Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

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

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

ISO copyright office
Case postale 56 CH-1211 Geneva 20
Tel. +41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>vi</td>
</tr>
<tr>
<td>Background</td>
<td>v</td>
</tr>
<tr>
<td>IEC 60559 floating-point standard</td>
<td>v</td>
</tr>
<tr>
<td>C support for IEC 60559</td>
<td>vi</td>
</tr>
<tr>
<td>Purpose</td>
<td>vii</td>
</tr>
<tr>
<td>Additional background on formats</td>
<td>vii</td>
</tr>
<tr>
<td>1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>2 Conformance</td>
<td>1</td>
</tr>
<tr>
<td>3 Normative references</td>
<td>1</td>
</tr>
<tr>
<td>4 Terms and definitions</td>
<td>2</td>
</tr>
<tr>
<td>5 C standard conformance</td>
<td>2</td>
</tr>
<tr>
<td>5.1 Freestanding implementations</td>
<td>2</td>
</tr>
<tr>
<td>5.2 Predefined macros</td>
<td>2</td>
</tr>
<tr>
<td>5.3 Standard headers</td>
<td>2</td>
</tr>
<tr>
<td>6 Types</td>
<td>7</td>
</tr>
<tr>
<td>7 Characteristics</td>
<td>12</td>
</tr>
<tr>
<td>8 Conversions</td>
<td>17</td>
</tr>
<tr>
<td>9 Constants</td>
<td>19</td>
</tr>
<tr>
<td>10 Expressions</td>
<td>19</td>
</tr>
<tr>
<td>11 Non-arithmetic interchange formats</td>
<td>21</td>
</tr>
<tr>
<td>12 Mathematics &lt;math.h&gt;</td>
<td>21</td>
</tr>
<tr>
<td>12.1 Macros</td>
<td>23</td>
</tr>
<tr>
<td>12.2 Floating-point environment</td>
<td>26</td>
</tr>
<tr>
<td>12.3 Functions</td>
<td>28</td>
</tr>
<tr>
<td>12.4 Encoding conversion functions</td>
<td>37</td>
</tr>
<tr>
<td>13 Numeric conversion functions in &lt;stdlib.h&gt;</td>
<td>40</td>
</tr>
<tr>
<td>14 Complex arithmetic &lt;complex.h&gt;</td>
<td>46</td>
</tr>
<tr>
<td>15 Type-generic macros &lt;tgmath.h&gt;</td>
<td>48</td>
</tr>
<tr>
<td>30 Bibliography</td>
<td>52</td>
</tr>
</tbody>
</table>
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, 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
— Part 3: Interchange and extended types
— Part 4: Supplementary functions
— Part 5: Supplementary attributes


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

Introduction

Background

IEC 60559 floating-point standard

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

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

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

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

4. Provide direct support for
   a. Execution-time diagnosis of anomalies
   b. Smoother handling of exceptions
   c. Interval arithmetic at a reasonable cost

5. Provide for development of
   a. Standard elementary functions such as exp and cos
   b. Very high precision (multiword) arithmetic
   c. Coupling of numerical and symbolic algebraic computation

6. Enable rather than preclude further refinements and extensions.

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

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

- **operations** – basic arithmetic operations (addition, multiplication, etc.) on the format data to compose a well-defined, closed arithmetic system; 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 data that may be exchanged between different implementations that conform to the specification.

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

C support for IEC 60559

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

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


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.

Part 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 Part 1.

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

Part 3, this document, specifies types and other support for interchange and extended formats recommended in ISO/IEC/IEEE 60559:2011. C implementations intending to provide an extension for these formats are expected to conform to Part 3.

Part 4 specifies functions for operations recommended in ISO/IEC/IEEE 60559:2011. C implementations intending to provide an extension for these operations are expected to conform to Part 4.

Part 5 specifies support for attributes recommended in ISO/IEC/IEEE 60559:2011. C implementations intending to provide an extension for these attributes are expected to conform to Part 5.

Additional background on formats

The revised floating-point arithmetic standard, ISO/IEC/IEEE 60559:2011, introduces a variety of new formats, both fixed and extendable. The new fixed formats include

- a 128-bit basic binary format (the 32 and 64 bit basic binary formats are carried over from ISO/IEC 60559:1989)
- 64 and 128 bit basic decimal formats
- interchange formats, whose precision and range are determined by the width k, where
  - for binary, k = 16, 32, 64, and k ≥ 128 and a multiple of 32, and
  - for decimal, k ≥ 32 and a multiple of 32
- extended formats, for each basic format, with minimum range and precision specified

Thus IEC 60559 defines five basic formats - binary32, binary64, binary128, decimal64, and decimal128 - and five corresponding extended formats, each with somewhat more precision and range than the basic format it extends. IEC 60559 defines an unlimited number of interchange formats, which include the basic formats.

Interchange formats may or may not be supported as arithmetic formats. If not, they may be used for the interchange of floating-point data but not for arithmetic computation. IEC 60559 provides conversions between non-arithmetic interchange formats and arithmetic formats which can be used for computation.

Extended formats are intended for intermediate computation, not input or output data. The extra precision often allows the computation of extended results which when converted to a narrower output format differ from the ideal results by little more than a unit in the last place. Also, the extra range often avoids any intermediate overflow or underflow that might occur if the computation were done in the format of the data. The essential
The property of extended formats is their sufficient extra widths, not their specific widths. Extended formats for any given basic format may vary among implementations.

Extendable formats, which provide user control over range and precision, are not covered in ISO/IEC TS 18661.

The 32 and 64 bit binary formats are supported in C by types `float` and `double`. If a C implementation defines the macro `__STDC_IEC_60559_BFP__` (see ISO/IEC TS 18661-1) signifying that it supports C Annex F for binary floating-point arithmetic, then its `float` and `double` formats must be IEC 60559 binary32 and binary64.

ISO/IEC TS 18661-2 defines types `Decimal32`, `Decimal64`, and `Decimal128` with IEC 60559 formats decimal32, decimal64, and decimal128. Although IEC 60559 does not require arithmetic support (other than conversions) for its decimal32 interchange format, ISO/IEC TS 18661-2 has full arithmetic and library support for `Decimal32`, just like for `Decimal64` and `Decimal128`.

The C Standard provides just three standard floating types (`float`, `double`, and `long double`) that are required of all implementations. C Annex F for binary floating-point arithmetic requires the standard floating types to be binary. The `long double` type must be at least as wide as `double`, but C does not further specify details of its format, even in Annex F.

ISO/IEC TS 18661-3, this document, provides nomenclatures for types with IEC 60559 arithmetic interchange formats and extended formats. The nomenclatures allow portable use of the formats as envisioned in IEC 60559. This document covers these aspects of the types:

- names
- characteristics
- conversions
- constants
- function suffixes
- character sequence conversion interfaces

This specification includes interchange and extended nomenclatures for formats that, in some cases, already have C nomenclatures. For example, types with the IEC 60559 double format may include `double`, `_Float64` (the type for the binary64 interchange format), and maybe `_Float32x` (the type for the binary32-extended format). This redundancy is intended to support the different programming models appropriate for the types with arithmetic interchange formats and extended formats and C standard floating types.

This document also supports the IEC 60559 non-arithmetic interchange formats with functions that convert among encodings and between encodings and character sequences, for all interchange formats.
Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —

Part 3:
Interchange and extended types

1 Scope

This part of ISO/IEC TS 18661 extends programming language C to include types with the arithmetic interchange and extended floating-point formats specified in ISO/IEC/IEEE 60559:2011, and to include functions that support the non-arithmetic interchange formats in that standard.

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 parts 1-3 of ISO/IEC TS 18661;

b) It conforms to ISO/IEC TS 18661-1 or ISO/IEC TS 18661-2 (or both); and

c) It defines __STDC_IEC_60559_TYPES__ to 201ymmL.

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.

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

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


ISO/IEC 18661-2:yyyy, Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 2: Decimal floating-point arithmetic

Changes specified in this part of ISO/IEC TS 18661 are relative to ISO/IEC 9899:2011, including Technical Corrigendum 1 (ISO/IEC 9899:2011/Cor. 1:2012), together with the changes from parts 1 and 2 of ISO/IEC TS 18661.
4 Terms and definitions

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


5 C standard conformance

5.1 Freestanding implementations

The specification in C11 + TS18661-1 + TS18661-2 allows freestanding implementations to conform to this part of ISO/IEC TS 18661.

5.2 Predefined macros

Change to C11 + TS18661-1 + TS18661-2:

In 6.10.8.3#1, add:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 __STDC_WANT_IEC_60559_TYPES_EXT__ is defined as a macro at the point in the source file where the appropriate header is first included. The following changes to C11 + TS18661-1 + TS18661-2 list these identifiers in each applicable library subclause.

Changes to C11 + TS18661-1 + TS18661-2:

After 5.2.4.2.2#6b, insert the paragraph:

[6c] The following identifiers are defined only if __STDC_WANT_IEC_60559_TYPES_EXT__ is defined as a macro at the point in the source file where <float.h> is first included:

for supported types _FloatN:

<table>
<thead>
<tr>
<th>FLT_N_MANT_DIG</th>
<th>FLT_N_MIN_10_EXP</th>
<th>FLT_N_EPSILON</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_N_DECIMAL_DIG</td>
<td>FLT_N_MAX_EXP</td>
<td>FLT_N_MIN</td>
</tr>
<tr>
<td>FLT_N_DIG</td>
<td>FLT_N_MAX_10_EXP</td>
<td>FLT_N_TRUE_MIN</td>
</tr>
<tr>
<td>FLT_N_MIN_EXP</td>
<td>FLT_N_MAX</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

<table>
<thead>
<tr>
<th>FLT_NX_MANT_DIG</th>
<th>FLT_NX_MIN_10_EXP</th>
<th>FLT_NX_EPSILON</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_NX_DECIMAL_DIG</td>
<td>FLT_NX_MAX_EXP</td>
<td>FLT_NX_MIN</td>
</tr>
<tr>
<td>FLT_NX_DIG</td>
<td>FLT_NX_MAX_10_EXP</td>
<td>FLT_NX_TRUE_MIN</td>
</tr>
<tr>
<td>FLT_NX_MIN_EXP</td>
<td>FLT_NX_MAX</td>
<td></td>
</tr>
</tbody>
</table>
for supported types _DecimalN, where \( N \neq 32, 64, \) and 128:

<table>
<thead>
<tr>
<th>( \text{DEC}_N\text{ MANT_DIG} )</th>
<th>( \text{DEC}_N\text{ MAX} )</th>
<th>( \text{DEC}_N\text{ TRUE_MIN} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{DEC}_N\text{ MIN_EXP} )</td>
<td>( \text{DEC}_N\text{ EPSILON} )</td>
<td>( \text{DEC}_N\text{ MIN} )</td>
</tr>
</tbody>
</table>

5

for supported types _DecimalNx:

<table>
<thead>
<tr>
<th>( \text{DEC}_N\text{X MANT_DIG} )</th>
<th>( \text{DEC}_N\text{X MAX} )</th>
<th>( \text{DEC}_N\text{X TRUE_MIN} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{DEC}_N\text{X MIN_EXP} )</td>
<td>( \text{DEC}_N\text{X EPSILON} )</td>
<td>( \text{DEC}_N\text{X MIN} )</td>
</tr>
</tbody>
</table>

10 After 7.3#2, insert the paragraph:

[2a] The following identifiers are declared or defined only if __STDC_WANT_IEC_60559_TYPES_EXT__ is defined as a macro at the point in the source file where \(<\text{complex.h}>\) is first included:

for supported types _FloatN:

15

<table>
<thead>
<tr>
<th>cacosfN</th>
<th>ccatanhfN</th>
<th>csqrtfN</th>
</tr>
</thead>
<tbody>
<tr>
<td>casinfN</td>
<td>cccoshfN</td>
<td>cargfN</td>
</tr>
<tr>
<td>ccatanfN</td>
<td>csinhfN</td>
<td>cimagfN</td>
</tr>
<tr>
<td>ccoshfN</td>
<td>ctanhfN</td>
<td>CMPLXFN</td>
</tr>
<tr>
<td>csinfN</td>
<td>cexpfN</td>
<td>conjfN</td>
</tr>
<tr>
<td>ctanfN</td>
<td>clogfN</td>
<td>cprojfN</td>
</tr>
<tr>
<td>cacoshfN</td>
<td>cabsfN</td>
<td>crealfN</td>
</tr>
<tr>
<td>casinhfN</td>
<td>cpowfN</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

25

<table>
<thead>
<tr>
<th>cacosfNx</th>
<th>ccatanhfNx</th>
<th>csqrtfNx</th>
</tr>
</thead>
<tbody>
<tr>
<td>casinfNx</td>
<td>cccoshfNx</td>
<td>cargfNx</td>
</tr>
<tr>
<td>ccatanfNx</td>
<td>csinhfNx</td>
<td>cimagfNx</td>
</tr>
<tr>
<td>ccoshfNx</td>
<td>ctanhfNx</td>
<td>CMPLXFNX</td>
</tr>
<tr>
<td>csinfNx</td>
<td>cexpfNx</td>
<td>conjfNx</td>
</tr>
<tr>
<td>ctanfNx</td>
<td>clogfNx</td>
<td>cprojfNx</td>
</tr>
<tr>
<td>cacoshfNx</td>
<td>cabsfNx</td>
<td>crealfNx</td>
</tr>
<tr>
<td>casinhfNx</td>
<td>cpowfNx</td>
<td></td>
</tr>
</tbody>
</table>

After 7.12#1c, insert the paragraph:

[1d] The following identifiers are defined or declared only if __STDC_WANT_IEC_60559_TYPES_EXT__ is defined as a macro at the point in the source file where \(<\text{math.h}>\) is first included:

long double_t

for supported types _FloatN:

40

<table>
<thead>
<tr>
<th>_FloatN_t</th>
<th>log1pfN</th>
<th>fromfpfN</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_FN</td>
<td>log2fN</td>
<td>ufromfpfN</td>
</tr>
<tr>
<td>SNANfN</td>
<td>logbfN</td>
<td>fromfpfN</td>
</tr>
<tr>
<td>FP_FAST_FMAfN</td>
<td>modffN</td>
<td>ufromfpfxN</td>
</tr>
<tr>
<td>acosfN</td>
<td>scalbnfN</td>
<td>fmodfN</td>
</tr>
<tr>
<td>asinfN</td>
<td>scalb1nfN</td>
<td>remainderfN</td>
</tr>
</tbody>
</table>
atanfN cbrtfN remquofN
cosfN hypotfN
cbrtfN sqrtfN nextafterfN
sinfN powfN nextupfN
tanfN sqrtfN
acoshfN erfN nextrnptfN
sinhfN erfcN canonicalizefN
atanhfN lgammafN encodefN
coshfN tgammafN decodefN
sinhfN ceilfN
atan2fN floorfN
fabsfN intnfN
hypotfN fmaxfN
remquofN lrintfN fminfN
acoshfN erffN fmaxmagfN
asinhfN erfcpfN fminmagfN
atanhfN lgammafN
expfN tanhfN
expfN nearbyintfN totalorderN
expfN intnfN
cbrtfN fmaxfN
fexpfN lrintfN
acoshfN lrintfN
asinhfN fmaxfN
atanhfN lrintfN
exp2fN fmaxfN
exp2fN roundfN
cosfN floorfN
expfN totalorderfN
expfN roundfN
acoshfN remquofN
asinhfN canonicalizefN
atanhfN encodefN
coshfN decodefN
sinhfN setpayloadfN
atanhfN setpayloadsigfN
tanhfN setpayloadsigfN

for supported types _FloatN where $M < N$:

**FP_FAST_F ADCDF**

FP_FAST_F MULF

**FP_FAST_F MUDF**

FP_FAST_F ADDF

**FP_FAST_F MUDF**

FP_FAST_F SQRTF

**FP_FAST_F MUDF**

FP_FAST_F MULF

**FP_FAST_F MUDF**

FP_FAST_F SDIVF

**FP_FAST_FMUDF**

FP_FAST_FMULDF

**FP_FAST_FMUDF**

FP_FAST_FMSQRT

**FP_FAST_FMUDF**

FP_FAST_FMSQR

**FP_FAST_FMUDF**

FP_FAST_FMSQRT

for supported types _FloatM and _FloatN where $M < N$:
for supported types _FloatM and _FloatNx where $M \leq N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_FAST_FMAFXN</td>
<td>FMulfN</td>
</tr>
<tr>
<td>FP_FAST_FMSQRTFN</td>
<td>FMDivfN</td>
</tr>
<tr>
<td>FP_FAST_FMMULFN</td>
<td>FMAddfN</td>
</tr>
<tr>
<td>FP_FAST_FMDIVFN</td>
<td>FMSubfN</td>
</tr>
</tbody>
</table>

5

for supported types _FloatMx and _FloatN where $M < N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_FAST_FMXADDFN</td>
<td>FMXmulfN</td>
</tr>
<tr>
<td>FP_FAST_FMXSUBFN</td>
<td>FMXdivfN</td>
</tr>
<tr>
<td>FP_FAST_FMXMULFN</td>
<td>FMXAddfN</td>
</tr>
<tr>
<td>FP_FAST_FMXDIVFN</td>
<td>FMXSubfN</td>
</tr>
</tbody>
</table>

10

for supported types _FloatMx and _FloatNx where $M < N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_FAST_FMXADDFN</td>
<td>FMXmulfN</td>
</tr>
<tr>
<td>FP_FAST_FMXSUBFN</td>
<td>FMXdivfN</td>
</tr>
<tr>
<td>FP_FAST_FMXMULFN</td>
<td>FMXAddfN</td>
</tr>
<tr>
<td>FP_FAST_FMXDIVFN</td>
<td>FMXSubfN</td>
</tr>
</tbody>
</table>

15

for supported IEC 60559 arithmetic or non-arithmetic binary interchange formats of widths M and N:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMencfN</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _DecimalN, where $N \neq 32, 64, \text{and } 128$:

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DecimalN_t</td>
<td></td>
</tr>
<tr>
<td>HUGE_VAL_DN</td>
<td></td>
</tr>
<tr>
<td>SNANDN</td>
<td></td>
</tr>
<tr>
<td>FP_FAST_FMADN</td>
<td></td>
</tr>
<tr>
<td>acosdN</td>
<td></td>
</tr>
<tr>
<td>asindN</td>
<td></td>
</tr>
<tr>
<td>atan2dN</td>
<td></td>
</tr>
<tr>
<td>cosdN</td>
<td></td>
</tr>
<tr>
<td>sindN</td>
<td></td>
</tr>
<tr>
<td>tandN</td>
<td></td>
</tr>
<tr>
<td>acoshdN</td>
<td></td>
</tr>
<tr>
<td>asinhdN</td>
<td></td>
</tr>
<tr>
<td>atanhdN</td>
<td></td>
</tr>
<tr>
<td>coshdN</td>
<td></td>
</tr>
<tr>
<td>sinhhdN</td>
<td></td>
</tr>
<tr>
<td>tanhdN</td>
<td></td>
</tr>
<tr>
<td>expdN</td>
<td></td>
</tr>
<tr>
<td>exp2dN</td>
<td></td>
</tr>
<tr>
<td>expmldN</td>
<td></td>
</tr>
<tr>
<td>ffrexpN</td>
<td></td>
</tr>
<tr>
<td>ilogbdN</td>
<td></td>
</tr>
<tr>
<td>llrounddN</td>
<td></td>
</tr>
<tr>
<td>ldexpdN</td>
<td></td>
</tr>
<tr>
<td>logdN</td>
<td></td>
</tr>
<tr>
<td>log10dN</td>
<td></td>
</tr>
<tr>
<td>log1pN</td>
<td></td>
</tr>
<tr>
<td>log2dN</td>
<td></td>
</tr>
</tbody>
</table>

20

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logbdN</td>
<td></td>
</tr>
<tr>
<td>modfdN</td>
<td></td>
</tr>
<tr>
<td>scalbdN</td>
<td></td>
</tr>
<tr>
<td>scalblndN</td>
<td></td>
</tr>
<tr>
<td>cbrtdN</td>
<td></td>
</tr>
<tr>
<td>fabsdN</td>
<td></td>
</tr>
<tr>
<td>hypotdN</td>
<td></td>
</tr>
<tr>
<td>powdN</td>
<td></td>
</tr>
<tr>
<td>sqrtfdN</td>
<td></td>
</tr>
<tr>
<td>erfcdN</td>
<td></td>
</tr>
<tr>
<td>lgammadN</td>
<td></td>
</tr>
<tr>
<td>tgammadN</td>
<td></td>
</tr>
<tr>
<td>ceildN</td>
<td></td>
</tr>
<tr>
<td>floordN</td>
<td></td>
</tr>
<tr>
<td>nearbyintdN</td>
<td></td>
</tr>
<tr>
<td>rintdN</td>
<td></td>
</tr>
<tr>
<td>lrintdN</td>
<td></td>
</tr>
<tr>
<td>llrintdN</td>
<td></td>
</tr>
<tr>
<td>rounddN</td>
<td></td>
</tr>
<tr>
<td>lrounddN</td>
<td></td>
</tr>
<tr>
<td>roundevendN</td>
<td></td>
</tr>
<tr>
<td>totalorderdN</td>
<td></td>
</tr>
<tr>
<td>totalordermagdN</td>
<td></td>
</tr>
<tr>
<td>fromfpdN</td>
<td></td>
</tr>
<tr>
<td>ufromfpdN</td>
<td></td>
</tr>
<tr>
<td>fromfpdxN</td>
<td></td>
</tr>
<tr>
<td>ufromfpdxN</td>
<td></td>
</tr>
</tbody>
</table>

30

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmoddN</td>
<td></td>
</tr>
<tr>
<td>remainderdN</td>
<td></td>
</tr>
<tr>
<td>copysigndN</td>
<td></td>
</tr>
<tr>
<td>nandN</td>
<td></td>
</tr>
<tr>
<td>nextafterdN</td>
<td></td>
</tr>
<tr>
<td>nextupdN</td>
<td></td>
</tr>
<tr>
<td>canonicalizedN</td>
<td></td>
</tr>
<tr>
<td>quantizedN</td>
<td></td>
</tr>
<tr>
<td>samequantumedN</td>
<td></td>
</tr>
<tr>
<td>quantumedN</td>
<td></td>
</tr>
<tr>
<td>llquantexpdN</td>
<td></td>
</tr>
<tr>
<td>encodedecdN</td>
<td></td>
</tr>
<tr>
<td>decodedecdN</td>
<td></td>
</tr>
<tr>
<td>encodebindN</td>
<td></td>
</tr>
<tr>
<td>decodebindN</td>
<td></td>
</tr>
<tr>
<td>fdimdN</td>
<td></td>
</tr>
<tr>
<td>fmaxdN</td>
<td></td>
</tr>
<tr>
<td>fmindN</td>
<td></td>
</tr>
<tr>
<td>fmaxmagdN</td>
<td></td>
</tr>
<tr>
<td>fminmagdN</td>
<td></td>
</tr>
</tbody>
</table>

40

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>llrounddN</td>
<td></td>
</tr>
<tr>
<td>truncdN</td>
<td></td>
</tr>
<tr>
<td>rounddevendN</td>
<td></td>
</tr>
<tr>
<td>totalorderdN</td>
<td></td>
</tr>
<tr>
<td>totalordermagdN</td>
<td></td>
</tr>
<tr>
<td>getpayloaddN</td>
<td></td>
</tr>
<tr>
<td>setpayloaddN</td>
<td></td>
</tr>
<tr>
<td>setpayloadsifgN</td>
<td></td>
</tr>
</tbody>
</table>

45

© ISO/IEC 2014 – All rights reserved
for supported types \texttt{_DecimalNx}:

\begin{verbatim}
    HUGE_VAL_D/Nx  log2d/Nx    ufomfpd/Nx
    SNAND/Nx       logbd/Nx    fromfpd/Nx
    FP_FAST_FMAD/Nx modfd/Nx    ufomfpd/Nx
    acosdNx        scalbndNx   fmoddNx
    asindNx        scalblndNx  remainderdNx
    atandNx        cbrtdNx     copysigndNx
    atan2dNx       fabsdNx     nandNx
    cosdNx         hypotdNx    nextafterdNx
    sindNx         powdNx      nextupdNx
    tandNx         sqrtldNx    nextdowndNx
    acoshdNx       erfdNx      canonicalizedNx
    asinhdNx       erfcldNx    quantizedNx
    atanhdNx       lgammadNx   samequantumdNx
    coshdN         tgammaN     quantumdN
    sinhN          ceildN      lquantexpdN
    tanhdN         floorD      fdimdN
    expdN          nearbyintdN  fmaxdN
    exp2dN         rintdN      fmindN
    frexpN         lrintdN     fmaxmagdN
    ilogbn         rounddN     fmadN
    lllogbn        lrounddN    totalorderdN
    ldexpN         llrounddN   totalordermagdN
    logdN          truncdN     getpayloaddN
    log10dN        roundevenN  setpayloaddN
    log1pdN        fromfpdN    setpayloadsigdN
\end{verbatim}

for supported types \texttt{_DecimalM} and \texttt{_DecimalN} where \( M < N \) and \( M \) and \( N \) are not both one of 32, 64, and 128:

\begin{verbatim}
    FP_FAST_D/ADDN     FP_FAST_D/MFMADN  dMmuldN
    FP_FAST_D/MSUBN     FP_FAST_D/MSQRTDN  dMdivdN
    FP_FAST_D/MULDN     dMaddN     dMfmadN
    FP_FAST_D/MDIVD     dMsubdN     dMsqrtdN
\end{verbatim}

for supported types \texttt{_DecimalM} and \texttt{_DecimalNx} where \( M \leq N \):

\begin{verbatim}
    FP_FAST_D/ADDNX     FP_FAST_D/MFMADNX  dMmuldNx
    FP_FAST_D/MSUBNX    FP_FAST_D/MSQRTDNX  dMdivdNx
    FP_FAST_D/MULDNX     dMaddNx     dMfmadNx
    FP_FAST_D/MDIVDX     dMsubdNx     dMsqrtdNx
\end{verbatim}

for supported types \texttt{_DecimalMx} and \texttt{_DecimalN} where \( M < N \):

\begin{verbatim}
    FP_FAST_D/MADDN     FP_FAST_D/MXFADN  dMxmuldN
    FP_FAST_D/MSUBD     FP_FAST_D/MSQRTDN  dMxdivdN
    FP_FAST_D/MULDN     dMxadddN     dMxfmadN
    FP_FAST_D/MDIVDN     dMxsubdN     dMxsqrtdN
\end{verbatim}
for supported types _DecimalMx and _DecimalNx where $M < N$:

- `FP_FAST_D/MXADD/Nx` to `dMxmuldNx`
- `FP_FAST_D/MXSUBD/Nx` to `dMxdivdNx`
- `FP_FAST_D/MXULD/Nx` to `dMxadddNx`
- `FP_FAST_D/MXDIVD/Nx` to `dMxsubdNx`

for supported IEC 60559 arithmetic and non-arithmetic decimal interchange formats of widths $M$ and $N$:

- `dMencdec/N` to `dMencbind/N`

After 7.22#1b, insert the paragraph:

10 [1c] The following identifiers are declared only if `__STDC_WANT_IEC_60559_TYPES_EXT__` is defined as a macro at the point in the source file where `<stdlib.h>` is first included:

for supported types _FloatN:

- `strfromfN` to `strtofN`

for supported types _FloatNx:

- `strfromfNx` to `strtofNx`

for supported types _DecimalN, where $N \neq 32, 64, \text{ and } 128$:

- `strfromdN` to `strtodN`

for supported types _DecimalNx:

- `strfromdNx` to `strtodNx`

for supported IEC 60559 arithmetic and non-arithmetic binary interchange formats of width $N$:

- `strfromencfN` to `strtoencfN`

for supported IEC 60559 arithmetic and non-arithmetic decimal interchange formats of width $N$:

- `strfromencdec/N` to `strtoencdec/N`
- `strfromencbind/N` to `strtoencbind/N`

6 Types

This clause specifies changes to C11 + TS18661-1 + TS18661-2 to include types that support IEC 60559 arithmetic formats:

- `_FloatN` for binary interchange formats
- `_DecimalN` for decimal interchange formats

for supported types _FloatNx:

- `_FloatNx` for binary extended formats
- `_DecimalNx` for decimal extended formats

The encoding conversion functions (12.4) and numeric conversion functions for encodings (13) support the non-arithmetic interchange formats specified in IEC 60559.
ISO/IEC TS 18661-2 defined *standard floating types* as a collective name for the types `float`, `double`, and `long double` and it defined *decimal floating types* as a collective name for the types `_Decimal32`, `_Decimal64`, and `_Decimal128`. This part of ISO/IEC TS 18661 extends the definition of decimal floating types and defines *binary floating types* to be collective names for types for all the appropriate IEC 60559 arithmetic formats. Thus real floating types are classified as follows:

```plaintext
standard floating types:
  float
  double
  long double

binary floating types:
  _FloatN
  _FloatNx

decimal floating types:
  _DecimalN
  _DecimalNx
```

Note that standard floating types (which have an implementation-defined radix) are not included in either decimal floating types (which all have radix 10) or binary floating types (which all have radix 2).

### Changes to C11 + TS18661-1 + TS18661-2:

Replace 6.2.5#10a-10b:

> [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] Together, the standard floating types and the decimal floating types comprise the real floating types.

with:

> [10a] IEC 60559 specifies interchange formats, identified by their width, which can be used for the exchange of floating-point data between implementations. The two tables below give parameters for the IEC 60559 interchange formats.

#### Binary interchange format parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>binary16</th>
<th>binary32</th>
<th>binary64</th>
<th>binary128</th>
<th>binaryN (N ≥ 128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N, storage width in bits</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>multiple of 32</td>
</tr>
<tr>
<td>p, precision in bits</td>
<td>11</td>
<td>24</td>
<td>53</td>
<td>113</td>
<td>(N - \text{round}(4 \log_2(N)) + 13)</td>
</tr>
<tr>
<td>emax, maximum exponent e</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
<td>(2^{(p-1)} - 1)</td>
</tr>
<tr>
<td>Encoding parameters</td>
<td>bias, $E-e$</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>----</td>
<td>-----</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>sign bit</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$w$, exponent field</td>
<td></td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>width in bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t$, trailing</td>
<td></td>
<td>10</td>
<td>23</td>
<td>52</td>
<td>112</td>
</tr>
<tr>
<td>significand field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>width in bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$N$, storage width</td>
<td></td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>in bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The function round() in the table above rounds to the nearest integer. For example, binary256 would have $p = 237$ and $emax = 262143$.

<table>
<thead>
<tr>
<th>Decimal interchange format parameters</th>
<th>Parameter</th>
<th>decimal32</th>
<th>decimal64</th>
<th>decimal128</th>
<th>decimal$N$ ($N \geq 32$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$, storage width in bits</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>multiple of 32</td>
<td></td>
</tr>
<tr>
<td>$p$, precision in digits</td>
<td>7</td>
<td>16</td>
<td>34</td>
<td>9×$N$/32 - 2</td>
<td></td>
</tr>
<tr>
<td>$emax$, maximum exponent $e$</td>
<td>96</td>
<td>384</td>
<td>6144</td>
<td>3×$2^{16(16 + 1)}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encoding parameters</th>
<th>bias, $E-e$</th>
<th>101</th>
<th>398</th>
<th>6176</th>
<th>emax + $p - 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign bit</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$W + 5$, combination</td>
<td></td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>$N/16 + 9$</td>
</tr>
<tr>
<td>field width in bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t$, trailing</td>
<td></td>
<td>20</td>
<td>50</td>
<td>110</td>
<td>15×$N$/16 - 10</td>
</tr>
<tr>
<td>significand field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>width in bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$N$, storage width</td>
<td></td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>1 + 5 + w + t</td>
</tr>
<tr>
<td>in bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, decimal256 would have $p = 70$ and $emax = 1572864$.

[10b] Types designated

```c
_Float\_N, where \( N \) is 16, 32, 64, or \( \geq 128 \) and a multiple of 32
```

and types designated

```c
_Decimal\_N, where \( N \geq 32 \) and a multiple of 32
```

are collectively called the *interchange floating types*. Each interchange floating type has the IEC 60559 interchange format corresponding to its width $N$ and radix (2 for _Float\_N, 10 for _Decimal\_N). Interchange floating types are not compatible with any other types.

[10c] An implementation that defines __STDC_IEC_60559_BFP__ and __STDC_IEC_60559_TYPES__ shall provide _Float32 and _Float64__ as interchange floating types with the same representation and alignment requirements as float and double, respectively. If the implementation’s _long double_ type supports an IEC 60559 interchange format of width $N > 64$, then the implementation shall also provide the type _Float\_N as an interchange floating type with the same representation and alignment requirements as _long double_. The implementation may provide other binary interchange floating types; the set of such types supported is implementation-defined.
An implementation that defines __STDC_IEC_60559_DFP__ shall provide the types __Decimal32__, __Decimal64__, and __Decimal128__. If the implementation also defines __STDC_IEC_60559_TYPES__, it may provide other decimal interchange floating types; the set of such types supported is implementation-defined.

Note that providing an interchange floating type entails supporting it as an IEC 60559 arithmetic format. An implementation supports IEC 60559 non-arithmetic interchange formats by providing the associated encoding-to-encoding conversion functions (7.12.11.7c), string-to-encoding functions (7.22.1.3c), and string-from-encoding functions (7.22.1.3d). An implementation that defines __STDC_IEC_60559_TYPES__ shall support the IEC 60559 binary16 format, at least as a non-arithmetic interchange format; the set of non-arithmetic interchange formats supported is implementation-defined.

For each of its basic formats, IEC 60559 specifies an extended format whose maximum exponent and precision exceed those of the basic format it is associated with. The table below gives the minimum values of these parameters:

Extended format parameters for floating-point numbers

<table>
<thead>
<tr>
<th>Parameter</th>
<th>binary32</th>
<th>binary64</th>
<th>binary128</th>
<th>decimal64</th>
<th>decimal128</th>
</tr>
</thead>
<tbody>
<tr>
<td>p digits ≥</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>22</td>
<td>40</td>
</tr>
<tr>
<td>emax ≥</td>
<td>1023</td>
<td>16383</td>
<td>65535</td>
<td>6144</td>
<td>24576</td>
</tr>
</tbody>
</table>

Types designated __Float32x__, __Float64x__, __Float128x__, __Decimal64x__, and __Decimal128x__ support the corresponding IEC 60559 extended formats and are collectively called the extended floating types. Extended floating types are not compatible with any other types. An implementation that defines __STDC_IEC_60559_BFP__ and __STDC_IEC_60559_TYPES__ shall provide __Float32x__, which may have the same set of values as double, and may provide any of the other two binary extended floating types. An implementation that defines __STDC_IEC_60559_DFP__ and __STDC_IEC_60559_TYPES__ shall provide: __Decimal64x__, which may have the same set of values as __Decimal128__, and may provide __Decimal128x__. Which (if any) of the optional extended floating types are provided is implementation-defined.

The standard floating types, interchange floating types, and extended floating types are collectively called the real floating types.

The interchange floating types designated __FloatN__ and the extended floating types designated __FloatNx__ are collectively called the binary floating types. The interchange floating types designated __DecimalN__ and the extended floating types designated __DecimalNx__ are collectively called the decimal floating types. Thus the binary floating types and the decimal floating types are real floating types.

The footnote reference above in new paragraph #10d is to the footnote referred to in removed paragraph #10a.

Replace 6.2.5#11:

There are three complex types, designated as float __Complex, double __Complex, and long double __Complex. (Complex types are a conditional feature that implementations need not support; see 6.10.8.3.) The real floating and complex types are collectively called the floating types.
with:

[11] For the standard real types *float*, *double*, and *long double*, the interchange floating types 
_\texttt{FloatN}_, and the extended floating types _\texttt{FloatNx}_, there are complex types designated 
respectively as *float* _\texttt{Complex}_, *double* _\texttt{Complex}_, *long double* _\texttt{Complex}_, _\texttt{FloatN}_ 
_\texttt{Complex}_, and _\texttt{FloatNx}_ _\texttt{Complex}_. (Complex types are a conditional feature that 
implementations need not support; see 6.10.8.3.) The real floating and complex types are collectively 
called the *floating types*.

In the list of keywords in 6.4.1, replace:

\begin{verbatim}
  _Decimal32
  _Decimal64
  _Decimal128
\end{verbatim}

with:

\begin{verbatim}
  _FloatN_, where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32
  _Float32x
  _Float64x
  _Float128x
  _DecimalN_, where \(N\) \(\geq 32\) and a multiple of 32
  _Decimal64x
  _Decimal128x
\end{verbatim}

In the list of type specifiers in 6.7.2, replace:

\begin{verbatim}
  _Decimal32
  _Decimal64
  _Decimal128
\end{verbatim}

with:

\begin{verbatim}
  _FloatN_, where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32
  _Float32x
  _Float64x
  _Float128x
  _DecimalN_, where \(N\) \(\geq 32\) and a multiple of 32
  _Decimal64x
  _Decimal128x
\end{verbatim}

In the list of constraints in 6.7.2#2, replace:

\begin{verbatim}
  — _Decimal32
  — _Decimal64
  — _Decimal128
\end{verbatim}

with:

\begin{verbatim}
  — _FloatN_, where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32
  — _Float32x
  — _Float64x
  — _Float128x
\end{verbatim}
— _Decimal1N, where $N \geq 32$ and a multiple of 32
— _Decimal64x
— _Decimal128x
— _Float/N _Complex, where $N$ is 16, 32, 64, or $\geq 128$ and a multiple of 32
— _Float32x _Complex
— _Float64x _Complex
— _Float128x _Complex

Replace 6.7.2#3a:

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

with:

[3a] The type specifiers _Float/N (where $N$ is 16, 32, 64, or $\geq 128$ and a multiple of 32), _Float32x, _Float64x, _Float128x, _Decimal/N (where $N \geq 32$ and a multiple of 32), _Decimal64x, and _Decimal128x shall not be used if the implementation does not support the corresponding types (see 6.10.8.3).

Replace 6.5#8a:

[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.

with:

[8a] Operators involving operands of interchange or extended floating type are evaluated according to the semantics of IEC 60559, including production of decimal floating-point results with the preferred quantum exponent as specified in IEC 60559 (see 5.2.4.2.2b).

Replace G.2#2:

[2] There are three imaginary types, designated as float _Imaginary, double _Imaginary, and long double _Imaginary. The imaginary types (along with the real floating and complex types) are floating types.

with:

[2] For the standard floating types float, double, and long double, the interchange floating types _Float/N, and the extended floating types _FloatNx, there are imaginary types designated respectively as float _Imaginary, double _Imaginary, long double _Imaginary, _Float/N _Imaginary, and _FloatNx _Imaginary. The imaginary types (along with the real floating and complex types) are floating types.

7 Characteristics

This clause specifies new <float.h> macros, analogous to the macros for standard floating types, that characterize the interchange and extended floating types. Some specification for decimal floating types introduced in ISO/IEC TS 18661-2 is subsumed under the general specification for interchange floating types.
Changes to C11 + TS18661-1 + TS18661-2:

Renumber and rename 5.2.4.2.2a:

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

to:

5.2.4.2.2b Alternate model for decimal floating-point numbers

and remove paragraphs 1-3:

[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. Whenever 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

DEC32_MANT_DIG 7
DEC64_MANT_DIG 16
DEC128_MANT_DIG 34

— minimum exponent

DEC32_MIN_EXP -94
DEC64_MIN_EXP -382
DEC128_MIN_EXP -6142

— maximum exponent

DEC32_MAX_EXP 97
DEC64_MAX_EXP 385
DEC128_MAX_EXP 6145
— maximum representable finite decimal floating-point number (there are 6, 15 and 33 9's after the decimal points respectively)

\[
\begin{align*}
\text{DEC32\_MAX} & = 9.999999E+96DF \\
\text{DEC64\_MAX} & = 9.99999999999999E+384DD \\
\text{DEC128\_MAX} & = 9.999999999999999999999999999999999E+6144DL
\end{align*}
\]

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

\[
\begin{align*}
\text{DEC32\_EPSILON} & = 1E^{-6DF} \\
\text{DEC64\_EPSILON} & = 1E^{-15DD} \\
\text{DEC128\_EPSILON} & = 1E^{-33DL}
\end{align*}
\]

— minimum normalized positive decimal floating-point number

\[
\begin{align*}
\text{DEC32\_MIN} & = 1E^{-95DF} \\
\text{DEC64\_MIN} & = 1E^{-383DD} \\
\text{DEC128\_MIN} & = 1E^{-6143DL}
\end{align*}
\]

— minimum positive subnormal decimal floating-point number

\[
\begin{align*}
\text{DEC32\_TRUE\_MIN} & = 0.000001E^{-95DF} \\
\text{DEC64\_TRUE\_MIN} & = 0.000000000000001E^{-383DD} \\
\text{DEC128\_TRUE\_MIN} & = 0.000000000000000000001E^{-6143DL}
\end{align*}
\]

After 5.2.4.2.2, insert:

5.2.4.2.2a Characteristics of interchange and extended floating types in `<float.h>`

[1] This subclause specifies macros in `<float.h>` that provide characteristics of interchange floating types and extended floating types in terms of the model presented in 5.2.4.2.2. The prefix `FLT\_N` indicates a binary interchange floating type of width \(N\). The prefix `FLT\_NX` indicates a binary extended floating type that extends a basic format of width \(N\). The prefix `DEC\_N` indicates a decimal interchange floating type of width \(N\). The prefix `DEC\_NX` indicates a decimal extended floating type that extends a basic format of width \(N\). The type parameters \(p, e_{\text{max}},\) and \(e_{\text{min}}\) for extended floating types are for the extended floating type itself, not for the basic format that it extends. For each interchange or extended floating type that the implementation provides, `<float.h>` shall define the associated macros in the following lists. Conversely, for each such type that the implementation does not provide, `<float.h>` shall not define the associated macros in the following lists.

[2] If `FLT\_RADIX` is 2, the value of the macro `FLT\_EVAL\_METHOD` (5.2.4.2.2) characterizes the use of evaluation formats for standard floating types and for binary interchange and extended floating types:

\[
\begin{align*}
-1 & \text{ indeterminable;} \\
0 & \text{ evaluate all operations and constants, whose semantic type has at most the range and precision of } \textit{float}, \text{ to the range and precision of } \textit{float}; \text{ evaluate all other operations and constants to the range and precision of the semantic type;}
\end{align*}
\]

\[
\begin{align*}
1 & \text{ evaluate operations and constants, whose semantic type has at most the range and precision of } \textit{double}, \text{ to the range and precision of } \textit{double}; \text{ evaluate all other operations and constants to the range and precision of the semantic type;}
\end{align*}
\]

\[
\begin{align*}
2 & \text{ evaluate operations and constants, whose semantic type has at most the range and precision of } \textit{long\ double}, \text{ to the range and precision of } \textit{long\ double}; \text{ evaluate all other operations and constants to the range and precision of the semantic type;}
\end{align*}
\]
N, where _FloatN is a supported interchange floating type  
evaluate operations and constants, whose semantic type has at most the range and  
precision of the _FloatN type, to the range and precision of the _FloatN type; evaluate  
all other operations and constants to the range and precision of the semantic type;

5  
N + 1, where _FloatNx is a supported extended floating type  
evaluate operations and constants, whose semantic type has at most the range and  
precision of the _FloatNx type, to the range and precision of the _FloatNx type;  
evaluate all other operations and constants to the range and precision of the semantic type.

If FLT_RADIX is not 2, the use of evaluation formats for operations and constants of binary  
interchange and extended floating types is implementation-defined.

[3] The implementation-defined value of the macro DEC_EVAL_METHOD characterizes the use of  
evaluation formats (see analogous FLT_EVAL_METHOD in 5.2.4.2.2) for decimal interchange and extended floating types:

-1 indeterminable;

0 evaluate all operations and constants just to the range and precision of the type;

1 evaluate operations and constants, whose semantic type has at most the range and  
precision of the _Decimal64 type, to the range and precision of the _Decimal64 type;  
evaluate all other operations and constants to the range and precision of the semantic type;

2 evaluate operations and constants, whose semantic type has at most the range and  
precision of the _Decimal128 type, to the range and precision of the _Decimal128 type;  
evaluate all other operations and constants to the range and precision of the semantic type;

N, where _DecimalN is a supported interchange floating type  
evaluate operations and constants, whose semantic type has at most the range and  
precision of the _DecimalN type, to the range and precision of the _DecimalN type;  
evaluate all other operations and constants to the range and precision of the semantic type;

N + 1, where _DecimalNx is a supported extended floating type  
evaluate operations and constants, whose semantic type has at most the range and  
precision of the _DecimalNx type, to the range and precision of the _DecimalNx type;  
evaluate all other operations and constants to the range and precision of the semantic type;

[4] 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 (= 2 for binary, 10 for decimal)

For the standard floating types, this value is implementation-defined and is specified by the macro  
FLT_RADIX. For the interchange and extended floating types there is no corresponding macro,  
since the radix is an inherent property of the types.

— number of bits in the floating-point significand, p

FLT_N_MANT_DIG
FLT_NX_MANT_DIG

— number of digits in the coefficient, p

DEC_N_MANT_DIG
DEC_NX_MANT_DIG
number of decimal digits, \( n \), such that any floating-point number with \( p \) bits can be rounded to a floating-point number with \( n \) decimal digits and back again without change to the value, \( \left\lfloor 1 + p \log_{10} 2 \right\rfloor \)

\[ \text{FLT}_N \text{DECIMAL_DIG} \]

\[ \text{FLT}_N \times \text{DECIMAL_DIG} \]

number of decimal digits, \( q \), such that any floating-point number with \( q \) decimal digits can be rounded into a floating-point number with \( p \) bits and back again without change to the \( q \) decimal digits, \( \left\lfloor (p - 1) \log_{10} 2 \right\rfloor \)

\[ \text{FLT}_N \text{DIG} \]

\[ \text{FLT}_N \times \text{DIG} \]

minimum negative integer such that the radix raised to one less than that power is a normalized floating-point number, \( e_{\text{min}} \)

\[ \text{FLT}_N \text{MIN.EXP} \]

\[ \text{FLT}_N \times \text{MIN.EXP} \]

\[ \text{DEC}_N \text{MIN.EXP} \]

\[ \text{DEC}_N \times \text{MIN.EXP} \]

minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, \( \left\lfloor \log_{10} 2^{e_{\text{min}}-1} \right\rfloor \)

\[ \text{FLT}_N \text{MIN.10.EXP} \]

\[ \text{FLT}_N \times \text{MIN.10.EXP} \]

maximum integer such that the radix raised to one less than that power is a representable finite floating-point number, \( e_{\text{max}} \)

\[ \text{FLT}_N \text{MAX.EXP} \]

\[ \text{FLT}_N \times \text{MAX.EXP} \]

\[ \text{DEC}_N \text{MAX.EXP} \]

\[ \text{DEC}_N \times \text{MAX.EXP} \]

maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, \( \left\lfloor \log_{10}(1 - 2^{-p})2^{e_{\text{max}}} \right\rfloor \)

\[ \text{FLT}_N \text{MAX.10.EXP} \]

\[ \text{FLT}_N \times \text{MAX.10.EXP} \]

maximum representable finite floating-point number, \( (1 - b^{-p})b^{e_{\text{max}}} \)

\[ \text{FLT}_N \text{MAX} \]

\[ \text{FLT}_N \times \text{MAX} \]

\[ \text{DEC}_N \text{MAX} \]

\[ \text{DEC}_N \times \text{MAX} \]

the difference between 1 and the least value greater than 1 that is representable in the given floating-point type, \( b^{1-p} \)

\[ \text{FLT}_N \text{EPSILON} \]

\[ \text{FLT}_N \times \text{EPSILON} \]

\[ \text{DEC}_N \text{EPSILON} \]

\[ \text{DEC}_N \times \text{EPSILON} \]
minimum normalized positive floating-point number, $b^{emin-1}$

$$\begin{align*}
\text{FLT}_N\_MIN \\
\text{FLT}_N\_X\_MIN \\
\text{DEC}_N\_MIN \\
\text{DEC}_N\_X\_MIN
\end{align*}$$

minimum positive subnormal floating-point number, $b^{emin-p}$

$$\begin{align*}
\text{FLT}_N\_TRUE\_MIN \\
\text{FLT}_N\_X\_TRUE\_MIN \\
\text{DEC}_N\_TRUE\_MIN \\
\text{DEC}_N\_X\_TRUE\_MIN
\end{align*}$$

With the following change, DECIMAL_DIG characterizes conversions of supported IEC 60559 encodings, which may be wider than supported floating types.

Change to C11 + TS18661-1 + TS18661-2:

In 5.2.4.2.2#11, change the bullet defining DECIMAL_DIG from:

— number of decimal digits, $n$, such that any floating-point number in the widest supported floating type with ...

to:

— number of decimal digits, $n$, such that any floating-point number in the widest of the supported floating types and the supported IEC 60559 encodings with ...

8 Conversions

The following change to C11 + TS18661-1 + TS18661-2 enhances the usual arithmetic conversions to handle interchange and extended floating types. IEC 60559 recommends against allowing implicit conversions of operands to obtain a common type where the conversion is between types where neither is a subset of (or equivalent to) the other. The following change supports this restriction.

Changes to C11 + TS18661-1 + TS18661-2:

Replace 6.3.1.4#1a:

[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.

with:

[1a] When a finite value of interchange or extended 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.

Replace 6.3.1.4#2a:

[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.
with:

[2a] When a value of integer type is converted to an interchange or extended 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.

In 6.3.1.8#1, replace the following items 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 _Decimal64, the other operand is converted to _Decimal64.

Otherwise, if the type of either operand is _Decimal32, the other operand is converted to _Decimal32.

If there are no decimal floating types in the operands:

First, if the corresponding real type of either operand is long double, the other operand is converted, without change of type domain, to a type whose corresponding real type is long double.

Otherwise, if the corresponding real type of either operand is double, the other operand is converted, without change of type domain, to a type whose corresponding real type is double.

Otherwise, if the corresponding real type of either operand is float, the other operand is converted, without change of type domain, to a type whose corresponding real type is float.

with:

If one operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

If both operands have floating types and neither of the sets of values of their corresponding real types is a subset of (or equivalent to) the other, the behavior is undefined.

Otherwise, if both operands are floating types and the sets of values of their corresponding real types are equivalent, then the following rules are applied:

If both operands have the same corresponding real type, no further conversion is needed.

Otherwise, if the corresponding real type of either operand is an interchange floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same interchange floating type.

Otherwise, if the corresponding real type of either operand is a standard floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same standard floating type.

Otherwise, if both operands have floating types, the operand, whose set of values of its corresponding real type is a (proper) subset of the set of values of the corresponding real type of the other operand, is converted, without change of type domain, to a type with the corresponding real type of that other operand.
Otherwise, if one operand has a floating type, the other operand is converted to the corresponding real type of the operand of floating type.

9 Constants

The following changes to C11 + TS18661-1 + TS18661-2 provide suffixes that designate constants of interchange and extended floating types.

Changes to C11 + TS18661-1 + TS18661-2:

Change floating-suffix in 6.4.4.2 from:

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

...to:

floating-suffix: one of  
   f  l  F  L  df  dd  dl  DF  DD  DL  fN  FN  fNx  FNx  dN  DN  dNx  DNx

Replace 6.4.4.2#2a:

[2a] A floating-suffix df, dd, dl, DF, DD, or DL shall not be used in a hexadecimal-floating-constant.

with:

[2a] A floating-suffix df, dd, dl, DF, DD, DN, dNx, or DNx shall not be used in a hexadecimal-floating-constant.

[2b] A floating-suffix shall not designate a type that the implementation does not provide.

Replace 6.4.4.2#4a:

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

with:

[4a] If a floating constant is suffixed by fN or FN, it has type _FloatN. If suffixed by fNx or FNx, it has type _FloatNx. If suffixed by df or DF, it has type _Decimal32. If suffixed by dd or DD, it has type _Decimal64. If suffixed by dl or DL, it has type _Decimal128. If suffixed by dN or DN, it has type _DecimalN. If suffixed by dNx or DNx, it has type _DecimalNx.

Replace the second sentence of 6.4.4.2#5a:

The quantum exponent is specified to be the same as for the corresponding strtod32, strtod64, or strtod128 function for the same numeric string.

with:

The quantum exponent is specified to be the same as for the corresponding strtodN or strtodNx function for the same numeric string.

10 Expressions

The following changes to C11 + TS18661-1 + TS18661-2 specify operator constraints for interchange and extended floating types.
Changes to C11 + TS18661-1 + TS18661-2:

Replace 6.5.5#2a:

\[[2a]\] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

with:

\[[2a]\] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

Replace 6.5.6#3a:

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

with:

\[[3a]\] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

Replace 6.5.8#2a:

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

with:

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

Replace 6.5.9#2a:

\[[2a]\] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

with:

\[[2a]\] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

Replace 6.5.15#3a:

\[[3a]\] 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.

with:

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

Replace 6.5.16#2a:

\[[2a]\] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

with:

\[[2a]\] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.
with:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

In F.9.2#1, replace the first sentence:

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

with:

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

11 Non-arithmetic interchange formats

An implementation supports IEC 60559 arithmetic interchange formats by providing the corresponding interchange floating types. An implementation supports IEC 60559 non-arithmetic formats by providing the encoding-to-encoding conversion functions in `<math.h>` and the string-to-encoding and string-from-encoding functions in `<string.h>` and `<math.h>` (see 6.2.5). These functions, together with functions required for interchange floating types, provide conversions between any two of the supported IEC 60559 arithmetic and non-arithmetic interchange formats and between character sequences and any supported IEC 60559 arithmetic or non-arithmetic format.

12 Mathematics `<math.h>`

This clause specifies changes to C11 + TS18661-1 + TS18661-2 to include functions and macros for interchange and extended floating types. The binary types are supported by functions and macros corresponding to those specified for standard floating types (`float`, `double`, and `long double`) in C11 + TS18661-1, including Annex F. The decimal types are supported by functions and macros corresponding to those specified for decimal floating types in TS18661-2.

All classification (7.12.3) and comparison (7.12.14) macros specified in C11 + TS18661-1 + TS18661-2 naturally extend to handle interchange and extended floating types.

This clause also specifies encoding conversion functions that are part of support for the non-arithmetic interchange formats in IEC 60559 (see 6.2.5).

Changes to C11 + TS18661-1 + TS18661-2:

In 7.12#1, change the second sentence from:

Most synopses specify a family of functions consisting of a principal function with one or more `double` parameters, a `double` return value, or both; and other functions with the same name but with `f` and `l` suffixes, which are corresponding functions with `float` and `long double` parameters, return values, or both.226)

to:

Most synopses specify a family of functions consisting of:

a principal function with one or more `double` parameters, a `double` return value, or both; and, other functions with the same name but with `f`, `l`, `fN`, `fNx`, `d`, and `dN` suffixes, which are corresponding functions whose parameters, return values, or both are of types `float`, `long double`, `_FloatN`, `_FloatNx`, `_DecimalN`, and `_DecimalNx`, respectively.226)
Add after 7.12#1d:

[1e] For each interchange or extended floating type that the implementation provides, `<math.h>` shall define the associated macros and declare the associated functions. Conversely, for each such type that the implementation does not provide, `<math.h>` shall not define the associated macros or declare the associated functions unless explicitly specified otherwise.

Change 7.12#2, from:

[2] The types

```c
float_t
double_t
```

are floating types at least as wide as `float` and `double`, respectively, and such that `double_t` is at least as wide as `float_t`. If `FLT_EVAL_METHOD` equals 0, `float_t` and `double_t` are `float` and `double`, respectively; if `FLT_EVAL_METHOD` equals 1, they are both `double`; if `FLT_EVAL_METHOD` equals 2, they are both `long double`; and for other values of `FLT_EVAL_METHOD`, they are otherwise implementation-defined.227)

to:

[2] The types

```c
float_t
double_t
long_double_t
```

and for each supported type `_FloatN`, the type

```c
_FloatN_t
```

and for each supported type `_DecimalN`, the type

```c
_DecimalN_t
```

are floating types, such that:

— each of the types has at least the range and precision of the corresponding real floating type `float`, `double`, `long double`, `_FloatN`, and `_DecimalN`, respectively;

— `double_t` has at least the range and precision of `float_t`;

— `long_double_t` has at least the range and precision of `double_t`;

— `_FloatN_t` has at least the range and precision of `_FloatM_t` if \( N > M \);

— `_DecimalN_t` has at least the range and precision of `_DecimalM_t` if \( N > M \).

If `FLT_RADIX` is 2 and `FLT_EVAL_METHOD` is nonnegative, then each of the types corresponding to a standard or binary floating type is the type whose range and precision are specified by 5.2.4.2.2a to be used for evaluating operations and constants of that standard or binary floating type. If `DEC_EVAL_METHOD` is nonnegative, then each of the types corresponding to a decimal floating type is the type whose range and precision are specified by 5.2.4.2.2a to be used for evaluating operations and constants of that decimal floating type.
Delete footnote 227:

227) The types float_t and double_t are intended to be the implementation’s most efficient
types at least as wide as float and double, respectively. For FLT_EVAL_METHOD equal to 0,
1, or 2, the type float_t is the narrowest type used by the implementation to evaluate floating
expressions.

12.1 Macros

Changes to C11 + TS18661-1 + TS18661-2:

Replace 7.12#3a:

[3a] The macro

    HUGE_VAL_D32

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

    HUGE_VAL_D64
    HUGE_VAL_D128

are respectively _Decimal64 and _Decimal128 analogues of HUGE_VAL_D32.

with:

[3a] The macros

    HUGE_VAL_FN
    HUGE_VAL_DN
    HUGE_VAL_FNX
    HUGE_VAL_DNX

expand to constant expressions of types _FloatN, _DecimalN, _FloatNx, and _DecimalNx,
respectively, representing positive infinity.

Replace 7.12#5b:

[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.

with:

[5b] The signaling NaN macros

    SNANFN
    SNANDN
    SNANFNX
    SNANDNX
expand to constant expressions of types \_FloatN, \_DecimalN, \_FloatNx, and \_DecimalNx, respectively, 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.

Replace 7.12#7b:

[7b] The macros

\texttt{FP\_FAST\_FMAD32}
\texttt{FP\_FAST\_FMAD64}
\texttt{FP\_FAST\_FMAD128}

are, respectively, \_Decimal32, \_Decimal64, and \_Decimal128 analogues of \texttt{FP\_FAST\_FMA}.

with:

[7b] The macros

\texttt{FP\_FAST\_FMAF/N}
\texttt{FP\_FAST\_FMAD/N}
\texttt{FP\_FAST\_FMAF/Nx}
\texttt{FP\_FAST\_FMAD/Nx}

are, respectively, \_FloatN, \_DecimalN, \_FloatNx, and \_DecimalNx analogues of \texttt{FP\_FAST\_FMA}.

Replace 7.12#7c:

[7c] The macros

\texttt{FP\_FAST\_D32ADDD64}
\texttt{FP\_FAST\_D32ADDD128}
\texttt{FP\_FAST\_D64ADDD128}
\texttt{FP\_FAST\_D32SUBD64}
\texttt{FP\_FAST\_D32SUBD128}
\texttt{FP\_FAST\_D64SUBD128}
\texttt{FP\_FAST\_D32MULD64}
\texttt{FP\_FAST\_D32MULD128}
\texttt{FP\_FAST\_D64MULD128}
\texttt{FP\_FAST\_D32DIVD64}
\texttt{FP\_FAST\_D32DIVD128}
\texttt{FP\_FAST\_D64DIVD128}
\texttt{FP\_FAST\_D32FMAD64}
\texttt{FP\_FAST\_D32FMAD128}
\texttt{FP\_FAST\_D64FMAD128}
\texttt{FP\_FAST\_D32SQRTD64}
\texttt{FP\_FAST\_D32SQRTD128}
\texttt{FP\_FAST\_D64SQRTD128}

are decimal analogues of \texttt{FP\_FAST\_FADD}, \texttt{FP\_FAST\_FADDL}, \texttt{FP\_FAST\_DADDL}, etc.

with:

[7c] The macros in the following lists are interchange and extended floating type analogues of

\texttt{FP\_FAST\_FADD}, \texttt{FP\_FAST\_FADDL}, \texttt{FP\_FAST\_DADDL}, etc.
[7d] For $M < N$, the macros

\[
\begin{align*}
\text{FP\_FAST\_FMADDN} \\
\text{FP\_FAST\_FMSUBN} \\
\text{FP\_FAST\_FMULFN} \\
\text{FP\_FAST\_FMDIVFN} \\
\text{FP\_FAST\_FMAPN} \\
\text{FP\_FAST\_FMQRTFN} \\
\text{FP\_FAST\_DMADDN} \\
\text{FP\_FAST\_DSUBN} \\
\end{align*}
\]

5

\[
\begin{align*}
\text{FP\_FAST\_DMULDN} \\
\text{FP\_FAST\_DMRDYN} \\
\text{FP\_FAST\_DMFMADN} \\
\text{FP\_FAST\_DMSQRTDN} \\
\end{align*}
\]

characterize the corresponding functions whose arguments are of an interchange floating type of width $N$ and whose return type is an interchange floating type of width $M$.

[7e] For $M \leq N$, the macros

\[
\begin{align*}
\text{FP\_FAST\_FMADDNX} \\
\text{FP\_FAST\_FMSUBNX} \\
\text{FP\_FAST\_FMULFNX} \\
\text{FP\_FAST\_FMDIVNX} \\
\text{FP\_FAST\_FMQRTNX} \\
\text{FP\_FAST\_DMADDNX} \\
\text{FP\_FAST\_DSUBNX} \\
\end{align*}
\]

20

\[
\begin{align*}
\text{FP\_FAST\_DMULDNX} \\
\text{FP\_FAST\_DMRDYNX} \\
\text{FP\_FAST\_DMFMADNX} \\
\text{FP\_FAST\_DMSQRTDNX} \\
\end{align*}
\]

characterize the corresponding functions whose arguments are of an extended floating type that extends a format of width $N$ and whose return type is an interchange floating type of width $M$.

[7f] For $M < N$, the macros

\[
\begin{align*}
\text{FP\_FAST\_FMXADDFN} \\
\text{FP\_FAST\_FMXSUBFN} \\
\text{FP\_FAST\_FMXULFN} \\
\text{FP\_FAST\_FMXDIVFN} \\
\text{FP\_FAST\_FMXQRTFN} \\
\text{FP\_FAST\_DMXADDN} \\
\text{FP\_FAST\_DSUBN} \\
\end{align*}
\]

35

\[
\begin{align*}
\text{FP\_FAST\_DMXULDN} \\
\text{FP\_FAST\_DMXRDN} \\
\text{FP\_FAST\_DMXFADN} \\
\text{FP\_FAST\_DMSQRTDN} \\
\end{align*}
\]

characterize the corresponding functions whose arguments are of an interchange floating type of width $N$ and whose return type is an extended floating type that extends a format of width $M$. 
[7g] For \( M < N \), the macros

\[
\begin{align*}
&\text{FP\_FAST\_FM\_XADD\_F\_NX} \\
&\text{FP\_FAST\_FM\_XSUB\_F\_NX} \\
&\text{FP\_FAST\_FM\_XMUL\_F\_NX} \\
&\text{FP\_FAST\_FM\_XDIV\_F\_NX} \\
&\text{FP\_FAST\_FM\_XFMA\_F\_NX} \\
&\text{FP\_FAST\_FM\_XSQR\_F\_NX} \\
&\text{FP\_FAST\_DM\_XADD\_D\_NX} \\
&\text{FP\_FAST\_DM\_XSUB\_D\_NX} \\
&\text{FP\_FAST\_DM\_XMUL\_D\_NX} \\
&\text{FP\_FAST\_DM\_XDIV\_D\_NX} \\
&\text{FP\_FAST\_DM\_XFMA\_D\_NX} \\
&\text{FP\_FAST\_DM\_XSQR\_D\_NX}
\end{align*}
\]

characterize the corresponding functions whose arguments are of an extended floating type that extends a format of width \( N \) and whose return type is an extended floating type that extends a format of width \( M \).

### 12.2 Floating-point environment

#### Changes to C11 + TS18661-1 + TS18661-2:

In 7.6.1a#2, change the first sentence from:

> 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.

To:

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

In 7.6.1a#3, change the first sentence from:

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

To:

> `direction` shall be one of the names of the supported rounding direction macros for use with `fegetround` and `fesetround` (7.6), or `FE_DYNAMIC`.

In 7.6.1a#4, change the first sentence from:

> The `FENV_ROUND` directive affects operations for standard floating types.

To:

> The `FENV_ROUND` directive affects operations for standard and binary floating types.

In 7.6.1a#4, change the table title from:

> Functions affected by constant rounding modes – for standard floating types
Functions affected by constant rounding modes – for standard and binary floating types

In 7.6.1a#4, replace the sentence following the table:

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`).

with:

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

After 7.6.1a#4, add:

[4a] The `fMencfN`, `strfromencfN`, and `strtoencfN` functions for binary interchange types are also affected by constant rounding modes.

In 7.6.1b#2 after the table, add:

Each `<math.h>` function listed in the table above indicates the family of functions of all decimal floating types (for example, `acosdNx`, as well as `acosdN`).

After 7.6.1b#2, add:

[3] The `dMencbindN`, `dMencdecdN`, `strfromencbindN`, `strfromencdecdN`, `strtoencbindN`, and `strtoencdecdN` functions for decimal interchange types are also affected by constant rounding modes.

Change 7.6.3 from:

The `fegetround` and `fesetround` functions provide control of rounding direction modes.

to:

The functions in this subclause provide control of rounding direction modes.

Change 7.6.3.1#2 from:

The `fegetround` function gets the current value of the dynamic rounding direction mode.

to:

The `fegetround` function gets the current value of the dynamic rounding direction mode for operations for standard and binary floating types.

In 7.6.3.2#2, change the first sentence from:

The `fesetround` function sets the dynamic rounding direction mode to the rounding direction represented by its argument `round`.

to:

The `fesetround` function sets the dynamic rounding direction mode to the rounding direction represented by its argument `round` for operations for standard and binary floating types.
12.3 Functions

Changes to C11 + TS18661-1 + TS18661-2:

Add the following list of function prototypes to the synopsis of the respective subclauses. In each synopsis where a prototype with a dN suffix is added, remove any prototypes with a d32, d64, or d128 suffix.

7.12.4 Trigonometric functions

```c
_FloatN acosN(_FloatN x);
_FloatNx acosfNx(_FloatNx x);
_DECIMALN acosdN(_DECIMALN x);
_DECIMALNx acosdNx(_DECIMALNx x);

_FloatN asinN(_FloatN x);
_FloatNx asinfNx(_FloatNx x);
_DECIMALN asindN(_DECIMALN x);
_DECIMALNx asindNx(_DECIMALNx x);

_FloatN atanN(_FloatN x);
_FloatNx atanfNx(_FloatNx x);
_DECIMALN atandN(_DECIMALN x);
_DECIMALNx atandNx(_DECIMALNx x);

_FLOATN atan2fN(_FloatN y, _FloatN x);
_FLOATNx atan2fNx(_FloatNx y, _FloatNx x);
_DECIMALN atan2dN(_DECIMALN y, _DECIMALN x);
_DECIMALNx atan2dNx(_DECIMALNx y, _DECIMALNx x);

_FLOATN cosfN(_FloatN x);
_FLOATNx cosfNx(_FloatNx x);
_DECIMALN cosdN(_DECIMALN x);
_DECIMALNx cosdNx(_DECIMALNx x);

_FLOATN sinfN(_FloatN x);
_FLOATNx sinfNx(_FloatNx x);
_DECIMALN sindN(_DECIMALN x);
_DECIMALNx sindNx(_DECIMALNx x);

_FLOATN tanfN(_FloatN x);
_FLOATNx tanfNx(_FloatNx x);
_DECIMALN tandN(_DECIMALN x);
_DECIMALNx tandNx(_DECIMALNx x);
```

7.12.5 Hyperbolic functions

```c
_FLOATN acoshfN(_FloatN x);
_FLOATNx acoshfNx(_FloatNx x);
_DECIMALN acoshdN(_DECIMALN x);
_DECIMALNx acoshdNx(_DECIMALNx x);

_FLOATN asinhfN(_FloatN x);
_FLOATNx asinhfNx(_FloatNx x);
_DECIMALN asinhdN(_DECIMALN x);
_DECIMALNx asinhdNx(_DECIMALNx x);
```
_FloatN atanhfN(_FloatN x);
_FloatNx atanhfNx(_FloatNx x);
DecimalN atanhdN(_DecimalN x);
_DecimalNx atanhdNx(_DecimalNx x);

_FloatN coshfN(_FloatN x);
_FloatNx coshfNx(_FloatNx x);
DecimalN coshdN(_DecimalN x);
_DecimalNx scoshdNx(_DecimalNx x);

_FloatN sinhfN(_FloatN x);
_FloatNx sinhfNx(_FloatNx x);
DecimalN sinhN(_DecimalN x);
_DecimalNx sinhNx(_DecimalNx x);

_FloatN tanhfN(_FloatN x);
_FloatNx tanhfNx(_FloatNx x);
DecimalN tanhdN(_DecimalN x);
_DecimalNx tanhdNx(_DecimalNx x);

7.12.6 Exponential and logarithmic functions

_FloatN expfN(_FloatN x);
_FloatNx expfNx(_FloatNx x);
 DecimalN expdN(_DecimalN x);
_DecimalNx expdNx(_DecimalNx x);

_FloatN exp2fN(_FloatN x);
_FloatNx exp2fNx(_FloatNx x);
DecimalN exp2dN(_DecimalN x);
_DecimalNx exp2dNx(_DecimalNx x);

_FloatN expmlfN(_FloatN x);
_FloatNx expmlfNx(_FloatNx x);
DecimalN expmldN(_DecimalN x);
_DecimalNx expmldNx(_DecimalNx x);

_FloatN frexfN(_FloatN value, int *exp);
_FloatNx frexfNx(_FloatNx value, int *exp);
DecimalN frexpdN(_DecimalN value, int *exp);
_DecimalNx frexpdNx(_DecimalNx value, int *exp);

int ilogbfN(_FloatN x);
int ilogbfNx(_FloatNx x);
int ilogbdN(_DecimalN x);
int ilogbdNx(_DecimalNx x);

_FloatN ldexpfN(_FloatN value, int exp);
_FloatNx ldexpfNx(_FloatNx value, int exp);
DecimalN ldexpdN(_DecimalN value, int exp);
_DecimalNx ldexpdNx(_DecimalNx value, int exp);
7.12.7 Power and absolute-value functions

```c
long int llogbfN(_FloatN x);
long int llogbfNx(_FloatNx x);
long int llogbdN(_DecimalN x);
long int llogbdNx(_DecimalNx x);

_FloatN logfN(_FloatN x);
_FloatNx logfNx(_FloatNx x);
_DeimalN logdN(_DecimalN x);
_DeimalNx logdNx(_DecimalNx x);

_FloatN log10fN(_FloatN x);
_FloatNx log10fNx(_FloatNx x);
_DeimalN log10dN(_DecimalN x);
_DeimalNx log10dNx(_DecimalNx x);

_FloatN log1pfN(_FloatN x);
_FloatNx log1pfNx(_FloatNx x);
_DeimalN log1pdN(_DecimalN x);
_DeimalNx log1pdNx(_DecimalNx x);

_FloatN log2fN(_FloatN x);
_FloatNx log2fNx(_FloatNx x);
_DeimalN log2dN(_DecimalN x);
_DeimalNx log2dNx(_DecimalNx x);

_FloatN logbfN(_FloatN x);
_FloatNx logbfNx(_FloatNx x);
_DeimalN logbdN(_DecimalN x);
_DeimalNx logbdNx(_DecimalNx x);

_FloatN modffN(_FloatN x, _FloatN *iptr);
_FloatNx modffNx(_FloatNx x, _FloatNx *iptr);
_DeimalN modfdN(_DecimalN x, _DecimalN *iptr);
_DeimalNx modfdNx(_DecimalNx x, _DecimalNx *iptr);

_FloatN scalbnfN(_FloatN value, int exp);
_FloatNx scalbnfNx(_FloatNx value, int exp);
_DeimalN scalbdN(_DecimalN value, int exp);
_DeimalNx scalbdNx(_DecimalNx value, int exp);

_FloatN scalbnfN(_FloatN value, long int exp);
_FloatNx scalbnfNx(_FloatNx value, long int exp);
_DeimalN scalbdN(_DecimalN value, long int exp);
_DeimalNx scalbdNx(_DecimalNx value, long int exp);

_FloatN cbrtfN(_FloatN x);
_FloatNx cbrtfNx(_FloatNx x);
_DeimalN cbrtdN(_DecimalN x);
_DeimalNx cbrtdNx(_DecimalNx x);

_FloatN fabsfN(_FloatN x);
_FloatNx fabsfNx(_FloatNx x);
_DeimalN fabsdN(_DecimalN x);
_DeimalNx fabsdNx(_DecimalNx x);
```
7.12.8 Error and gamma functions

_FloatN erffN(_FloatN x);
_FloatN x erffNx(_FloatN x);
_DecimalN erfdN(_DecimalN x);
_DecimalN x erfdNx(_DecimalN x);

_FloatN erfcfN(_FloatN x);
_FloatN x erfcfNx(_FloatN x);
_DecimalN erfcdN(_DecimalN x);
_DecimalN x erfcdNx(_DecimalN x);

_FloatN lgammafN(_FloatN x);
_FloatN x lgammafNx(_FloatN x);
_DecimalN lgammadN(_DecimalN x);
_DecimalN x lgammadNx(_DecimalN x);

_FloatN tgammafN(_FloatN x);
_FloatN x tgammafNx(_FloatN x);
_DecimalN tgammadN(_DecimalN x);
_DecimalN x tgammadNx(_DecimalN x);

7.12.9 Nearest integer functions

_FloatN ceilfN(_FloatN x);
_FloatN x ceilfNx(_FloatN x);
_DecimalN ceildN(_DecimalN x);
_DecimalN x ceildNx(_DecimalN x);

_FloatN floorfN(_FloatN x);
_FloatN x floorfNx(_FloatN x);
_DecimalN floordN(_DecimalN x);
_DecimalN x floordNx(_DecimalN x);

_FloatN nearbyintfN(_FloatN x);
_FloatN x nearbyintfNx(_FloatN x);
_DecimalN nearbyintdN(_DecimalN x);
_DecimalN x nearbyintdNx(_DecimalN x);
_FloatN rintfN(_FloatN x);
_FloatNx rintfNx(_Float Nx x);
_DecimalN rintdN(_DecimalN x);
_DecimalNx rintdNx(_Decimal Nx x);

long int lrintfN(_FloatN x);
long int lrintfNx(_Float Nx x);
long int lrintdN(_DecimalN x);
long int lrintdNx(_Decimal Nx x);

long long int lllrintfN(_FloatN x);
long long int lllrintfNx(_Float Nx x);
long long int lllrintdN(_DecimalN x);
long long int lllrintdNx(_Decimal Nx x);

_FloatN roundfN(_FloatN x);
_FloatNx roundfNx(_FloatNx x);
_DecimalN rounddN(_DecimalN x);
_DecimalNx rounddNx(_Decimal Nx x);

long int lroundfN(_FloatN x);
long int lroundfNx(_Float Nx x);
long int lrounddN(_DecimalN x);
long int lrounddNx(_Decimal Nx x);

long long int lllroundfN(_FloatN x);
long long int lllroundfNx(_Float Nx x);
long long int lllrounddN(_DecimalN x);
long long int lllrounddNx(_Decimal Nx x);

_FloatN roundevenfN(_FloatN x);
_FloatNx roundevenfNx(_FloatNx x);
_DecimalN roundevendN(_DecimalN x);
_DecimalNx roundevendNx(_Decimal Nx x);

_FloatN truncfN(_FloatN x);
_FloatNx truncfNx(_FloatNx x);
_DecimalN truncdN(_DecimalN x);
_DecimalNx truncdNx(_Decimal Nx x);

intmax_t fromfpfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpfNx(_Float Nx x, int round, unsigned int width);
intmax_t fromfpdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpdNx(_Decimal Nx x, int round, unsigned int width);
uintmax_t ufromfpfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpfNx(_Float Nx x, int round, unsigned int width);
uintmax_t ufromfpdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpdNx(_Decimal Nx x, int round, unsigned int width);
uintmax_t ufromfpxfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpfNx(_FloatNx x, int round, unsigned int width);
uintmax_t ufromfpdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpdNx(_Decimal Nx x, int round, unsigned int width);
uintmax_t ufromfpfxdN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpfxfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpfxdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpfxfN(_DecimalN x, int round, unsigned int width);

7.12.10 Remainder functions

_FloatN fmodN(_FloatN x, _FloatN y);
_FloatN fmodfN(_FloatN x, _FloatN y);
DecimalN fmoddfN(_DecimalN x, _DecimalN y);
DecimalN fmodfN(_DecimalN x, _DecimalN y);

_FloatN remainderfN(_FloatN x, _FloatN y);
_FloatN remainderfNx(_FloatN x, _FloatN y);
DecimalN remainderdN(_DecimalN x, _DecimalN y);
DecimalN remainderdN(_DecimalN x, _DecimalN y);

_FloatN remquoN(_FloatN x, _FloatN y, int *quo);
_FloatN remquoNf(_FloatN x, _FloatN y, int *quo);

7.12.11 Manipulation functions

_FloatN copysignN(_FloatN x, _FloatN y);
_FloatN copysignfN(_FloatN x, _FloatN y);
DecimalN copysigndN(_DecimalN x, _DecimalN y);
DecimalN copysignfNx(_DecimalN x, _DecimalN y);

_FloatN nanfN(const char *tagp);
_FloatN nanzN(const char *tagp);
DecimalN nandN(const char *tagp);
DecimalN nanzN(const char *tagp);

_FloatN nextafterfN(_FloatN x, _FloatN y);
_FloatN nextafterfNx(_FloatN x, _FloatN y);
DecimalN nextafterdN(_DecimalN x, _DecimalN y);
DecimalN nextafterdN(_DecimalN x, _DecimalN y);

_FloatN nextupfN(_FloatN x);
_FloatN nextupfNf(_FloatN x);
DecimalN nextupdN(_DecimalN x);
DecimalN nextupdNf(_DecimalN x);

_FloatN nextdownfN(_FloatN x);
_FloatN nextdownfNx(_FloatN x);
DecimalN nextdowndN(_DecimalN x);
DecimalN nextdowndNf(_DecimalN x);

int canonicalizefN(_FloatN * cx, const _FloatN * x);
int canonicalizefNx(_FloatN * cx, const _FloatN * x);
int canonicalizedN(_DecimalN * cx, const _DecimalN * x);
int canonicalizedNf(_DecimalN * cx, const _DecimalN * x);

DecimalN quantizedN(_DecimalN x, _DecimalN y);
DecimalN quantizedNf(_DecimalN x, _DecimalN y);

_Bool samequantumdN(_DecimalN x, _DecimalN y);
_Bool samequantumdNf(_DecimalN x, _DecimalN y);

7.12.12 Maximum, minimum, and positive difference functions

\[
\begin{align*}
&_\text{Float}N \ fmaxf(_,FloatN x,_,FloatN y) \\
&_\text{Float}N \ fmaxfN(_,FloatN x,_,FloatN y) \\
&_\text{Decimal}N \ fmaxd(_,DecimalN x,_,DecimalN y) \\
&_\text{Decimal}N \ fmaxdN(_,DecimalN x,_,DecimalN y) \\
&_\text{Float}N \ fmaxmagf(_,FloatN x,_,FloatN y) \\
&_\text{Float}N \ fmaxmagfN(_,FloatN x,_,FloatN y) \\
&_\text{Decimal}N \ fmaxmagd(_,DecimalN x,_,DecimalN y) \\
&_\text{Decimal}N \ fmaxmagdN(_,DecimalN x,_,DecimalN y) \\
\end{align*}
\]

7.12.13 Floating multiply-add

\[
\begin{align*}
&_\text{Float}N \ fmaf(_,FloatN x,_,FloatN y,_,FloatN z) \\
&_\text{Float}N \ fmafN(_,FloatN x,_,FloatN y,_,FloatN z) \\
&_\text{Decimal}N \ fmadd(_,DecimalN x,_,DecimalN y,_,DecimalN z) \\
&_\text{Decimal}N \ fmaddN(_,DecimalN x,_,DecimalN y,_,DecimalN z) \\
\end{align*}
\]

7.12.14 Functions that round result to narrower format

\[
\begin{align*}
&_\text{Float}M \ fMaddfN(_,FloatN x,_,FloatN y); \ // M < N \\
&_\text{Float}M \ fMaddfNx(_,FloatN x,_,FloatN y); \ // M \leq N \\
&_\text{Float}Mx \ fMaddfN(_,FloatN x,_,FloatN y); \ // M < N \\
&_\text{Float}Mx \ fMaddfNx(_,FloatN x,_,FloatN y); \ // M < N \\
\end{align*}
\]
_DecimalM dMaddN(_DecimalN x, _DecimalN y);  // M < N
_decimalM dMaddNx(_DecimalN x, _DecimalN y);  // M <= N
_decimalM dMxaddNx(_DecimalN x, _DecimalN y);  // M < N

_FloatM fMsubfN(_FloatN x, _FloatN y);  // M < N
_FloatM fMsubfNx(_FloatN x, _FloatN y);  // M <= N
_FloatM fMxsubfN(_FloatN x, _FloatN y);  // M < N
_FLOATM dMsubdN(_DecimalN x, _DecimalN y);  // M < N
_decimalM dMsubdN(_DecimalN x, _DecimalN y);  // M <= N
_decimalM dMxsubdN(_DecimalN x, _DecimalN y);  // M < N

_FLOATM fMmulfN(_FloatN x, _FloatN y);  // M < N
_FLOATM fMmulfNx(_FloatN x, _FloatN y);  // M <= N
_FLOATM fMxmulfN(_FloatN x, _FloatN y);  // M < N
_decimalM dMmuldN(_DecimalN x, _DecimalN y);  // M < N
_decimalM dMmuldNx(_DecimalN x, _DecimalN y);  // M <= N
_decimalM dMxmuldN(_DecimalN x, _DecimalN y);  // M < N

_FLOATM fMdivfN(_FloatN x, _FloatN y);  // M < N
_FLOATM fMdivfNx(_FloatN x, _FloatN y);  // M <= N
_FLOATM fMxdivfN(_FloatN x, _FloatN y);  // M < N
_decimalM dMdivdN(_DecimalN x, _DecimalN y);  // M < N
_decimalM dMdivdNx(_DecimalN x, _DecimalN y);  // M <= N
_decimalM dMxdivdN(_DecimalN x, _DecimalN y);  // M < N

_FLOATM fMfmafN(_FloatN x, _FloatN y, _FloatN z);  // M < N
_FLOATM fMfmafNx(_FloatN x, _FloatN y, _FloatN z);  // M <= N
_FLOATM fMxfmafN(_FloatN x, _FloatN y, _FloatN z);  // M < N
_decimalM dMfmadN(_DecimalN x, _DecimalN y, _DecimalN z);  // M < N
_decimalM dMfmadNx(_DecimalN x, _DecimalN y, _DecimalN z);  // M <= N
_decimalM dMxfmadN(_DecimalN x, _DecimalN y, _DecimalN z);  // M < N

_FLOATM fMsqrtfN(_FloatN x);  // M < N
_FLOATM fMsqrtfNx(_FloatN x);  // M <= N
_FLOATM fMsqrtfN(_FloatN x);  // M < N
_decimalM dMsqrtdN(_DecimalN x);  // M < N
_decimalM dMsqrtdNx(_DecimalN x);  // M <= N
_decimalM dMsqrtdN(_DecimalN x);  // M < N
_decimalM dMsqrtdNx(_DecimalN x);  // M <= N
F.10.12 Total order functions

```c
int totalorderfN(_FloatN x, _FloatN y);
int totalorderfNx(_FloatNx x, _FloatNx y);
int totalorderdN(_DecimalN x, _DecimalN y);
int totalorderdNx(_DecimalNx x, _DecimalNx y);
int totalordermagfN(_FloatN x, _FloatN y);
int totalordermagfNx(_FloatNx x, _FloatNx y);
int totalordermagdN(_DecimalN x, _DecimalN y);
int totalordermagdNx(_DecimalNx x, _DecimalNx y);
```

F.10.13 Payload functions

```c
_FloatN getpayloadfN(const _FloatN *x);
_FloatN getpayloadfNx(const _FloatNx *x);
_DecimalN getpayloaddN(const _DecimalN *x);
_DecimalN getpayloaddNx(const _DecimalNx *x);
int setpayloadfN(_FloatN *res, _FloatN pl);
int setpayloadfNx(_FloatNx *res, _FloatNx pl);
int setpayloaddN(_DecimalN *res, _DecimalN pl);
int setpayloaddNx(_DecimalNx *res, _DecimalNx pl);
int setpayloadsigfN(_FloatN *res, _FloatN pl);
int setpayloadsigfNx(_FloatNx *res, _FloatNx pl);
int setpayloadsigdN(_DecimalN *res, _DecimalN pl);
int setpayloadsigdNx(_DecimalNx *res, _DecimalNx pl);
```

In 7.12.6.4#2, change the third sentence from:

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

To:

> If the type of the function is a standard or binary floating type, the exponent is an integral power of 2.

In 7.12.6.4#3, change the second sentence from:

> Otherwise, the `frexp` functions return the value `x`, such that: `x` has a magnitude in the interval `[1/2, 1)` or zero, and `value` equals `x × 2^`exp`, when the type of the function is a standard floating type; ...

To:

> Otherwise, the `frexp` functions return the value `x`, such that: `x` has a magnitude in the interval `[1/2, 1)` or zero, and `value` equals `x × 2^`exp`, when the type of the function is a standard or binary floating type; ...

In 7.12.6.6#2, change the first sentence from:

> The `ldexp` functions multiply a floating-point number by an integral power of 2 when the type of the function is a standard floating type, or by an integral power of 10 when the type of the function is a decimal floating type.

To:

> The `ldexp` functions multiply a floating-point number by an integral power of 2 when the type of the function is a standard floating type, or by an integral power of 10 when the type of the function is a decimal floating type.
The `ldexp` functions multiply a floating-point number by an integral power of 2 when the type of the function is a standard or binary floating type, or by an integral power of 10 when the type of the function is a decimal floating type.

Change 7.12.6.6#3 from:

[3] The `ldexp` functions return $x \times 2^{\text{exp}}$ when the type of the function is a standard floating type, or return $x \times 10^{\text{exp}}$ when the type of the function is a decimal floating type.

to:

[3] The `ldexp` functions return $x \times 2^{\text{exp}}$ when the type of the function is a standard or binary floating type, or return $x \times 10^{\text{exp}}$ when the type of the function is a decimal floating type.

In 7.12.6.11#2, change the second sentence from:

If $x$ is subnormal it is treated as though it were normalized; thus, for positive finite $x$,

$$1 \leq x \times b^{-\log_b(x)} < b$$

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.

to:

If $x$ is subnormal it is treated as though it were normalized; thus, for positive finite $x$,

$$1 \leq x \times b^{-\log_b(x)} < b$$

where $b = \text{FLT\_RADIX}$ if the type of the function is a standard floating type, $b = 2$ if the type of the function is a binary floating type, or $b = 10$ if the type of the function is a decimal floating type.

In 7.12.6.13#2, change the first sentence from:

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.

to:

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, $b = 2$ if the type of the function is a binary floating type, or $b = 10$ if the type of the function is a decimal floating type.

12.4 Encoding conversion functions

The functions in this subclause, together with the numerical conversion functions for encodings in clause 13, support the non-arithmetic interchange formats specified by IEC 60559.
Changes to C11 + TS18661-1 + TS18661-2:

After 7.12.11.7, add:

7.12.11.7a The encodefN functions

Synopsis

[1] #include <math.h>
   void encodefN(unsigned char * restrict encptr, const _FloatN * restrict xptr);

Description

[2] The encodefN functions convert *xptr into an IEC 60559 binaryN encoding 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


7.12.11.7b The decodefN functions

Synopsis

[1] #include <math.h>
   void decodefN(_FloatN * restrict xptr, const unsigned char * restrict encptr);

Description

[2] The decodefN functions interpret the N/8 element array pointed to by encptr as an IEC 60559 binaryN encoding, with 8 bits per array element. The order of bytes in the array is implementation-defined. These functions convert the given encoding into a representation in the type _FloatN, 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


7.12.11.7c Encoding-to-encoding conversion functions

[1] An implementation shall declare a fMencfN function for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall provide both dMencecdN and dMencebindN functions for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.
7.12.11.7c.1 The $f_{MencfN}$ functions

Synopsis

[1] #include <math.h>
    void $f_{MencfN}$(unsigned char * restrict $enc_Mptr$, const unsigned char * restrict $enc_Nptr$);

Description

[2] These functions convert between IEC 60559 binary interchange formats. These functions interpret the $N/8$ element array pointed to by $enc_Nptr$ as an encoding of width $N$ bits. They convert the encoding to an encoding of width $M$ bits and store the resulting encoding as an $M/8$ element array in the object pointed to by $enc_Mptr$. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

7.12.11.7c.2 The $d_{MencdecN}$ and $d_{MencbindN}$ functions

Synopsis

[1] #include <math.h>
    void $d_{MencdecN}$(unsigned char * restrict $enc_Mptr$, const unsigned char * restrict $enc_Nptr$);
    void $d_{MencbindN}$(unsigned char * restrict $enc_Mptr$, const unsigned char * restrict $enc_Nptr$);

Description

[2] These functions convert between IEC 60559 decimal interchange formats that use the same encoding scheme. The $d_{MencdecN}$ functions convert between formats using the encoding scheme based on decimal encoding of the significand. The $d_{MencbindN}$ functions convert between formats using the encoding scheme based on binary encoding of the significand. These functions interpret the $N/8$ element array pointed to by $enc_Nptr$ as an encoding of width $N$ bits. They convert the encoding to an encoding of width $M$ bits and store the resulting encoding as an $M/8$ element array in the object pointed to by $enc_Mptr$. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

In F.3, change the row:

<table>
<thead>
<tr>
<th>convertFormat – different formats</th>
<th>cast and implicit conversions</th>
<th>6.3.1.5, 6.5.4</th>
</tr>
</thead>
</table>

to:

| convertFormat – different formats | cast, implicit conversions, conversion functions (details below) | 6.3.1.5, 6.5.4, 7.12.11, 7.22.11b, 7.22.1 (details below) |
After F.3 [3], add:

[3a] C operations provide the convertFormat operations for the different kinds of IEC 60559 formats as follows:

— For conversions between arithmetic formats supported by floating types - casts and implicit conversions.
— For same-radix conversions between non-arithmetic interchange formats - encoding-to-encoding conversion functions (7.12.11.7c).
— For conversions between non-arithmetic interchange formats (same or different radix) – compositions of string-from-encoding functions (7.22.1.3c) (converting exactly) and string-to-encoding functions (7.22.1.3b).
— For same-radix conversions from interchange formats supported by interchange floating types to non-arithmetic interchange formats – compositions of encode functions (7.12.11.7a, 7.12.11b.1, 7.12.11b.3) and encoding-to-encoding (7.12.11.7c) functions.
— For same radix conversions from non-arithmetic interchange formats to interchange formats supported by interchange floating types – compositions of encoding-to-encoding conversion functions (7.12.11.7c) and decode functions (7.12.11.7b, 7.12.11b.2, 7.12.11b.4).
— For conversions from non-arithmetic interchange formats to arithmetic formats supported by floating types (same or different radix) – compositions of string-from-encoding functions (7.22.1.3c) (converting exactly) and strtof functions (7.22.1.3, 7.22.1.3a).
— For conversions from arithmetic formats supported by floating types to non-arithmetic interchange formats (same or different radix) – compositions of strfrom functions (7.22.1.2a) (converting exactly) and string-to-encoding functions (7.22.1.3b).

13 Numeric conversion functions in <stdlib.h>

This clause specifies functions to convert between character sequences and the interchange and extended floating types. Conversions from character sequences are provided by functions analogous to the strtod function in <stdlib.h>. Conversions to character sequences are provided by functions analogous to the strfromd function in <stdlib.h>.

This clause also specifies functions to convert between character sequences and IEC 60559 interchange format encodings.

Changes to C11 + TS18661-1 + TS18661-2:

After 7.22.1#1, insert

[1a] For each interchange or extended floating type that the implementation provides, <stdlib.h> shall declare the associated functions. Conversely, for each such type that the implementation does not provide, <stdlib.h> shall not declare the associated functions unless specified otherwise.

Replace 7.22.1.2a and 7.22.1.2b:

7.22.1.2a The strfromd, strfromf, and strfroml functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_BFP_EXT__
#include <stdlib.h>
int strfromd (char * restrict s, size_t n, const char * restrict format, double fp);
int strfromf (char * restrict s, size_t n, const char * restrict format, float fp);
int strfroml (char * restrict s, size_t n, const char * restrict format, long double fp);
Description

[2] The \texttt{strfromd}, \texttt{strfromf}, and \texttt{strfroml} functions are equivalent to \texttt{snprintf(s, n, format, fp)} (7.21.6.5), except the \texttt{format} string contains only the character \texttt{%}, an optional precision that does not contain an asterisk \texttt{*}, and one of the conversion specifiers \texttt{a}, \texttt{A}, \texttt{e}, \texttt{E}, \texttt{f}, \texttt{F}, \texttt{g}, or \texttt{G}, which applies to the type (\texttt{double}, \texttt{float}, or \texttt{long double}) indicated by the function suffix (rather than by a length modifier). Use of these functions with any other \texttt{format} string results in undefined behavior.

Returns

[3] The \texttt{strfromd}, \texttt{strfromf}, and \texttt{strfroml} functions return the number of characters that would have been written had \texttt{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 \texttt{n}.

7.22.1.2b The \texttt{strfromdN} functions

Synopsis

[1] 
\begin{verbatim}
#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);
\end{verbatim}

Description

[2] The \texttt{strfromdN} functions are equivalent to \texttt{snprintf(s, n, format, fp)} (7.21.6.5), except the \texttt{format} string contains only the character \texttt{%}, an optional precision that does not contain an asterisk \texttt{*}, and one of the conversion specifiers \texttt{a}, \texttt{A}, \texttt{e}, \texttt{E}, \texttt{f}, \texttt{F}, \texttt{g}, or \texttt{G}, which applies to the type (\texttt{Decimal32}, \texttt{Decimal64}, or \texttt{Decimal128}) indicated by the function suffix (rather than by a length modifier). Use of these functions with any other \texttt{format} string results in undefined behavior.

Returns

[3] The \texttt{strfromdN} functions return the number of characters that would have been written had \texttt{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 \texttt{n}. 
with:

7.22.1.2a The `strfromd`, `strfromf`, `strfroml`, `strfromfN`, `strfromfNx`, `strfromdN`, and `strfromdNx` functions

**Synopsis**

[1] `#include <stdlib.h>`

```c
int strfromd (char * restrict s, size_t n, const char * restrict format, double fp);
int strfromf (char * restrict s, size_t n, const char * restrict format, float fp);
int strfroml (char * restrict s, size_t n, const char * restrict format, long double fp);
int strfromfN(char * restrict s, size_t n, const char * restrict format, _Float fp);
int strfromfNx(char * restrict s, size_t n, const char * restrict format, _FloatN fp);
int strfromdN(char * restrict s, size_t n, const char * restrict format, _Decimal fp);
int strfromdNx(char * restrict s, size_t n, const char * restrict format, _DecimalN fp);
```

**Description**

[2] The `strfromd`, `strfromf`, `strfroml`, `strfromfN`, `strfromfNx`, `strfromdN`, and `strfromdNx` 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 (`double`, `float`, `long double`, `_FloatN`, `_FloatNx`, `_DecimalN`, or `_DecimalNx`) 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] These 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`.

Change the first part of 7.22.1.3:

**7.22.1.3 The `strtod`, `strtof`, and `strtold` functions**

**Synopsis**

[1] `#include <stdlib.h>`

```c
double strtod(const char * restrict nptr, char ** restrict endptr);
float strtof(const char * restrict nptr, char ** restrict endptr);
long double strtold (const char * restrict nptr, char ** restrict endptr);
```

**Description**

[2] The `strtod`, `strtof`, and `strtold` functions convert the initial portion of the string pointed to by `nptr` to `double`, `float`, and `long double` representation, respectively.
7.22.1.3 The `strtod`, `strtof`, `strtold`, `strtofN`, and `strtofNx` functions

Synopsis

[1]  

```c
#include <stdlib.h>

double strtod(const char * restrict nptr, char ** restrict endptr);
float strtof(const char * restrict nptr, char ** restrict endptr);
long double strtold (const char * restrict nptr, char ** restrict endptr);
_FloatN strtofN(const char * restrict nptr, char ** restrict endptr);
_FloatNx strtofNx(const char * restrict nptr, char ** restrict endptr);
```

Description

[2] The `strtod`, `strtof`, `strtold`, `strtofN`, and `strtofNx` functions convert the initial portion of the string pointed to by `nptr` to `double`, `float`, `long double`, `_FloatN`, and `_FloatNx` representation, respectively.

Change 7.22.1.3 #5:

[5] If the subject sequence has the hexadecimal form and `FLT_RADIX` is a power of 2, the value resulting from the conversion is correctly rounded.

to:

[5] If the subject sequence has the hexadecimal form and the radix of the return type of the function is a power of 2, the value resulting from the conversion is correctly rounded.

In 7.22.1.3 #8, change the first sentence:

[8] If the subject sequence has the hexadecimal form, `FLT_RADIX` is not a power of 2, ...

to:

[8] If the subject sequence has the hexadecimal form, the radix of the return type of the function is not a power of 2, and ...

In 7.22.1.3 #10, in the third sentence, change:

plus or minus `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` is returned

to:

plus or minus `HUGE_VAL`, `HUGE_VALF`, `HUGE_VALL`, `HUGE_VALFN`, or `HUGE_VALFNx` is returned
In 7.22.1.3a, change:

**Synopsis**

```c
[1] #define __STDC_WANT_IEC_60559_DFP_EXT__
#include <stdlib.h>
_Declimal32 strtod32(const char * restrict nptr, char ** restrict endptr);
_Declimal64 strtod64(const char * restrict nptr, char ** restrict endptr);
_Declimal128 strtod128(const char * restrict nptr, char ** restrict endptr);
```
to:

**Synopsis**

```c
[1] #include <stdlib.h>
_DeclimalN strtodN(const char * restrict nptr, char ** restrict endptr);
_DeclimalNx strtodNx(const char * restrict nptr, char ** restrict endptr);
```

After 7.22.1.3a, insert:

### 7.22.1.3b String-to-encoding functions

[1] An implementation shall declare the `strtencfN` function for each `N` equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall declare both the `strtoencdecN` and `strtoenbindN` functions for each `N` equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.

#### 7.22.1.3b.1 The `strtencfN` functions

**Synopsis**

```c
[1] #include <stdlib.h>
void strtoencfN(unsigned char * restrict encptr, const char * restrict nptr, char ** restrict endptr);
```

**Description**

[2] The `strtoencfN` functions are similar to the `strtofN` functions, except they store an IEC 60559 encoding of the result as an `N/8` element array in the object pointed to by `encptr`. The order of bytes in the arrays is implementation-defined.

**Returns**

[3] These functions return no value.
7.22.1.3b.2 The `strtoencdecN` and `strtoencbindN` functions

Synopsis

[1] `#include <stdlib.h>`
void strtoencdecN(unsigned char * restrict encptr, const char * restrict nptr, char ** restrict endptr);
void strtoencbindN(unsigned char * restrict encptr, const char * restrict nptr, char ** restrict endptr);

Description

[2] The `strtoencdecN` and `strtoencbindN` functions are similar to the `strtodN` functions, except they store an IEC 60559 encoding of the result as an N/8 element array in the object pointed to by `encptr`. The `strtoencdecN` functions produce an encoding in the encoding scheme based on decimal encoding of the significand. The `strtoencbindN` functions produce an encoding in the encoding scheme based on binary encoding of the significand. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

7.22.1.3c String-from-encoding functions

Synopsis

[1] #include <stdlib.h>
int strfromencfN(char * restrict s, size_t n, const char * restrict format, const unsigned char * restrict encptr);

Description

[2] The `strfromencfN` functions are similar to the `strfromfN` functions, except the input is the value of the N/8 element array pointed to by `encptr`, interpreted as an IEC 60559 binary N encoding. The order of bytes in the arrays is implementation-defined.

Returns

[3] The `strfromencfN` functions return the same values as corresponding `strfromfN` functions.

7.22.1.3c.2 The `strfromencdecN` and `strfromencbindN` functions

Synopsis

[1] #include <stdlib.h>
int strfromencdecN(char * restrict s, size_t n, const char * restrict format, const unsigned char * restrict encptr);
int strfromencbindN(char * restrict s, size_t n, const char * restrict format, const unsigned char * restrict encptr);
Description

[2] The strfromencdecd_N functions are similar to the strfromd_N functions except the input is the value of the N/8 element array pointed to by encptr, interpreted as an IEC 60559 decimal_N encoding in the coding scheme based on decimal encoding of the significand. The strfromencbind_N functions are similar to the strfromd_N functions except the input is the value of the N/8 element array pointed to by encptr, interpreted as an IEC 60559 decimal_N encoding in the coding scheme based on binary encoding of the significand. The order of bytes in the arrays is implementation-defined.

Returns

[3] The strfromencdecd_N and strfromencbind_N functions return the same values as corresponding strfromd_N functions.

14 Complex arithmetic <complex.h>

This clause specifies complex functions for corresponding real types that are interchange and extended floating types.

Changes to C11 + TS18661-1 + TS18661-2:

Change 7.3.1#3 from:

[3] Each synopsis specifies a family of functions consisting of a principal function with one or more double complex parameters and a double complex or double return value; and other functions with the same name but with f and l suffixes which are corresponding functions with float and long double parameters and return values.

to:

[3] Each synopsis specifies a family of functions consisting of:

a principal function with one or more double complex parameters and a double complex or double return value; and,

other functions with the same name but with f, l, fN, and fNx suffixes which are corresponding functions whose parameters and return values have corresponding real types float, long double, _Float_N, and _Float_Nx.

Add after 7.3.1#3:

[3a] For each interchange or extended floating type that the implementation provides, <complex.h> shall declare the associated functions. Conversely, for each such type that the implementation does not provide, <complex.h> shall not declare the associated functions.

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

7.3.5 Trigonometric functions

_Float_N complex cacosfN(_Float_N complex z);
_Float_Nx complex cacosfNx(_Float_Nx complex z);
_Float_N complex casinfN(_Float_N complex z);
_Float_Nx complex casinfNx(_Float_Nx complex z);
7.3.6 Hyperbolic functions

_FloatN complex cacoshf(_FloatN complex z);
_FloatN complex cacoshfN(_FloatN complex z);
_FloatN complex casinhf(_FloatN complex z);
_FloatN complex casinhfN(_FloatN complex z);
_FloatN complex catanhf(_FloatN complex z);
_FloatN complex catanhfN(_FloatN complex z);
_FloatN complex ccoshf(_FloatN complex z);
_FloatN complex ccoshfN(_FloatN complex z);
_FloatN complex csinhf(_FloatN complex z);
_FloatN complex csinhfN(_FloatN complex z);
_FloatN complex ctanhf(_FloatN complex z);
_FloatN complex ctanhfN(_FloatN complex z);

7.3.7 Exponential and logarithmic functions

_FloatN complex cexpf(_FloatN complex z);
_FloatN complex cexpfN(_FloatN complex z);
_FloatN complex clogf(_FloatN complex z);
_FloatN complex clogfN(_FloatN complex z);

7.3.8 Power and absolute value functions

_FloatN cabsf(_FloatN complex z);
_FloatN complex cabsfN(_FloatN complex z);
_FloatN complex cpowf(_FloatN complex x, _FloatN complex y);
_FloatN complex cpowfN(_FloatN complex x, _FloatN complex y);
_FloatN complex csqrtf(_FloatN complex z);
_FloatN complex csqrtfN(_FloatN complex z);
7.3.9 Manipulation functions

```c
_FloatN cargfN(_FloatN complex z);
_FloatNx cargfNx(_FloatNx complex z);

_FloatN cimagfN(_FloatN complex z);
_FloatNx cimagfNx(_FloatNx complex z);

_FloatN complex CMPLXFN(_FloatN x, _FloatN y);
_FloatNx complex CMPLXFNx(_FloatNx x, _FloatNx y);

_FloatN complex conjfN(_FloatN complex z);
_FloatNx complex conjfNx(_FloatNx complex z);

_FloatN complex cprojfN(_FloatN complex z);
_FloatNx complex cprojfNx(_FloatNx complex z);

_FloatN crealfN(_FloatN complex z);
_FloatNx crealfNx(_FloatNx complex z);
```

20 In 7.31.1, change:

```
… and the same names suffixed with f or l may be added to the declarations in the <complex.h> header.
```

to:

```
… and the same names suffixed with f, l, fn, or fnx may be added to the declarations in the <complex.h> header.
```

15 Type-generic macros <tgmath.h>

The following changes to C11 + TS18661-1 + TS18661-2 enhance the specification of type-generic macros in <tgmath.h> to apply to interchange and extended floating types, as well as standard floating types.

Changes to C11 + TS18661-1 + TS18661-2:

30 In 7.25, replace paragraph #3b:

```
[3b] If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type, the behavior is undefined.
```

with:

```
[3b] If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is a standard floating type or a floating type of radix 2 and another argument is of decimal floating type, the behavior is undefined.
```

In 7.25#3c, replace the bullets:

```
— First, if any argument for generic parameters has type _Decimal128, the type determined is _Decimal128.
```
— Otherwise, if any argument for generic parameters has type _Decimal64, or if any argument for
  generic parameters is of integer type and another argument for generic parameters has type
  _Decimal32, the type determined is _Decimal64.

— Otherwise, if any argument for generic parameters has type _Decimal32, the type determined is
  _Decimal32.

— Otherwise, if the corresponding real type of any argument for generic parameters is long
double, the type determined is long double.

— Otherwise, if the corresponding real type of any argument for generic parameters is double or is
  of integer type, the type determined is double.

— Otherwise, if any argument for generic parameters is of integer type, the type determined is
double.

— Otherwise, the type determined is float.

with:

— If two arguments have floating types and neither of the sets of values of their corresponding real
types is a subset of (or equivalent to) the other, the behavior is undefined.

— If any arguments for generic parameters have type _DecimalM where M ≥ 64 or _DecimalNx
  where N ≥ 32, the type determined is the widest of the types of these arguments. If _DecimalM
  and _DecimalNx are both widest types (with equivalent sets of values) of these arguments, the
type determined is _DecimalM.

— Otherwise, if any argument for generic parameters is of integer type and another argument for
  generic parameters has type _Decimal32, the type determined is _Decimal64.

— Otherwise, if any argument for generic parameters has type _Decimal32, the type determined is
  _Decimal32.

— Otherwise, if the corresponding real type of any argument for generic parameters has type long
double, _FloatM where M ≥ 128, or _FloatNx where N ≥ 64, the type determined is the widest of the
  corresponding real types of these arguments. If _FloatM and either long double or _FloatNx are both
  widest corresponding real types (with equivalent sets of values) of these arguments, the type
determined is _FloatM. Otherwise, if long double and _FloatNx are both widest corresponding real
  types (with equivalent sets of values) (with equivalent sets of values) of these arguments, the type
determined is long double.

— Otherwise, if the corresponding real type of any argument for generic parameters has type
double, _Float64, or _Float32x, the type determined is the widest of the corresponding real
types of these arguments. If _Float64 and either double or _Float32x are both widest
  corresponding real types (with equivalent sets of values) of these arguments, the type determined
  is _Float64. Otherwise, if double and _Float32x are both widest corresponding real types
  (with equivalent sets of values) of these arguments, the type determined is double.

— Otherwise, if any argument for generic parameters is of integer type, the type determined is
double.

— Otherwise, if the corresponding real type of any argument for generic parameters has type
  _Float32, the type determined is _Float32.

— Otherwise, the type determined is float.
In the second bullet 7.25#3c, attach a footnote to the wording:

the type determined is the widest

where the footnote is:

*) The term widest here refers to a type whose set of values is a superset of (or equivalent to) the sets of values of the other types.

In 7.25#6, replace:

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.

with:

Use of the macro with any argument of standard floating type, floating type of radix 2, or complex type, invokes a complex function. Use of the macro with an argument of a decimal floating type results in undefined behavior.

After 7.25#6c, add the paragraph:

[6d] For an implementation that provides the following real floating types:

<table>
<thead>
<tr>
<th>type</th>
<th>IEC 60559 format</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>binary32</td>
</tr>
<tr>
<td>double</td>
<td>binary64</td>
</tr>
<tr>
<td>long double</td>
<td>binary128</td>
</tr>
<tr>
<td>_Float32</td>
<td>binary32</td>
</tr>
<tr>
<td>_Float64</td>
<td>binary64</td>
</tr>
<tr>
<td>_Float128</td>
<td>binary128</td>
</tr>
<tr>
<td>_Float32x</td>
<td>binary64</td>
</tr>
<tr>
<td>_Float64x</td>
<td>binary128</td>
</tr>
</tbody>
</table>

a type-generic macro `cbrt` that conforms to the specification in this clause and that is affected by constant rounding modes could be implemented as follows:

```c
#define cbrt(X) _Generic((X),
  _Float128: cbrtf128(X),
  _Float64: cbrtf64(X),
  _Float32: cbrtf32(X),
  _Float64x: cbrtf64x(X),
  _Float32x: cbrtf32x(X),
  long double: cbrtl(X),
  default: _Roundwise_cbrt(X),
  float: cbrtf(X)
)
```

where `_Roundwise_cbrt()` is equivalent to `cbrt()` invoked without macro-replacement suppression.
In 7.25#7, insert at the beginning of the example:

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
```

In 7.25#7, append to the declarations:

```
#if __STDC_IEC_60559_TYPES__ >= 201ymmL
  _Float32x f32x;
  _Float64 f64;
  _Float128 f128;
  _Float64x complex f64xc;
#endif
```

In 7.25#7, append to the table:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cos(f64xc)</td>
<td>ccosf64x(f64xc)</td>
</tr>
<tr>
<td>pow(dc, f128)</td>
<td>cpowf128(dc, f128)</td>
</tr>
<tr>
<td>fmax(f64, d)</td>
<td>fmaxf64(f64, d)</td>
</tr>
<tr>
<td>fmax(d, f32x)</td>
<td>fmax(d, f32x), the function, if the set of values of _Float32x is a subset of (or equivalent to) the set of values of double, or fmaxf32x(d, f32x), if the set of values of double is a proper subset of the set of values of _Float32x, or undefined, if neither of the sets of values of double and _Float32x is a subset of the other (and the sets are not equivalent)</td>
</tr>
<tr>
<td>pow(f32x, n)</td>
<td>powf32x(f32x, n)</td>
</tr>
</tbody>
</table>
Bibliography


