td>_DEC64_MANT_DIG</td>
<td>16</td>
</tr>
<tr>
<td>_DEC128_MANT_DIG</td>
<td>34</td>
</tr>
</tbody>
</table>

--- minimum exponent

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_DEC32_MIN_EXP</td>
<td>-94</td>
</tr>
<tr>
<td>_DEC64_MIN_EXP</td>
<td>-382</td>
</tr>
<tr>
<td>_DEC128_MIN_EXP</td>
<td>-6142</td>
</tr>
</tbody>
</table>

--- maximum exponent

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_DEC32_MAX_EXP</td>
<td>97</td>
</tr>
<tr>
<td>_DEC64_MAX_EXP</td>
<td>385</td>
</tr>
<tr>
<td>_DEC128_MAX_EXP</td>
<td>6145</td>
</tr>
</tbody>
</table>
maximum representable finite decimal floating-point number (there are 6, 15 and 33 9's after the decimal points respectively)

<table>
<thead>
<tr>
<th>Type</th>
<th>Decimal32</th>
<th>Decimal64</th>
<th>Decimal128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Quantum Exponent ($q_{max}$)</td>
<td>90</td>
<td>369</td>
<td>6111</td>
</tr>
<tr>
<td>Minimum Quantum Exponent ($q_{min}$)</td>
<td>-101</td>
<td>-398</td>
<td>-6176</td>
</tr>
</tbody>
</table>

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

minimum normalized positive decimal floating-point number

minimum positive subnormal decimal floating-point number

[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:

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

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 $c = f_0 f_1 f_2 \ldots$ an integer between 0 and $b^e - 1$ inclusive. Thus, $x = s \times c^* \ b^q$ 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 $b^q$.

Quantum exponent ranges

[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). The table 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 exponential, 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] The following table shows, for each operation, how the preferred quantum exponents of the operands, Q(x), Q(y), etc., determine the preferred quantum exponent of the operation result:
### Preferred quantum exponents

<table>
<thead>
<tr>
<th>Decimal operation (shown without suffixes)</th>
<th>Preferred quantum exponent of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>roundeven, round, trunc, ceil, floor, rint, nearbyint</td>
<td>(\max(Q(x), 0))</td>
</tr>
<tr>
<td>nextup, nextdown, nextafter, nexttoward</td>
<td>least possible</td>
</tr>
<tr>
<td>remainder</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>fmin, fmax, fminmag, fmaxmag</td>
<td>(Q(x)) if (x) gives the result, (Q(y)) if (y) gives the result</td>
</tr>
<tr>
<td>scalbn, scalbln</td>
<td>(Q(x) + n)</td>
</tr>
<tr>
<td>ldexp</td>
<td>(Q(x) + \exp)</td>
</tr>
<tr>
<td>logb</td>
<td>0</td>
</tr>
<tr>
<td>+, d32add, d64add</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>-, d32sub, d64sub</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>*, d32mul, d64mul</td>
<td>(Q(x) + Q(y))</td>
</tr>
<tr>
<td>/, d32div, d64div</td>
<td>(Q(x) - Q(y))</td>
</tr>
<tr>
<td>sqrt, d32sqrt, d64sqrt</td>
<td>floor((Q(x)/2))</td>
</tr>
<tr>
<td>fma, d32fma, d64fma</td>
<td>(\min(Q(x) + Q(y), Q(z)))</td>
</tr>
<tr>
<td>conversion from integer type</td>
<td>0</td>
</tr>
<tr>
<td>exact conversion from non-decimal floating type</td>
<td>0</td>
</tr>
<tr>
<td>inexact conversion from non-decimal floating type</td>
<td>least possible</td>
</tr>
<tr>
<td>conversion between decimal floating types</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>*(x) returned by canonicalize</td>
<td>(Q(*x))</td>
</tr>
<tr>
<td>strtod, wcstod, scanf, floating constants of decimal floating type</td>
<td>see 7.22.1.3a</td>
</tr>
<tr>
<td>-((x))</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>fabs</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>copysign</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>quantize</td>
<td>(Q(y))</td>
</tr>
<tr>
<td>quantum</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>*(x)ptr returned by encodedec, encodebin</td>
<td>(Q(*x)ptr)</td>
</tr>
<tr>
<td>*(x)ptr returned by decodedec, decodebin</td>
<td>(Q(*)xptr)</td>
</tr>
<tr>
<td>fmod</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>fdim</td>
<td>(\min((Q(x), Q(y))) if (x &gt; y), 0 if (x \leq y)</td>
</tr>
<tr>
<td>cbrt</td>
<td>floor((Q(x)/3))</td>
</tr>
<tr>
<td>hypot</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>pow</td>
<td>floor((y \times Q(x)))</td>
</tr>
<tr>
<td>modf</td>
<td>(Q(\text{value}))</td>
</tr>
<tr>
<td>*(x)ptr returned by modf</td>
<td>(\max(Q(\text{value}), 0))</td>
</tr>
<tr>
<td>frexp</td>
<td>(Q(\text{value})) if (\text{value} = 0), (- (\text{length of coefficient of } \text{value})) otherwise</td>
</tr>
<tr>
<td>*(x)ptr returned by setpayload</td>
<td>0 if (\text{pl}) does not represent a valid payload, not applicable otherwise (NaN returned)</td>
</tr>
<tr>
<td>setpayloadsig</td>
<td>0 if (*x) is a NaN, unspecified otherwise</td>
</tr>
<tr>
<td>getpayload</td>
<td>0</td>
</tr>
<tr>
<td>transcendental functions</td>
<td>0</td>
</tr>
</tbody>
</table>
8 Operation binding

The table and subsequent text in F.3 as specified in ISO/IEC TS 18661-1, with the further change below, show how the C decimal operations specified in this document, ISO/IEC TS 18661-2, provide the operations required by IEC 60559 for decimal floating-point arithmetic.

Change to C2X + TS18661-1 (2019-02-11):

After F.3#12, append the following:

[13] Decimal versions of the C remquo function are not provided. (The C decimal remainder functions provide the remainder operation defined by IEC 60559.)

[14] The C quantizeN functions (7.12.11a.1) provide the quantize operation defined in IEC 60559 for decimal floating-point arithmetic.

[15] The binding for the convertFormat operation applies to all conversions among IEC 60559 formats. Therefore, for implementations that conform to annex E, 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.

[16] IEC 60559 specifies the convertFromHexCharacter and convertToHexCharacter operations only for binary floating-point arithmetic.

[17] The C integer constant 10 provides the radix operation defined in IEC 60559 for decimal floating-point arithmetic.

[18] The C samequantumN functions (7.12.11a.2) provide the sameQuantum operation defined in IEC 60559 for decimal floating-point arithmetic.

[19] 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. The macros (7.6) FE_DEC_DOWNWARD, FE_DEC_TONEAREST, FE_DEC_TONEARESTFROMZERO, FE_DEC_TOWARDZERO, and FE_DEC_UPWARD, which are used in conjunction with the fe_dec_getround and fe_dec_setround functions, represent the IEC 60559 rounding-direction attributes roundTowardNegative, roundTiesToEven, roundTiesToAway, roundTowardZero, and roundTowardPositive, respectively.

[20] The C quantumN (7.12.11a.3) and llquantexpN (7.12.11a.4) functions compute the quantum and the (quantum) exponent q defined in IEC 60559 for decimal numbers viewed as having integer significands.

[21] The C encodeN (7.12.11b.1) and decodeN (7.12.11b.2) functions provide the encodeDecimal and decodeDecimal operations defined in IEC 60559 for decimal floating-point arithmetic.

[22] The C encodebindN (7.12.11b.3) and decodebindN (7.12.11b.4) 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 (F.3 and F.4 define the behavior). To help writing portable code, this part of ISO/IEC TS 18661 provides defined behavior for decimal floating types.

Changes to C2X + TS18661-1 (2019-02-11):

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 following 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

In the following change to C11 + TS18661-1, 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, the specification requires correct rounding. It also requires correct rounding for conversions from standard to decimal floating types. The specification in annex F requires correct rounding for conversions from decimal to the standard floating types that conform to IEC 60559.
Change to C2X + TS18661-1 (2019-02-11):

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 C2X + TS18661-1 (2019-02-11):

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

If one operand has decimal floating type, the other operand 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 \texttt{_Decimal64}, the other operand is converted to \texttt{_Decimal64}.

Otherwise, if the type of either operand is \texttt{_Decimal32}, the other operand is converted to \texttt{_Decimal32}.

If there are no decimal floating types in the operands:

\textit{Editor: The rest of 6.3.1.8#1 "First, if the corresponding real type of either operand is \texttt{long double}, the other operand is converted, without \ldots" remains the same except for indents.}

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: \texttt{df} and \texttt{DF} for \texttt{_Decimal32}, \texttt{dd} and \texttt{DD} for \texttt{_Decimal64}, and \texttt{dl} and \texttt{DL} for \texttt{_Decimal128}.

This specification does not carry forward two features introduced in TR 24732: the \texttt{FLOAT\_CONST\_DECIMAL64} pragma and the \texttt{d} and \texttt{D} suffixes for floating constants. The pragma changed the interpretation of unsuffixed floating constants between \texttt{double} and \texttt{_Decimal64}. The suffixes provided a way to designate \texttt{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 \texttt{d} and \texttt{D} suffixes for other purposes.

Changes to C2X + TS18661-1 (2019-02-11):

Change \texttt{floating-suffix} in 6.4.4.2 from:

\begin{verbatim}
    floating-suffix: one of
        f  l  F  L
\end{verbatim}

to:

\begin{verbatim}
    floating-suffix: one of
        f  l  F  L  \texttt{df} \texttt{dd} \texttt{dl} \texttt{DF} \texttt{DD} \texttt{DL}
\end{verbatim}

Add the following after 6.4.4.2#2:

\textbf{Constraints}

\begin{verbatim}
[2a] A floating-suffix \texttt{df}, \texttt{dd}, \texttt{dl}, \texttt{DF}, \texttt{DD}, or \texttt{DL} shall not be used in a
hexadecimal-floating-constant.
\end{verbatim}

Add the following paragraph after 6.4.4.2#4:

\begin{verbatim}
[4a] If a floating constant is suffixed by \texttt{df} or \texttt{DF}, it has type \texttt{_Decimal32}. If suffixed by \texttt{dd} or \\texttt{DD}, it has type \texttt{_Decimal64}. If suffixed by \texttt{dl} or \texttt{DL}, it has type \texttt{_Decimal128}.
\end{verbatim}
Add the following paragraph after 6.4.4.2#5:

[5a] Floating constants of decimal floating type 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 (7.12.1.3a) 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 C2X + TS18661-1 (2019-02-11):

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, or 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, or 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, or imaginary type.

Add the following after 6.5.15#3:

[3a] If either of the second or third operands has decimal floating type, the other operand shall not have standard floating type, complex type, or 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, or imaginary type.

11.2 Functions

The headers and library supply a number of functions and function-like macros that support 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 ISO/IEC TS 18661-1, 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`,
- `isgreateror_equal`, `isless`, `islessequal` and `islessgreater`.

From the `<math.h>` extensions specified in ISO/IEC TS 18661-1, the decimal floating-point versions of:

- `roundeven`, `nextup`, `nextdown`, `fminmag`, `fmaxmag`, `lllogb`, `fadd`, `faddl`, `daddl`, `fsub`, `fsubl`, `dsubl`, `fmul`, `fmull`, `dmull`, `fdiv`, `fdivl`, `ddivl`, `fsqrt`, `fsqrrtl`, `dsqrtrl`, `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:


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

- `feraiseexcept`, `feclearexcept`, `fetestexcept`, `fesetexceptflag`, `fegetexceptflag`, `fesetenv`, `feupdateenv`, and `feholdexcept`.

From the `<fenv.h>` extensions specified in ISO/IEC TS 18661-1, facilities dealing with decimal context:

- `fetestexceptflag`, `fesetexcept`, `fegetmode`, and `fesetmode`.

From the `<fenv.h>` extensions specified in this part of ISO/IEC TS 18661, facilities dealing with decimal context:

- `fe_dec_getround` and `fe_dec_setround`.

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 ISO/IEC TS 18661-1, the decimal floating-point versions of:

- `strtod` and `wcstod`.

From the `<stdlib.h>` extensions specified in ISO/IEC TS 18661-1, 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 + TS18661-1 alert implementors that some expression transformations must be avoided in order to preserve the quantum exponent (7) of decimal floating-point numbers.

Changes to C2X + TS18661-1 (2019-02-11):

In F.9.2, insert before paragraph #1:

[0] Valid expression transformations must preserve numerical values.

In F.9.2, insert at the beginning of paragraph #1:

[1] The equivalences noted below apply to expressions of standard floating types.

In F.9.2, append:

[2] For expressions of decimal floating types, transformations must preserve quantum exponents, as well as numerical values (5.2.4.2.2a).

[3] EXAMPLE 1. × x -> x is valid for decimal floating-point expressions x, but 1.0 × x -> x is not:

\[ 1.0 \times 12.34 = (1, 10) \times (1, 1234, -2) = (1, 12340, -3) = 12.340 \]

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_60559_DFP_EXT__ is defined at the point in the source file where the appropriate header is first included.

12.2 Decimal floating-point environment in <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 (binary) 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.

### Rounding mode macros

<table>
<thead>
<tr>
<th>For decimal floating types</th>
<th>For standard floating types</th>
<th>IEC 60559</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE_DEC_TOWARDZERO</td>
<td>FE_TOWARDZERO</td>
<td>Toward zero</td>
</tr>
<tr>
<td>FE_DEC_TONEAREST</td>
<td>FE_TONEAREST</td>
<td>To nearest, ties to even</td>
</tr>
<tr>
<td>FE_DEC_UPWARD</td>
<td>FE_UPWARD</td>
<td>Toward plus infinity</td>
</tr>
<tr>
<td>FE_DEC_DOWNWARD</td>
<td>FE_DOWNWARD</td>
<td>Toward minus infinity</td>
</tr>
<tr>
<td>FE_DEC_TONEARESTFROMZERO</td>
<td>n/a</td>
<td>To nearest, ties away from zero</td>
</tr>
</tbody>
</table>

5 **Changes to C2X + TS18661-1 (2019-02-11):**

Add the following after 7.6#9:

> [9a] Decimal floating-point operations and IEC 60559 binary floating-point operations (annex F) access the same floating-point exception status flags.

In 7.6#12, delete the sentence (and retain footnote 218 at the end of the paragraph):

10 **The defined macros expand to integer constant expressions whose values are distinct nonnegative values.**

Add the following after 7.6#12:

> [12a] Each of the macros

    ```plaintext
    FE_DEC_DOWNWARD
    FE_DEC_TONEAREST
    FE_DEC_TONEARESTFROMZERO
    FE_DEC_TOWARDZERO
    FE_DEC_UPWARD
    ```

20 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_DEC_ROUND` rounding control pragma (7.6.1b) 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, unless stated otherwise. The macros expand to integer constant expressions whose values are distinct nonnegative values.

> [12b] 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`.

30 > [12c] At program startup the dynamic rounding direction mode for decimal floating-point arithmetic is initialized to `FE_DEC_TONEAREST`.

In 7.6.2#2, change the first sentence from:

> The `FENV_ROUND` pragma provides a means to specify a constant rounding direction for floating-point operations within a translation unit or compound statement.

> The `FENV_ROUND` pragma provides a means to specify a constant rounding direction for floating-point operations within a translation unit or compound statement.
The **FENV_ROUND** pragma provides a means to specify a constant rounding direction for floating-point operations for standard floating types within a translation unit or compound statement.

In 7.6.2#3, change the first sentence from:

**direction** shall be one of the rounding-direction macro names defined in 7.6, or **FE_DYNAMIC**.

To:

**direction** shall be one of the names of the supported rounding direction macros for operations for standard floating types (7.6), or **FE_DYNAMIC**.

In 7.6.2#4, replace the first sentence:

Within the scope of an **FENV_ROUND** directive establishing a mode other than **FE_DYNAMIC**, all floating-point operators, ...

With:

The **FENV_ROUND** directive affects operations for standard floating types. Within the scope of an **FENV_ROUND** directive establishing a mode other than **FE_DYNAMIC**, floating-point operators, ...

In 7.6.2#4, change the table title from:

Functions affected by constant rounding modes

to:

Functions affected by constant rounding modes – for standard floating types

In 7.6.2#4, change the sentence following the table from:

Each `<math.h>` function listed in the table above indicates the family of functions of all supported types (for example, **acosf** and **acosl** as well as **acos**).

to:

Each `<math.h>` function listed in the table above indicates the family of functions of all standard floating types (for example, **acosf** and **acosl** as well as **acos**).

In 7.6.2#4, change the last sentence before the table from:

Floating constants (6.4.4.2) that occur in the scope of a constant rounding mode shall be interpreted according to that mode.

to:

Floating constants (6.4.4.2) of a standard floating type that occur in the scope of a constant rounding mode shall be interpreted according to that mode.
After 7.6.2, insert:

7.6.2a Decimal rounding control pragma

Synopsis

[1] #define __STDC_WANT_IEC_60559_DFP_EXT__
#include <fenv.h>
#pragma STDC FENV_DEC_ROUND dec-direction

Description

[2] The FENV_DEC_ROUND pragma is a decimal floating-point analogue of the FENV_ROUND pragma. If FLT_RADIX is not 10, the FENV_DEC_ROUND pragma affects operators, functions, and floating constants only for decimal floating types. The affected functions are listed in the table below. If FLT_RADIX is 10, whether the FENV_ROUND and FENV_DEC_ROUND pragmas alter the rounding direction of both standard and decimal floating-point operations is implementation-defined. dec-direction shall be one of the decimal rounding direction macro names (FE_DEC_DOWNWARD, FE_DEC_TONEAREST, FE_DEC_TONEARESTFROMZERO, FE_DEC_TOWARDZERO, and FE_DEC_UPWARD) defined in 7.6, to specify a constant rounding mode, or FE_DEC_DYNAMIC, to specify dynamic rounding. The corresponding dynamic rounding mode can be established by a call to fe_dec_setround.

Functions affected by constant rounding modes – for decimal floating types

<table>
<thead>
<tr>
<th>Header</th>
<th>Function groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;math.h&gt;</td>
<td>acosdN, asindN, atandN, atan2dN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>cosdN, sindN, tandN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>acoshdN, asinhdN, atanhdN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>coshdN, sinhN, tanhdN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>expdN, exp2dN, expm1dN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>logdN, log10dN, log1pdN, log2dN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>scalbNdN, scalblndN, ldexpN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>cbrtdN, hypotdN, powdN, sqrtldN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>erfNdN, erfcdN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>lgammadN, lgammadN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>rintdN, nearbyintdN, lrintdN, llrintdN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>quantizedN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>fdimdN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>fmadN</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>dMadddN, dMsubdN, dMmuldN, dMdivdN, dMfmadN,</td>
</tr>
<tr>
<td></td>
<td>dMsqrtdN</td>
</tr>
<tr>
<td>&lt;stdlib.h&gt;</td>
<td>strfromdN, strtodN</td>
</tr>
<tr>
<td>&lt;wchar.h&gt;</td>
<td>wcstodN</td>
</tr>
<tr>
<td>&lt;stdio.h&gt;</td>
<td>printf and scanf families</td>
</tr>
<tr>
<td>&lt;wchar.h&gt;</td>
<td>wprintf and wscanf families</td>
</tr>
</tbody>
</table>
Add the following after 7.6.4.2:

### 7.6.4.2a The fe_dec_getround function

**Synopsis**

```c
#define __STDC_WANT_IEC_60559_DFP_EXT__
#include <fenv.h>
int fe_dec_getround(void);
```

**Description**

[2] The `fe_dec_getround` function gets the current value of the dynamic rounding direction mode for decimal floating-point operations.

**Returns**

[3] The `fe_dec_getround` function returns the value of the rounding direction macro representing the current dynamic 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.

### 7.6.4.2b The fe_dec_setround function

**Synopsis**

```c
#define __STDC_WANT_IEC_60595_DFP_EXT__
#include <fenv.h>
int fe_dec_setround(int round);
```

**Description**

[2] The `fe_dec_setround` function sets the dynamic rounding direction mode for decimal floating-point operations to be the rounding direction represented by its argument `round`. If the argument is not equal to the value of a decimal 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 decimal rounding direction macro (that is, if and only if the dynamic rounding direction mode for decimal floating-point operations was set to the requested rounding direction).
12.3 Decimal mathematics in <math.h>

The list of types, macros, and functions specified in the mathematics library is extended to handle decimal floating 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 ISO/IEC TS 18661-1 (14.1, 14.2, 14.3, 14.4, 14.5, 14.8, 14.9, and 14.0). 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 ISO/IEC TS 18661-1 (14.7) are also extended to handle decimal floating types. The same applies to all comparison macros specified in C11 (7.12.14) and in ISO/IEC TS 18661-1 (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 C2X + TS18661-1 (2019-02-11):

Add after 7.12#3:

3a] The types

```c
__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 after 7.12#4:

4a] The macro

```c
HUGE_VAL_D32
```

expands to a constant expression of type __Decimal32 representing positive infinity. The macros

```c
HUGE_VAL_D64
HUGE_VAL_D128
```

are respectively __Decimal64 and __Decimal128 analogues of HUGE_VAL_D32.

Add after 7.12#5:

5a] The macro

```c
DEC_INFINITY
```

expands to a constant expression of type __Decimal32 representing positive infinity.
Add after 7.12#6:

[6a] The macro

```
DEC_NAN
```

expands to a constant expression of type `__Decimal32` representing a quiet NaN.

Add after 7.12#5a:

[5b] The decimal signaling NaN macros

```
__SNAND32
__SNAND64
__SNAND128
```

each expands to 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 after 7.12#10:

[10a] The macros

```
__FP_FAST_FMAD32
__FP_FAST_FMAD64
__FP_FAST_FMAD128
```

are, respectively, `Decimal32`, `Decimal64`, and `Decimal128` analogues of `FP_FAST_FMA`.

Add after 7.12#11:

[11a] The macros

```
__FP_FAST_D32ADDD64
__FP_FAST_D32ADDD128
__FP_FAST_D64ADDD128
__FP_FAST_D32SUBD64
__FP_FAST_D32SUBD128
__FP_FAST_D64SUBD128
__FP_FAST_D32MULD64
__FP_FAST_D32MULD128
__FP_FAST_D64MULD128
__FP_FAST_D32DIVD64
__FP_FAST_D32DIVD128
__FP_FAST_D64DIVD128
__FP_FAST_D32FMAD64
__FP_FAST_D32FMAD128
__FP_FAST_D64FMAD128
__FP_FAST_D32SQRTD64
__FP_FAST_D32SQRTD128
__FP_FAST_D64SQRTD128
```

are decimal analogues of `FP_FAST_FADD`, `FP_FAST_FADDL`, `FP_FAST_DADDL`, etc.
Add the following list of function prototypes to the synopsis of the respective subclauses:

7.12.4 Trigonometric functions

```c
__Decimal32 acosd32(_Decimal32 x);
__Decimal64 acosd64(_Decimal64 x);
__Decimal128 acosd128(_Decimal128 x);

__Decimal32 asind32(_Decimal32 x);
__Decimal64 asind64(_Decimal64 x);
__Decimal128 asind128(_Decimal128 x);

__Decimal32 atand32(_Decimal32 x);
__Decimal64 atand64(_Decimal64 x);
__Decimal128 atand128(_Decimal128 x);

__Decimal32 atan2d32(_Decimal32 y, _Decimal32 x);
__Decimal64 atan2d64(_Decimal64 y, _Decimal64 x);
__Decimal128 atan2d128(_Decimal128 y, _Decimal128 x);

__Decimal32 cosd32(_Decimal32 x);
__Decimal64 cosd64(_Decimal64 x);
__Decimal128 cosd128(_Decimal128 x);

__Decimal32 sind32(_Decimal32 x);
__Decimal64 sind64(_Decimal64 x);
__Decimal128 sind128(_Decimal128 x);

__Decimal32 tand32(_Decimal32 x);
__Decimal64 tand64(_Decimal64 x);
__Decimal128 tand128(_Decimal128 x);
```

7.12.5 Hyperbolic functions

```c
__Decimal32 acoshd32(_Decimal32 x);
__Decimal64 acoshd64(_Decimal64 x);
__Decimal128 acoshd128(_Decimal128 x);

__Decimal32 asinh32(_Decimal32 x);
__Decimal64 asinh64(_Decimal64 x);
__Decimal128 asinh128(_Decimal128 x);

__Decimal32 atanh32(_Decimal32 x);
__Decimal64 atanh64(_Decimal64 x);
__Decimal128 atanh128(_Decimal128 x);

__Decimal32 coshd32(_Decimal32 x);
__Decimal64 coshd64(_Decimal64 x);
__Decimal128 coshd128(_Decimal128 x);

__Decimal32 sinh32(_Decimal32 x);
__Decimal64 sinh64(_Decimal64 x);
__Decimal128 sinh128(_Decimal128 x);
```
7.12.6 Exponential and logarithmic functions

Decimal32 tanhd32( Decimal32 x);
Decimal64 tanhd64( Decimal64 x);
Decimal128 tanhd128( Decimal128 x);

Decimal32 expd32( Decimal32 x);
Decimal64 expd64( Decimal64 x);
Decimal128 expd128( Decimal128 x);

Decimal32 exp2d32( Decimal32 x);
Decimal64 exp2d64( Decimal64 x);
Decimal128 exp2d128( Decimal128 x);

Decimal32 expm1d32( Decimal32 x);
Decimal64 expm1d64( Decimal64 x);
Decimal128 expm1d128( Decimal128 x);

Decimal32 frexp32( Decimal32 value, int *exp);
Decimal64 frexp64( Decimal64 value, int *exp);
Decimal128 frexp128( Decimal128 value, int *exp);

int ilogbd32( Decimal32 x);
int ilogbd64( Decimal64 x);
int ilogbd128( Decimal128 x);

Decimal32 ldexpd32( Decimal32 x, int exp);
Decimal64 ldexpd64( Decimal64 x, int exp);
Decimal128 ldexpd128( Decimal128 x, int exp);

long int llogbd32( Decimal32 x);
long int llogbd64( Decimal64 x);
long int llogbd128( Decimal128 x);

Decimal32 logd32( Decimal32 x);
Decimal64 logd64( Decimal64 x);
Decimal128 logd128( Decimal128 x);

Decimal32 log10d32( Decimal32 x);
Decimal64 log10d64( Decimal64 x);
Decimal128 log10d128( Decimal128 x);

Decimal32 log1pd32( Decimal32 x);
Decimal64 log1pd64( Decimal64 x);
Decimal128 log1pd128( Decimal128 x);

Decimal32 log2d32( Decimal32 x);
Decimal64 log2d64( Decimal64 x);
Decimal128 log2d128( Decimal128 x);

Decimal32 logbd32( Decimal32 x);
Decimal64 logbd64( Decimal64 x);
Decimal128 logbd128( Decimal128 x);
Decimal32 modfd32( Decimal32 value, Decimal32 *iptr);
Decimal64 modfd64( Decimal64 value, Decimal64 *iptr);
Decimal128 modfd128( Decimal128 value, Decimal128 *iptr);

Decimal32 scalbnd32( Decimal32 x, int n);
Decimal64 scalbnd64( Decimal64 x, int n);
Decimal128 scalbnd128( Decimal128 x, int n);

Decimal32 scalblnd32( Decimal32 x, long int n);
Decimal64 scalblnd64( Decimal64 x, long int n);
Decimal128 scalblnd128( Decimal128 x, long int n);

7.12.7 Power and absolute-value functions

Decimal32 cbrtd32( Decimal32 x);
Decimal64 cbrtd64( Decimal64 x);
Decimal128 cbrtd128( Decimal128 x);

Decimal32 fabsd32( Decimal32 x);
Decimal64 fabsd64( Decimal64 x);
Decimal128 fabsd128( Decimal128 x);

Decimal32 hypotd32( Decimal32 x, Decimal32 y);
Decimal64 hypotd64( Decimal64 x, Decimal64 y);
Decimal128 hypotd128( Decimal128 x, Decimal128 y);

Decimal32 powd32( Decimal32 x, Decimal32 y);
Decimal64 powd64( Decimal64 x, Decimal64 y);
Decimal128 powd128( Decimal128 x, Decimal128 y);

Decimal32 sqrtd32( Decimal32 x);
Decimal64 sqrtd64( Decimal64 x);
Decimal128 sqrtd128( Decimal128 x);

7.12.8 Error and gamma functions

Decimal32 erfd32( Decimal32 x);
Decimal64 erfd64( Decimal64 x);
Decimal128 erfd128( Decimal128 x);

Decimal32 erfcd32( Decimal32 x);
Decimal64 erfcd64( Decimal64 x);
Decimal128 erfcd128( Decimal128 x);

Decimal32 lgammad32( Decimal32 x);
Decimal64 lgammad64( Decimal64 x);
Decimal128 lgammad128( Decimal128 x);

Decimal32 tgammad32( Decimal32 x);
Decimal64 tgammad64( Decimal64 x);
Decimal128 tgammad128( Decimal128 x);
7.12.9 Nearest integer functions

```c
Decimal32 ceild32( Decimal32 x);
Decimal164 ceild64( Decimal164 x);
Decimal128 ceild128( Decimal128 x);

Decimal32 floord32( Decimal32 x);
Decimal164 floord64( Decimal164 x);
Decimal128 floord128( Decimal128 x);

Decimal32 nearbyintd32( Decimal32 x);
Decimal164 nearbyintd64( Decimal164 x);
Decimal128 nearbyintd128( Decimal128 x);

Decimal32 rintd32( Decimal32 x);
Decimal164 rintd64( Decimal164 x);
Decimal128 rintd128( Decimal128 x);

long int lrintd32( Decimal32 x);
long int lrintd64( Decimal164 x);
long int lrintd128( Decimal128 x);

long long int llrintd32( Decimal32 x);
long long int llrintd64( Decimal164 x);
long long int llrintd128( Decimal128 x);

Decimal32 roundd32( Decimal32 x);
Decimal164 roundd64( Decimal164 x);
Decimal128 roundd128( Decimal128 x);

long int lroundd32( Decimal32 x);
long int lroundd64( Decimal164 x);
long int lroundd128( Decimal128 x);

long long int llroundd32( Decimal32 x);
long long int llroundd64( Decimal164 x);
long long int llroundd128( Decimal128 x);

Decimal32 roundevend32( Decimal32 x);
Decimal164 roundevend64( Decimal164 x);
Decimal128 roundevend128( Decimal128 x);

Decimal32 truncd32( Decimal32 x);
Decimal164 truncd64( Decimal164 x);
Decimal128 truncd128( Decimal128 x);

intmax_t fromfpd32( Decimal32 x, int round, unsigned int width);
intmax_t fromfpd64( Decimal164 x, int round, unsigned int width);
intmax_t fromfpd128( Decimal128 x, int round, unsigned int width);
```

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

intmax_t fromfpxd32(_Decimal32 x, int round, unsigned int width);
intmax_t fromfpxd64(_Decimal64 x, int round, unsigned int width);
intmax_t fromfpxd128(_Decimal128 x, int round, unsigned int width);

uintmax_t ufromfpxd32(_Decimal32 x, int round, unsigned int width);
uintmax_t ufromfpxd64(_Decimal64 x, int round, unsigned int width);
uintmax_t ufromfpxd128(_Decimal128 x, int round, unsigned int width);

7.12.10 Remainder functions

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

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

7.12.11 Manipulation functions

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

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

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

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

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

_Decimal32 nextdownd32(_Decimal32 x);
_Decimal64 nextdownd64(_Decimal64 x);
_Decimal128 nextdownd128(_Decimal128 x);
7.12.12 Maximum, minimum, and positive difference functions

```c
int canonicalized32( Decimal32 * cx, const Decimal32 * x);
int canonicalized64( Decimal64 * cx, const Decimal64 * x);
int canonicalized128( Decimal128 * cx, const Decimal128 * x);
```

```c
Decimal32 fdimd32( Decimal32 x, Decimal32 y);
Decimal64 fdimd64( Decimal64 x, Decimal64 y);
Decimal128 fdimd128( Decimal128 x, Decimal128 y);
```

```c
Decimal32 fmaxd32( Decimal32 x, Decimal32 y);
Decimal64 fmaxd64( Decimal64 x, Decimal64 y);
Decimal128 fmaxd128( Decimal128 x, Decimal128 y);
```

```c
Decimal32 fmind32( Decimal32 x, Decimal32 y);
Decimal64 fmind64( Decimal64 x, Decimal64 y);
Decimal128 fmind128( Decimal128 x, Decimal128 y);
```

```c
Decimal32 fmaxmagd32( Decimal32 x, Decimal32 y);
Decimal64 fmaxmagd64( Decimal64 x, Decimal64 y);
Decimal128 fmaxmagd128( Decimal128 x, Decimal128 y);
```

```c
Decimal32 fminmagd32( Decimal32 x, Decimal32 y);
Decimal64 fminmagd64( Decimal64 x, Decimal64 y);
Decimal128 fminmagd128( Decimal128 x, Decimal128 y);
```

7.12.13 Floating multiply-add

```c
Decimal32 fmad32( Decimal32 x, Decimal32 y, Decimal32 z);
Decimal64 fmad64( Decimal64 x, Decimal64 y, Decimal64 z);
Decimal128 fmad128( Decimal128 x, Decimal128 y, Decimal128 z);
```

7.12.14 Functions that round result to narrower type

```c
Decimal32 d32add64( Decimal64 x, Decimal64 y);
Decimal32 d32add128( Decimal128 x, Decimal128 y);
Decimal64 d64add128( Decimal128 x, Decimal128 y);
```

```c
Decimal32 d32sub64( Decimal64 x, Decimal64 y);
Decimal32 d32sub128( Decimal128 x, Decimal128 y);
Decimal64 d64sub128( Decimal128 x, Decimal128 y);
```

```c
Decimal32 d32mul64( Decimal64 x, Decimal64 y);
Decimal32 d32mul128( Decimal128 x, Decimal128 y);
Decimal64 d64mul128( Decimal128 x, Decimal128 y);
```

```c
Decimal32 d32div64( Decimal64 x, Decimal64 y);
Decimal32 d32div128( Decimal128 x, Decimal128 y);
Decimal64 d64div128( 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.10.12 Total order functions

int totalorderd32(_Decimal32 x, _Decimal32 y);
int totalorderd64(_Decimal64 x, _Decimal64 y);
int totalorderd128(_Decimal128 x, _Decimal128 y);

int totalordermagd32(_Decimal32 x, _Decimal32 y);
int totalordermagd64(_Decimal64 x, _Decimal64 y);
int totalordermagd128(_Decimal128 x, _Decimal128 y);

F.10.13 Payload functions

Decimal32 getpayload32(const _Decimal32 *x);
Decimal64 getpayload64(const _Decimal64 *x);
Decimal128 getpayload128(const _Decimal128 *x);

int setpayload32(_Decimal32 *res, _Decimal32 pl);
int setpayload64(_Decimal64 *res, _Decimal64 pl);
int setpayload128(_Decimal128 *res, _Decimal128 pl);

int setpayloadsigd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadsigd64(_Decimal64 *res, _Decimal64 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 7.12.15#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 \(\text{value} = x \times 2^{\text{exp}}\). If \(\text{value}\) is zero, both parts of the result are zero.

with the following:

[2] The \(\text{frexp}\) functions break a floating-point number into a normalized fraction and an integer \(\text{exponent}\). They store the integer in the \text{int} object pointed to by \text{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 \(\text{value}\) is not a floating-point number or the integral power is outside the range of \text{int}, the results are unspecified. Otherwise, the \(\text{frexp}\) functions return the value \(x\), such that; \(x\) has a magnitude in the interval \([1/2, 1)\) or zero, and \(\text{value} = x \times 2^{\text{exp}}\), 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 \(\text{value} = x \times 10^{\text{exp}}\), when the type of the function is a decimal floating type. If \(\text{value}\) is zero, both parts of the result are zero.

Replace 7.12.6.6 paragraphs 2 and 3:

[2] The \(\text{ldexp}\) functions multiply a floating-point number by an integral power of 2. A range error may occur.

[3] The \(\text{ldexp}\) functions return \(x \times 2^{\text{exp}}\).

with the following:

[2] The \(\text{ldexp}\) functions multiply a floating-point number by an integral power of 2 \textit{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 \(\text{ldexp}\) functions return \(x \times 2^{\text{exp}}\) \textit{when the type of the function is a standard floating type, or return} \(x \times 10^{\text{exp}}\) \textit{when the type of the function is a decimal floating type.}

Replace 7.12.6.12#2:

[2] The \(\text{logb}\) functions extract the exponent of \(x\), as a signed integer value in floating-point format. If \(x\) is subnormal it is treated as though it were normalized; thus, for positive finite \(x\),

\[
1 \leq x \times \text{FLT\_RADIX}^{-\text{logb}(x)} < \text{FLT\_RADIX}
\]

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

with the following:

[2] The \(\text{logb}\) functions extract the exponent of \(x\), as a signed integer value in floating-point format. If \(x\) is subnormal it is treated as though it were normalized; thus, for positive finite \(x\),

\[
1 \leq x \times b^{-\logb(x)} < b
\]

where \(b = \text{FLT\_RADIX}\) \textit{if the type of the function is a standard floating type, or} \(b = 10\) \textit{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 7.12.6.14 paragraphs 2 and 3:

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

[3] The scalbn and scalbln functions return $x \times \text{FLT\_RADIX}^n$.

with the following:

[2] The scalbn and scalbln functions compute $x \times b^n$, where $b = \text{FLT\_RADIX}$ 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 $x \times b^n$.

12.4 Decimal-only functions in <math.h>

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

12.4.1 Quantum and quantum exponent functions

This specification does not carry forward the quantexpdN functions from TR 24732, which return the quantum exponent of their argument as an int. Instead it introduces the quantumdN functions, which return the quantum rather than the quantum exponent, and the llquantexpdN functions, which return the quantum exponent as a long long int, instead of int. The new interfaces offer natural extensions for support of wider IEC 60559 decimal formats in part 3 of ISO/IEC TS 18661.

Change to C2X + TS18661-1 (2019-02-11):

After subclause 7.12.14, add a new subclause:

7.12.14a Quantum and quantum exponent functions

7.12.14a.1 The quantizedN functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_DFP_EXT__
#include <math.h>

Decimal32 quantized32 ( Decimal32 x, Decimal32 y);
Decimal64 quantized64 ( Decimal64 x, Decimal64 y);
Decimal128 quantized128 ( Decimal128 x, Decimal128 y);

Description

[2] The quantizedN functions compute, if possible, a value with the numerical value of x and the quantum exponent of y. If the quantum exponent is being increased, the value shall be correctly rounded; 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, the “invalid” floating-point exception is raised, and a domain error occurs. If one or both operands are NaN the result is NaN. Otherwise if only one operand is infinite, the result is NaN, the “invalid” floating-point exception is raised, and a domain error occurs. If both operands are infinite, the result is NaN.
**DEC_INFINITY** with the sign of \( x \), converted to the type of the function. The quantize functions do not raise the "overflow" and "underflow" floating-point exceptions.

**Returns**

[3] The quantized functions return a value with the numerical value of \( x \) (except for any rounding) and the quantum exponent of \( y \).

### 7.12.14a.2 The samequantumd functions

**Synopsis**

[1] `#define __STDC_WANT_IEC_60559_DFP_EXT__`  
    include <math.h>  
    _Bool samequantumd32( _Decimal32 x, _Decimal32 y);  
    _Bool samequantumd64( _Decimal64 x, _Decimal64 y);  
    _Bool samequantumd128( _Decimal128 x, _Decimal128 y);

**Description**

[2] The samequantumd functions determine if the quantum exponents of \( x \) and \( y \) are the same. If both \( x \) and \( y \) are NaN, or both infinite, they have the same quantum exponents; if exactly one operand is infinite or exactly one operand is NaN, they do not have the same quantum exponents. The samequantumd functions raise no floating-point exception.

**Returns**

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

### 7.12.14a.3 The quantumd functions

**Synopsis**

[1] `#define __STDC_WANT_IEC_60559_DFP_EXT__`  
    include <math.h>  
    _Decimal32 quantumd32( _Decimal32 x);  
    _Decimal64 quantumd64( _Decimal64 x);  
    _Decimal128 quantumd128( _Decimal128 x);

**Description**

[2] The quantumd functions compute the quantum \((5.2.4.2.2a)\) of a finite argument. If \( x \) is infinite, the result is \(+\infty\).

**Returns**

[3] The quantumd functions return the quantum of \( x \).
7.12.14a.4 The \texttt{llquantexpdN} functions

**Synopsis**

[1] `#define _STDC_WANT_IEC_60559_DFP_EXT_
#include <math.h>

long long int llquantexpd32( _Decimal32 x);
long long int llquantexpd64( _Decimal64 x);
long long int llquantexpd128( _Decimal128 x);`

**Description**

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

**Returns**

[3] The \texttt{llquantexpdN} functions return the quantum exponent of \texttt{x}.

12.4.2 Decimal re-encoding functions

IEC 60559 defines two alternative encoding schemes for its decimal interchange formats: one based on decimal encoding of the significand, the other based on binary encoding of the significand. (See IEC 60559 for details.) The two encoding schemes encode the same values. The re-encoding functions in this subclause allow the user to convert data, in either of the encoding schemes, to and from values of the corresponding decimal floating type.

**Change to C2X + TS18661-1 (2019-02-11):**

After subclause 7.12.14a, add a new subclause:

7.12.14b Decimal re-encoding functions

7.12.14b.1 The \texttt{encodedecdN} functions

**Synopsis**

[1] `#define _STDC_WANT_IEC_60559_DFP_EXT_
#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 \texttt{encodedecdN} functions convert \texttt{xptr} into an IEC 60559 decimal\texttt{N} encoding in the encoding scheme based on decimal encoding of the significand and store the resulting encoding as an \texttt{N}/8 element array, with 8 bits per array element, in the object pointed to by \texttt{encptr}. The order of bytes in the array is implementation-defined. These functions preserve the value of \texttt{xptr} and raise no floating-point exceptions. If \texttt{xptr} is non-canonical, these functions may or may not produce a canonical encoding.
Returns


7.12.14b.2 The `encodedecdN` functions

Synopsis

[1] 

```c
#define _STDC_WANT_IEC_60559_DFP_EXT_
#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 `encodedecdN` 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 value of type `DecimalN`, 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


7.12.14b.3 The `encodebindN` functions

Synopsis

[1] 

```c
#define _STDC_WANT_IEC_60559_DFP_EXT_
#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 `encodebindN` 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


7.12.14b.4 The `decodebinN` functions

Synopsis

[1] #define _STDC_WANT_IEC_60559_DFP_EXT_
    #include <math.h>
    void decodebin32( _Decimal32 * restrict xptr, _
        const unsigned char * restrict encptr);
    void decodebin64( _Decimal64 * restrict xptr, _
        const unsigned char * restrict encptr);
    void decodebin128( _Decimal128 * restrict xptr, _
        const unsigned char * restrict encptr);

Description

[2] The `decodebinN` functions interpret the N/8 element array pointed to by `encptr` as an IEC 60559 decimalN 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 value of type `_DecimalN`, 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


12.5 Formatted input/output specifiers

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

Changes to C2X + TS18661-1 (2019-02-11):

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:

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

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

30   **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, at the end of the specification for aA 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 significant digits in the coefficient \(c\),

- if \(-(n+5) \leq q \leq 0\), use style f (or style F in the case of an A conversion specifier) with formatting precision equal to \(-q\),
- otherwise, use style e (or style E in the case of an A conversion specifier) 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 \(P\) 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 \(P\) is present (and nonzero) and less than the precision \(p\) of the decimal floating type, the conversion first obtains an intermediate result as follows, where \(n\) is the number of significant digits in the coefficient:

- If \(n \leq P\), set the intermediate result to the input.
- If \(n > P\), round the input value, according to the current rounding direction for decimal floating-point operations, to \(P\) decimal digits, with unbounded exponent range, representing the result with a \(P\)-digit integer coefficient when in the form \((s, c, q)\).

Convert the intermediate result in the manner described above for the case where the precision is missing.

**EXAMPLE 1** Following are representations of _Decimal64_ arguments as triples \((s, c, q)\) and the corresponding character sequences `printf` produces with \#Da:

<table>
<thead>
<tr>
<th>((s, c, q))</th>
<th>(\text{printf})</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1, 123, 0))</td>
<td>123</td>
</tr>
<tr>
<td>((-1, 123, 0))</td>
<td>-123</td>
</tr>
<tr>
<td>((1, 123, -2))</td>
<td>1.23e+3</td>
</tr>
<tr>
<td>((-1, 123, 1))</td>
<td>-1.23e+3</td>
</tr>
<tr>
<td>((1, 123, -8))</td>
<td>0.000000123</td>
</tr>
<tr>
<td>((1, 123, -9))</td>
<td>1.23e-7</td>
</tr>
<tr>
<td>((-1, 120, -8))</td>
<td>0.00000120</td>
</tr>
<tr>
<td>((-1, 120, -9))</td>
<td>1.20e-7</td>
</tr>
<tr>
<td>((1, 1234567890123456, 0))</td>
<td>1234567890123456</td>
</tr>
<tr>
<td>((1, 1234567890123456, 1))</td>
<td>1.234567890123456e+16</td>
</tr>
<tr>
<td>((1, 1234567890123456, -1))</td>
<td>123456789012345.6</td>
</tr>
<tr>
<td>((1, 1234567890123456, -21))</td>
<td>0.000001234567890123456</td>
</tr>
<tr>
<td>((1, 1234567890123456, -22))</td>
<td>1.234567890123456e-7</td>
</tr>
<tr>
<td>((1, 0, 0))</td>
<td>0</td>
</tr>
<tr>
<td>((-1, 0, 0))</td>
<td>0</td>
</tr>
<tr>
<td>((1, 0, -6))</td>
<td>0.000006</td>
</tr>
<tr>
<td>((1, 0, -7))</td>
<td>0e-7</td>
</tr>
</tbody>
</table>
{(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;  // (1, 654300, -2)
    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;  // (1, 9543210, 87)
    _Decimal32 y = 9500000e90DF;  // (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
         9.54e+93
         9.5e+93
          1e+94
          1e+97
EXAMPLE 4  To further illustrate the effects of the exponent range, the sequence:

```c
Decimal32 x = 9512345e90DF; // (1, 9512345, 90)
Decimal32 y = 9512345e86DF; // (1, 9512345, 86)
printf("%.3Ha\n", x);
printf("%.2Ha\n", x);
printf("%.1Ha\n", x);
printf("%.2Ha\n", y);
```

assuming default rounding, results in:

```
9.51e+96
9.5e+96
1e+97
9.5e+92
```

12.6 `strtodN` functions in `<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 C2X + TS18661-1 (2019-02-11):

After 7.22.1.4, add:

```
7.22.1.4a The strtodN functions
```

**Synopsis**

```
#define __STDC_WANT_IEC_60559_DFP_EXT__
#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 `strtodN` functions convert the initial portion of the string pointed to by `nptr` to `_DecimaldN` representation. 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 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

— \texttt{INF} or \texttt{INFINITY}, ignoring case

— \texttt{NAN} or \texttt{NAN}(\texttt{d-char-sequence}\textsubscript{opt}), ignoring case in the \texttt{NAN} part, where:

\begin{verbatim}
d-char-sequence:
  digit
  nondigit
  d-char-sequence digit
  d-char-sequence nondigit
\end{verbatim}

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 \texttt{INF} or \texttt{INFINITY} is interpreted as an infinity. A character sequence \texttt{NAN} or \texttt{NAN}(\texttt{d-char-sequence}\textsubscript{opt}), is interpreted as a quiet NaN; the meaning of the \texttt{d-char-sequence} is implementation-defined. A pointer to the final string is stored in the object pointed to by \texttt{endptr}, provided that \texttt{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 \texttt{fractional-constant} or \texttt{digit-sequence} and the \texttt{exponent-part} (if any) are extracted from the subject sequence. If there is an \texttt{exponent-part}, then \(q\) is set to the value of \texttt{sign}\textsubscript{opt}\texttt{digit-sequence} in the \texttt{exponent-part}. If there is no \texttt{exponent-part}, \(q\) is set to 0.

— If there is a \texttt{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 \texttt{digit-sequence}.

— \(c\) is set to the value of the \texttt{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 `strtof64`. Note that for `Decimal64`, the precision (maximum
coefficient length) is 16 and the quantum exponent range is \(-398 \leq q \leq 369\).

\[
\begin{align*}
"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) \\
"12.3E7" & (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) \\
"-0.00" & (-1, 0, -2) \\
"0E+7" & (1, 0) \\
"-0E-7" & (-1, 0, -7) \\
"12345678901234567890" & (1, 1234567890123457, 4) or (1, 1234567890123456, 4) \(\text{depending on rounding mode}\) \\
"1234E-400" & (1, 12, -398) or (1, 13, -398) \\
"1234E-402" & (1, 0, -398) or (1, 1, -398) \(\text{depending on rounding mode}\) \\
"1000." & (1, 1000) \\
".0001" & (1, -4) \\
"1000.e0" & (1, 1000) \\
".001e0" & (1, -4) \\
"1000.0" & (1, 10000, -1) \\
"0.0001" & (1, -4) \\
"1000.00" & (1, 100000, -2) \\
"0.0001" & (1, -4) \\
"001000." & (1, 1000) \\
"001000.0" & (1, 10000, -1) \\
"001000.00" & (1, 100000, -2) \\
"00.00" & (1, -2) \\
"00." & (1, 0) \\
".00" & (1, -2) \\
"00.00e-5" & (1, -7) \\
"00.e-5" & (1, -5) \\
".00e-5" & (1, -7) \\
"0x1.8p+4" & (1, 0), and a pointer to "x1.8p+4" is stored in the object pointed to by
\end{verbatim}

\begin{verbatim}
endptr, provided endptr is not a null pointer
\end{verbatim}

"infinite" infinity, and a pointer to "inite" is stored in the object pointed to by `endptr`,

\begin{verbatim}
provided endptr is not a null pointer
\end{verbatim}

[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 correctly rounded 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, the value of the macro `ERANGE` is stored in `errno`. If the result underflows (7.12.1), whether `errno` acquires the value `ERANGE` is implementation-defined.

In 7.22.1.4a#4, attach a footnote to the wording:

the meaning of the d-char sequence 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 wcstodN functions in `<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 C2X + TS18661-1 (2019-02-11):**

After 7.29.4.1.1, add:

#### 7.29.4.1.1a The wcstodN functions

**Synopsis**

[1] #define __STDC_WANT_IEC_60559_DFP_EXT__
#include `<wchar.h>`

```c
    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 `wcstodN` functions convert the initial portion of the wide string pointed to by `nptr` to `DecimalN` representation. 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 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
  
  nondigit
  
  d-wchar-sequence digit
  
  d-wchar-sequence nondigit

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 sequence 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 \( \text{nptr} \) is stored in the object pointed to by \( \text{endptr} \), provided that \( \text{endptr} \) 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 \text{HUGE\_VAL\_D32}, \text{HUGE\_VAL\_D64}, or \text{HUGE\_VAL\_D128} is returned (according to the return type and sign of the value), and the value of the macro \text{ERANGE} is stored in \( \text{errno} \). 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 \( \text{errno} \) acquires the value \text{ERANGE} is implementation-defined.

In 7.29.4.1.1a\#4, attach a footnote to the wording:

the meaning of the d-wchar sequence 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 \text{strfromdN} functions in <stdlib.h>

The specifications of these functions are similar to those of \text{strfromd}, \text{strfromf}, and \text{strfromld} (7.22.1.2a) as defined in subclause 10.2 of ISO/IEC TS 18661-1. These functions are declared in <stdlib.h>.

Change to C2X + TS18661-1 (2019-02-11):

After 7.22.1.3, add:

7.22.1.3a The \text{strfromdN} functions

Synopsis

[1] #define \_STDC\_WANT\_IEC\_60559\_DFP\_EXT\_  
#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

[2] The `strfroN` functions are equivalent to `snprintf(s, n, format, fp)` (7.21.6.5), except the `format` string contains only the character `%`, an optional precision that does not contain an asterisk `*`, 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

[3] The `strfroN` functions return the number of characters that would have been written had `n` been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is less than `n`.

12.9 Type-generic math for decimal in `<tgmath.h>`

The following changes to C11 + TS18661-1 enhance the specification of type-generic macros in `<tgmath.h>` to apply to decimal floating types, as well as standard floating types.

Changes to C2X + TS18661-1 (2019-02-11):

In 7.25, replace paragraphs 3 and 4:

[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 the functions that round result to narrower type (7.12.14) (which are covered below) and `modf`, there is a corresponding type-generic macro. 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`.

[4] 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:

[3] This clause specifies a many-to-one correspondence of functions in `<math.h>` and `<complex.h>` with type-generic macros. Use of a type-generic macro invokes a corresponding function whose type is determined by the types of the arguments for particular parameters called the `generic parameters`.

[4] 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 the functions that round result to narrower type
(7.12.14) (which are covered below) and \texttt{modf}, there is a corresponding type-generic macro.\texttt{313} The parameters whose corresponding real type is \texttt{double} in the function synopsis are generic parameters.

\textbf{[4a]} Some of the \texttt{<math.h>} functions for decimal floating types have no unsuffixed counterpart. Of these functions with a \texttt{d64} suffix, some have one or more parameters whose type is \texttt{Decimal64}. For each such function, except \texttt{decodedc64}, \texttt{encodedc64}, \texttt{decodedb64}, and \texttt{encodedb64}, there is a corresponding type-generic macro. The parameters whose real type is \texttt{Decimal64} in the function synopsis are generic parameters.

\textbf{[4b]} 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.

\textbf{[4c]} Except for the macros for functions that round result to a narrower type (7.12.14), use of a type-generic macro invokes a function whose generic parameters have the corresponding real type determined by the types of the arguments for the generic parameters as follows:

\begin{itemize}
  \item Arguments of integer type are regarded as having type \texttt{Decimal64} if any argument has decimal floating type, and as having type \texttt{double} otherwise.
  \item If the function has exactly one generic parameter, the type determined is the corresponding real type of the argument for the generic parameter.
  \item If the function has exactly two generic parameters, the type determined is the corresponding real type determined by the usual arithmetic conversions (6.3.1.8) applied to the arguments for the generic parameters.
  \item If the function has more than two generic parameters, the type determined is the corresponding real type determined by repeatedly applying the usual arithmetic conversions, first to the first two arguments for generic parameters, then to that result type and the next argument for a generic parameter, and so forth until the usual arithmetic conversions have been applied to the last argument for a generic parameter.
\end{itemize}

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

In 7.25\#6, replace the last sentence:

\begin{itemize}
  \item 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.
\end{itemize}

with:

\begin{itemize}
  \item If all arguments for generic parameters are real, then use of the macro invokes a real function \textit{(provided \texttt{<math.h>} defines a function of the determined type)}; otherwise, use of the macro results in undefined behavior.
\end{itemize}

In 7.25\#7, replace the last sentence:

\begin{itemize}
  \item Use of the macro with any \texttt{real} or \texttt{complex} argument invokes a complex function.
\end{itemize}
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 decimal floating type results in undefined behavior.

Change 7.25#8 from:

[8] The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any suffix 330) from the function names. Thus, the macros are:

<table>
<thead>
<tr>
<th>fadd</th>
<th>fmul</th>
<th>ffma</th>
</tr>
</thead>
<tbody>
<tr>
<td>dadd</td>
<td>dmul</td>
<td>dfma</td>
</tr>
<tr>
<td>fsub</td>
<td>fdiv</td>
<td>fsqrt</td>
</tr>
<tr>
<td>dsub</td>
<td>ddiv</td>
<td>dsqrt</td>
</tr>
</tbody>
</table>

All arguments shall be real. If any argument has type _Decimal128, or if the macro prefix is d, the function invoked has the name of the macro with an suffix. Otherwise, the function invoked has the name of the macro (with no suffix).

to:

[8] The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any suffix from the function names. Thus, the macros with f or d prefix are:

<table>
<thead>
<tr>
<th>fadd</th>
<th>fmul</th>
<th>ffma</th>
</tr>
</thead>
<tbody>
<tr>
<td>dadd</td>
<td>dmul</td>
<td>dfma</td>
</tr>
<tr>
<td>fsub</td>
<td>fdiv</td>
<td>fsqrt</td>
</tr>
<tr>
<td>dsub</td>
<td>ddiv</td>
<td>dsqrt</td>
</tr>
</tbody>
</table>

and the macros with d32 or d64 prefix are:

<table>
<thead>
<tr>
<th>d32add</th>
<th>d32mul</th>
<th>d32fma</th>
</tr>
</thead>
<tbody>
<tr>
<td>d64add</td>
<td>d64mul</td>
<td>d64fma</td>
</tr>
<tr>
<td>d32sub</td>
<td>d32div</td>
<td>d32sqrt</td>
</tr>
<tr>
<td>d64sub</td>
<td>d64div</td>
<td>d64sqrt</td>
</tr>
</tbody>
</table>

All arguments shall be real. If the macro prefix is f or d, use of an argument of decimal floating type results in undefined behavior. If the macro prefix is d32 or d64, use of an argument of standard floating type results in undefined behavior. The function invoked is determined as follows:

— If any argument has type _Decimal128, or if the macro prefix is d64, the function invoked has the name of the macro, with a d128 suffix.

— Otherwise, if the macro prefix is d32, the function invoked has the name of the macro, with a d64 suffix.

— Otherwise, if any argument has type long double, or if the macro prefix is d, the function invoked has the name of the macro, with an l suffix.

— Otherwise, the function invoked has the name of the macro (with no suffix).
After 7.25#8, add the paragraph:

[6a+] For each d64-suffixed function in <math.h>, except decodedcd64, encodedcd64, 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:

<table>
<thead>
<tr>
<th><code>&lt;math.h&gt;</code></th>
<th>type-generic function</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantizedN</td>
<td>quantize</td>
</tr>
<tr>
<td>samequantumdN</td>
<td>samequantum</td>
</tr>
<tr>
<td>quantumdN</td>
<td>quantum</td>
</tr>
<tr>
<td>llquantexpdN</td>
<td>llquantexp</td>
</tr>
</tbody>
</table>

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

In 7.25#11, insert at the beginning of the example:

```c
#define _STDC_WANT_IEC_60559_DFP_EXT_
```

In 7.25#11, append to the declarations:

```c
#if __STDC_IEC_60559_DFP__ >= 201504L
  _Decimal32 d32;
  _Decimal64 d64;
  _Decimal128 d128;
#endif
```

In 7.25#11, append to the table:

<table>
<thead>
<tr>
<th>exp(d64)</th>
<th>expd64(d64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt(d32)</td>
<td>sqrt(d32)</td>
</tr>
<tr>
<td>fmax(d64, d128)</td>
<td>fmax(d64, d128)</td>
</tr>
<tr>
<td>pow(d32, n)</td>
<td>pow(d32, n)</td>
</tr>
<tr>
<td>remainder(d64, d)</td>
<td>undefined behavior</td>
</tr>
<tr>
<td>creal(d64)</td>
<td>undefined behavior</td>
</tr>
<tr>
<td>remquo(d32, d32, &amp;n)</td>
<td>undefined behavior</td>
</tr>
<tr>
<td>llquantexp(d)</td>
<td>undefined behavior</td>
</tr>
<tr>
<td>quantize(dc)</td>
<td>undefined behavior</td>
</tr>
<tr>
<td>samequantum(n, n)</td>
<td>undefined behavior</td>
</tr>
<tr>
<td>d32sub(d32, d128)</td>
<td>d32sub(d32, d128)</td>
</tr>
<tr>
<td>d32div(d64, n)</td>
<td>d32div(d64, n)</td>
</tr>
<tr>
<td>d64fma(d32, d64, d128)</td>
<td>d64fma(d32, d64, d128)</td>
</tr>
<tr>
<td>d64add(d32, d32)</td>
<td>d64add(d32, d32)</td>
</tr>
<tr>
<td>d64sqrt(d)</td>
<td>undefined behavior</td>
</tr>
<tr>
<td>dadd(n, d64)</td>
<td>undefined behavior</td>
</tr>
</tbody>
</table>
Bibliography


