ISO/IEC JTC 1/SC 22/WG 14 N1784

Date: yyyy-mm-dd

Reference number of document: ISO/IEC TS 18661-3

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

Secretariat: ANSI

Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 3: Interchange and extended types

Technologies de l’information — Langages de programmation, leurs environnements et interfaces du logiciel système — Extensions à virgule flottante pour C — Partie 3: Types d’échange et prolongée

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.
Foreword

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

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

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

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

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

ISO/IEC TS 18661 consists of the following parts, under the general title 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 IEC 60559:2011 international standard is equivalent to the IEEE 754-2008 standard for floating-point arithmetic, which is a major revision to IEEE 754-1985.

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

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

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

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

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

The revised standard, like its predecessor, defines 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.


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

**Purpose**

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

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

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

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

**Additional background on formats**

The 2011 revision of the ISO/IEC 60559 standard for floating-point arithmetic 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 \( k \geq 128 \) and a multiple of 32, and
  - for decimal, \( k \geq 32 \) 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 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 Technical Specification 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 Part 1 of Technical Specification 18661) signifying that it
supports Annex F of the C Standard, then its \texttt{float} and \texttt{double} formats must be IEC 60559 binary32 and binary64.

Part 2 of Technical Specification 18661 defines types \_\texttt{Decimal32}, \_\texttt{Decimal64}, and \_\texttt{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, Part 2 of Technical Specification 18661 has full arithmetic and library support for \_\texttt{Decimal32}, just like for \_\texttt{Decimal64} and \_\texttt{Decimal128}.

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

Part 3 of Technical Specification 18661, 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 types that, in some cases, already have C nomenclatures. For example, a type with the IEC 60559 double format may be referred to as \texttt{double}, \_\texttt{Float64} (the type for the binary64 interchange format), and maybe \_\texttt{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 document, Part 3 of Technical Specification 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 Part 3 of Technical Specification 18661 if

a) It meets the requirements for a conforming implementation of C11 with all the changes to C11 as specified in Parts 1-3 of Technical Specification 18661;

b) It conforms to Part 1 or Part 2 (or both) of Technical Specification 18661; and

c) It defines __STDC_IEC_60559_TYPES__ to 201.000.

3 Normative references

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

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

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


ISO/IEC 18661-1:yyyy, Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 1: Binary floating-point arithmetic

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


4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2011 and ISO/IEC/IEEE 60559:2011 and the following apply.
4.1
C11

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 Technical Specification 18661.

5.2 Predefined macros

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

In 6.10.8.3#1, change:

    __STDC_IEC_60559_DFP__    The integer constant 201ymmL, intended to indicate support of the decimal floating types and conformance with Annex F for IEC 60559 decimal floating-point arithmetic.

to:

    __STDC_IEC_60559_DFP__    The integer constant 201ymmL, intended to indicate support of the decimal floating types _Decimal32, _Decimal64, and _Decimal128 and conformance with Annex F for IEC 60559 decimal floating-point arithmetic.

In 6.10.8.3#1, add:

    __STDC_IEC_60559_TYPES__    The integer constant 201ymmL, intended to indicate support of interchange and extended floating types according to IEC 60559.

5.3 Standard headers

The new identifiers added to C11 library headers by this Part of Technical Specification 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:

- FLT\text{N\_MANT\_DIG}
- FLT\text{N\_DECIMAL\_DIG}
- FLT\text{N\_DIG}
- FLT\text{N\_MIN\_EXP}
- FLT\text{N\_MAX\_EXP}
- FLT\text{N\_MAX\_10\_EXP}
- FLT\text{N\_EPSILON}
- FLT\text{N\_MIN}
- FLT\text{N\_MAX}
- FLT\text{N\_TRUE\_MIN}

for supported types _FloatNx:

- FLT\text{N\_MANT\_DIG}
- FLT\text{N\_DECIMAL\_DIG}
- FLT\text{N\_DIG}
- FLT\text{N\_MIN\_EXP}
- FLT\text{N\_MAX\_EXP}
- FLT\text{N\_MAX\_10\_EXP}
- FLT\text{N\_EPSILON}
- FLT\text{N\_MIN}
- FLT\text{N\_MAX}
- FLT\text{N\_TRUE\_MIN}

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

- DEC\text{N\_MANT\_DIG}
- DEC\text{N\_MIN\_EXP}
- DEC\text{N\_MAX\_EXP}
- DEC\text{N\_MAX}
- DEC\text{N\_MIN}
- DEC\text{N\_EPSILON}
- DEC\text{N\_MIN}

for supported types _DecimalNx:

- DEC\text{N\_MANT\_DIG}
- DEC\text{N\_MIN\_EXP}
- DEC\text{N\_MAX\_EXP}
- DEC\text{N\_MIN}
- DEC\text{N\_MAX}
- DEC\text{N\_EPSILON}
- DEC\text{N\_MIN}

20 After 7.12#1c, insert the following 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 <math.h> is first included:
for supported types _FloatN:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_FN</td>
<td>modffN</td>
</tr>
<tr>
<td>SNANF0</td>
<td>scalbnfN</td>
</tr>
<tr>
<td>FP_FAST_FMAF</td>
<td>scalblnfN</td>
</tr>
<tr>
<td>acosfN</td>
<td>cbrtfN</td>
</tr>
<tr>
<td>asinfN</td>
<td>fabsfN</td>
</tr>
<tr>
<td>atanfN</td>
<td>hypotfN</td>
</tr>
<tr>
<td>atan2fN</td>
<td>powfN</td>
</tr>
<tr>
<td>cosfN</td>
<td>sqrtfN</td>
</tr>
<tr>
<td>sinfN</td>
<td>erfN</td>
</tr>
<tr>
<td>tanfN</td>
<td>erfcfN</td>
</tr>
<tr>
<td>acoshfN</td>
<td>lgammafN</td>
</tr>
<tr>
<td>asinhfN</td>
<td>gammalnfN</td>
</tr>
<tr>
<td>expfN</td>
<td>floorfN</td>
</tr>
<tr>
<td>exp2fN</td>
<td>nearbyintfN</td>
</tr>
<tr>
<td>expm1fN</td>
<td>rintfN</td>
</tr>
<tr>
<td>frexpfn</td>
<td>lrintfN</td>
</tr>
<tr>
<td>ilogbfN</td>
<td>llrintfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>roundfN</td>
</tr>
<tr>
<td>ldexpfN</td>
<td>lroundfN</td>
</tr>
<tr>
<td>logfN</td>
<td>llroundfN</td>
</tr>
<tr>
<td>log10fN</td>
<td>truncfN</td>
</tr>
<tr>
<td>log1pfN</td>
<td>roundevenfN</td>
</tr>
<tr>
<td>log2fN</td>
<td>fromfpfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>ufomfpfN</td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_FNX</td>
<td>logbfxN</td>
</tr>
<tr>
<td>SNANF0</td>
<td>modffN</td>
</tr>
<tr>
<td>FP_FAST_FMAF</td>
<td>scalblnfN</td>
</tr>
<tr>
<td>acosfN</td>
<td>cbrtfN</td>
</tr>
<tr>
<td>asinfN</td>
<td>fabsfN</td>
</tr>
<tr>
<td>atanfN</td>
<td>hypotfN</td>
</tr>
<tr>
<td>atan2fN</td>
<td>powfN</td>
</tr>
<tr>
<td>cosfN</td>
<td>sqrtfN</td>
</tr>
<tr>
<td>sinfN</td>
<td>erfN</td>
</tr>
<tr>
<td>tanfN</td>
<td>erfcfN</td>
</tr>
<tr>
<td>acoshfN</td>
<td>lgammafN</td>
</tr>
<tr>
<td>asinhfN</td>
<td>gammalnfN</td>
</tr>
<tr>
<td>expfN</td>
<td>floorfN</td>
</tr>
<tr>
<td>exp2fN</td>
<td>nearbyintfN</td>
</tr>
<tr>
<td>expm1fN</td>
<td>rintfN</td>
</tr>
<tr>
<td>frexpfn</td>
<td>lrintfN</td>
</tr>
<tr>
<td>ilogbfN</td>
<td>llrintfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>roundfN</td>
</tr>
<tr>
<td>ldexpfN</td>
<td>lroundfN</td>
</tr>
<tr>
<td>logfN</td>
<td>llroundfN</td>
</tr>
<tr>
<td>log10fN</td>
<td>truncfN</td>
</tr>
<tr>
<td>log1pfN</td>
<td>roundevenfN</td>
</tr>
<tr>
<td>log2fN</td>
<td>fromfpfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>ufomfpfN</td>
</tr>
</tbody>
</table>

for supported types _FloatNz:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_FNZ</td>
<td>logbfnz</td>
</tr>
<tr>
<td>SNANFN</td>
<td>modffN</td>
</tr>
<tr>
<td>FP_FAST_FMAFN</td>
<td>scalblnfN</td>
</tr>
<tr>
<td>acosfN</td>
<td>cbrtfN</td>
</tr>
<tr>
<td>asinfN</td>
<td>fabsfN</td>
</tr>
<tr>
<td>atanfN</td>
<td>hypotfN</td>
</tr>
<tr>
<td>atan2fN</td>
<td>powfN</td>
</tr>
<tr>
<td>cosfN</td>
<td>sqrtfN</td>
</tr>
<tr>
<td>sinfN</td>
<td>erfN</td>
</tr>
<tr>
<td>tanfN</td>
<td>erfcfN</td>
</tr>
<tr>
<td>acoshfN</td>
<td>lgammafN</td>
</tr>
<tr>
<td>asinhfN</td>
<td>gammalnfN</td>
</tr>
<tr>
<td>expfN</td>
<td>floorfN</td>
</tr>
<tr>
<td>exp2fN</td>
<td>nearbyintfN</td>
</tr>
<tr>
<td>expm1fN</td>
<td>rintfN</td>
</tr>
<tr>
<td>frexpfn</td>
<td>lrintfN</td>
</tr>
<tr>
<td>ilogbfN</td>
<td>llrintfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>roundfN</td>
</tr>
<tr>
<td>ldexpfN</td>
<td>lroundfN</td>
</tr>
<tr>
<td>logfN</td>
<td>llroundfN</td>
</tr>
<tr>
<td>log10fN</td>
<td>truncfN</td>
</tr>
<tr>
<td>log1pfN</td>
<td>roundevenfN</td>
</tr>
<tr>
<td>log2fN</td>
<td>fromfpfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>ufomfpfN</td>
</tr>
</tbody>
</table>

50

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log10fN</td>
<td>llroundfN</td>
</tr>
<tr>
<td>log1pfN</td>
<td>truncfN</td>
</tr>
<tr>
<td>log2fN</td>
<td>roundevenfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>ufomfpfN</td>
</tr>
</tbody>
</table>
for supported types \_Float\text{M} and \_Float\text{N} where \text{M} < \text{N}:

\begin{align*}
\text{fM} & \text{addfN} & \text{fM} & \text{mulfN} & \text{fM} & \text{sqrtfN} \\
\text{fM} & \text{subfN} & \text{fM} & \text{divfN} & \text{fM} & \text{fmafN}
\end{align*}

for supported types \_Float\text{M} and \_Float\text{N}x where \text{M} \leq \text{N}:

\begin{align*}
5 & \text{fM} & \text{addfNx} & \text{fM} & \text{mulfNx} & \text{fM} & \text{sqrtfNx} \\
& \text{fM} & \text{subfNx} & \text{fM} & \text{divfNx} & \text{fM} & \text{fmafNx}
\end{align*}

for supported types \_Float\text{M}x and \_Float\text{N} where \text{M} < \text{N}:

\begin{align*}
\text{fM} & \text{xaddfN} & \text{fM} & \text{xmulfN} & \text{fM} & \text{xsqrtfN} \\
\text{fM} & \text{xsubfN} & \text{fM} & \text{xdivfN} & \text{fM} & \text{xfmafN}
\end{align*}

for supported types \_Float\text{M}x and \_Float\text{N}x where \text{M} < \text{N}:

\begin{align*}
\text{fM} & \text{xaddfNx} & \text{fM} & \text{xmulfNx} & \text{fM} & \text{xsqrtfNx} \\
\text{fM} & \text{xsubfNx} & \text{fM} & \text{xdivfNx} & \text{fM} & \text{xfmafNx}
\end{align*}

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

\begin{align*}
\text{fM} & \text{encfN}
\end{align*}

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

\begin{align*}
\text{HUGE} & \text{VAL}_\text{D}\text{N} & \text{scalblndN} & \text{remquodN} \\
\text{SNAND} & \text{cbrtdN} & \text{copsigndN} \\
\text{FP FAST FMAD} & \text{fabsdN} & \text{nandN} \\
\text{acosdN} & \text{hypotdN} & \text{nextafterdN} \\
\text{asindN} & \text{powdN} & \text{nextupdN} \\
\text{atanandN} & \text{sqrtdN} & \text{nextdowndN} \\
\text{atan2dN} & \text{erfdN} & \text{canonicalizedN} \\
\text{cosdN} & \text{erfcdN} & \text{quantizedN} \\
\text{sindN} & \text{lgammadN} & \text{samequantumedN} \\
\text{tanandN} & \text{tgammadN} & \text{quantumN} \\
\text{acoshdN} & \text{ceildN} & \text{llquantexpdN} \\
\text{asinhdN} & \text{floordN} & \text{encodedcN} \\
\text{atanhdN} & \text{nearbyintdN} & \text{decodedcN} \\
\text{expdN} & \text{rintdN} & \text{encodebindN} \\
\text{exp2dN} & \text{lrintdN} & \text{decodebindN} \\
\text{expmldN} & \text{llrintdN} & \text{fdimdN} \\
\text{frexpN} & \text{rounddN} & \text{fmaxdN} \\
\text{i1logbdN} & \text{lrounddN} & \text{fmindN} \\
\text{i1logbdN} & \text{llrounddN} & \text{fmaxmagdN} \\
\text{ldexpdN} & \text{truncdN} & \text{fminmagdN} \\
\text{logdN} & \text{roundevendN} & \text{fmadN} \\
\text{log10dN} & \text{fromfpdN} & \text{totalorderdN} \\
\text{log1pdN} & \text{ufromfpdN} & \text{totalordermagdN} \\
\text{log2dN} & \text{fromfpxdN} & \text{getpayloadN} \\
\text{logbdN} & \text{ufromfpzdN} & \text{setpayloadN} \\
\text{modfdN} & \text{fmoddN} & \text{setpayloadsigdN} \\
\text{scalbndN} & \text{remainderdN}
\end{align*}
for supported types \_DecimalNX:

HUGE \_VAL_\_D\_NX  scalbn\_D\_Nx  fmod\_D\_Nx  
SNAN\_D\_NX  scalbln\_D\_Nx  remainder\_D\_Nx  
FP\_FAST\_FMAD\_D\_NX  cbrt\_D\_Nx  remquo\_D\_Nx  
acsdf\_D\_NX  fabs\_D\_Nx  copysign\_D\_Nx  
asind\_D\_NX  hypot\_D\_Nx  nand\_D\_Nx  
atand\_D\_NX  pow\_D\_Nx  nextafter\_D\_Nx  
atan2d\_NX  sqrt\_D\_Nx  nextup\_D\_Nx  
cosd\_NX  erf\_D\_Nx  nextdown\_D\_Nx  
sind\_NX  erfc\_D\_Nx  canonicalized\_D\_Nx  
tand\_NX  lgamma\_D\_Nx  quantized\_D\_Nx  
acoshd\_NX  tgamma\_D\_Nx  samequantum\_D\_Nx  
asinhd\_NX  ceild\_D\_Nx  quantum\_D\_Nx  
atanhd\_NX  floord\_D\_Nx  llquantexp\_D\_Nx  
expd\_NX  nearbyint\_D\_Nx  fdim\_D\_Nx  
expreal\_NX  rint\_D\_Nx  fma\_D\_Nx  
expmld\_NX  lrint\_D\_Nx  fm\_D\_Nx  
frexp\_D\_NX  llrint\_D\_Nx  fmaxmag\_D\_Nx  
ilogbd\_NX  roundd\_D\_Nx  fminmag\_D\_Nx  
15  logd\_NX  truncd\_D\_Nx  totalordermag\_D\_Nx  
log10d\_NX  round\_r\_D\_Nx  totalorder\_D\_Nx  
log1pd\_NX  fromfpd\_D\_Nx  setpayload\_D\_Nx  
25  log2d\_NX  ufromfpd\_D\_Nx  setpayload\_\_s\_D\_N\_X  
logbd\_NX  fromfp\_D\_D\_N\_X  
mod\_D\_N  ufromfp\_D\_D\_N\_X

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

30  FP\_FAST\_D\_M\_ADD\_N\_X  FP\_FAST\_D\_M\_S\_Q\_R\_T\_D\_N\_X  d\_M\_muld\_N\_X  
FP\_FAST\_D\_M\_SUB\_D\_N\_X  FP\_FAST\_D\_M\_F\_M\_A\_D\_N\_X  d\_M\_d\_ivd\_N\_X  
FP\_FAST\_D\_M\_M\_U\_L\_D\_N\_X  d\_M\_add\_N\_X  d\_M\_sq\_r\_t\_d\_N\_X  
FP\_FAST\_D\_M\_D\_I\_V\_D\_N\_X  d\_M\_subd\_N\_X  d\_M\_f\_m\_a\_d\_N\_X

for supported types \_DecimalM and \_DecimalN where \textit{M} ≤ \textit{N}:

35  FP\_FAST\_D\_M\_A\_D\_D\_N\_X  FP\_FAST\_D\_M\_S\_Q\_R\_T\_D\_N\_X  d\_M\_muld\_N\_X  
FP\_FAST\_D\_M\_S\_U\_B\_D\_N\_X  FP\_FAST\_D\_M\_F\_M\_A\_D\_N\_X  d\_M\_d\_ivd\_N\_X  
FP\_FAST\_D\_M\_M\_U\_L\_D\_N\_X  d\_M\_add\_N\_X  d\_M\_sq\_r\_t\_d\_N\_X  
FP\_FAST\_D\_M\_D\_I\_V\_D\_N\_X  d\_M\_subd\_N\_X  d\_M\_f\_m\_a\_d\_N\_X

for supported types \_DecimalMx and \_DecimalN where \textit{M} < \textit{N}:

40  FP\_FAST\_D\_M\_X\_A\_D\_D\_N\_X  FP\_FAST\_D\_M\_X\_S\_Q\_R\_T\_D\_N\_X  d\_M\_x\_m\_u\_l\_d\_N\_X  
FP\_FAST\_D\_M\_X\_S\_U\_B\_D\_N\_X  FP\_FAST\_D\_M\_X\_F\_M\_A\_D\_N\_X  d\_M\_x\_divd\_N\_X  
FP\_FAST\_D\_M\_X\_M\_U\_L\_D\_N\_X  d\_M\_x\_add\_N\_X  d\_M\_x\_q\_r\_t\_d\_N\_X  
FP\_FAST\_D\_M\_X\_D\_I\_V\_D\_N\_X  d\_M\_x\_subd\_N\_X  d\_M\_x\_f\_m\_a\_d\_N\_X
for supported types \_DecimalMx and \_DecimalNx where \( M < N \):

- \texttt{FP\_FAST\_DM\_XADDD/NX}
- \texttt{FP\_FAST\_DM\_XSQRTD/NX}
- \texttt{FP\_FAST\_DM\_XFMAD/NX}
- \texttt{FP\_FAST\_DM\_XMULD/NX}
- \texttt{FP\_FAST\_DM\_XDIVD/NX}

for supported types \_FloatN:

- \texttt{strfromfN}
- \texttt{strtofN}

for supported types \_FloatNx:

- \texttt{strfromfNx}
- \texttt{strtofNx}

for supported types \_DecimalN, where \( N \neq 32, 64, \) and \( 128 \):

- \texttt{strfromdN}
- \texttt{strtodN}

for supported types \_DecimalNx:

- \texttt{strfromdNx}
- \texttt{strtodNx}

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

- \texttt{dMencdecN}
- \texttt{dMencbindN}

After 7.22#1b, insert the paragraph:

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

for supported types \_FloatN:

- \texttt{strfromfN}
- \texttt{strtofN}

for supported types \_FloatNx:

- \texttt{strfromfNx}
- \texttt{strtofNx}

for supported types \_DecimalN, where \( N \neq 32, 64, \) and \( 128 \):

- \texttt{strfromdN}
- \texttt{strtodN}

for supported types \_DecimalNx:

- \texttt{strfromdNx}
- \texttt{strtodNx}

for supported IEC 60559 arithmetic and non-arithmetic binary interchange formats of width \( N \):

- \texttt{strfromencfN}
- \texttt{strtoencfN}

for supported IEC 60559 arithmetic and non-arithmetic decimal interchange formats of width \( N \):

- \texttt{strfromencdecN}
- \texttt{strtoencdecN}
- \texttt{strfromencbindN}
- \texttt{strtoencbindN}

6 Types

This clause specifies changes to C11 + TS18661-1 + TS18661-2 to include types that support the arithmetic interchange formats and extended formats specified in IEC 60559. The encoding conversion functions (11.3) and numeric conversion functions for encodings (12) support the non-arithmetic interchange formats specified in IEC 60559.

30 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.
Together, the standard floating types and the decimal floating types comprise the real floating types.

with:

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.

Table 1 – 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 – round(4×log2(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^((N−p−1)/13) – 1</td>
</tr>
</tbody>
</table>

Encoding 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>bias, E−e</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
<td>emax</td>
</tr>
<tr>
<td>sign bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>w, exponent field width in bits</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>round(4×log2(N)) – 13</td>
</tr>
<tr>
<td>t, trailing significand field width in bits</td>
<td>10</td>
<td>23</td>
<td>52</td>
<td>112</td>
<td>N − w − 1</td>
</tr>
<tr>
<td>N, storage width in bits</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>1 + w + t</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 2 – Decimal interchange format parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>decimal32</th>
<th>decimal64</th>
<th>decimal128</th>
<th>decimalN (N ≥ 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>
</tr>
<tr>
<td>p, precision in digits</td>
<td>7</td>
<td>16</td>
<td>34</td>
<td>9 × N/32 − 2</td>
</tr>
<tr>
<td>emax, maximum exponent e</td>
<td>96</td>
<td>384</td>
<td>6144</td>
<td>3 × 2^{(N−16−3)}</td>
</tr>
</tbody>
</table>

Encoding parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>decimal32</th>
<th>decimal64</th>
<th>decimal128</th>
<th>decimalN (N ≥ 32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bias, E−e</td>
<td>101</td>
<td>398</td>
<td>6176</td>
<td>emax + p − 2</td>
</tr>
<tr>
<td>sign bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>w, exponent field width in bits</td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>N/16 + 9</td>
</tr>
<tr>
<td>t, trailing significand field width in bits</td>
<td>20</td>
<td>50</td>
<td>110</td>
<td>15×N/16 − 10</td>
</tr>
<tr>
<td>N, storage width in bits</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>1 + 5 + w + t</td>
</tr>
</tbody>
</table>

For example, decimal256 would have p = 70 and emax = 1572864.
[10b] Types designated

\_Decima1\_N, where \( N \geq 32 \) and a multiple of 32

are collectively called the decimal floating types. The decimal floating types together with types
designated

\_FloatN, where \( N \) is 16, 32, 64, or \( \geq 128 \) 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 \_FloatN, 10 for

\_Decima1\_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 \_FloatN\_ as an interchange floating type with the same representation and alignment requirements as \_long\_ \_double\_. The implementation may provide other binary interchange floating types.

[10d] An implementation that defines \_STDC\_IEC\_60559\_DFP\_ shall provide the decimal floating types \_Decima1\_32\_, \_Decima1\_64\_, and \_Decima1\_128\_. If the implementation also defines \_STDC\_IEC\_60559\_TYPES\_, it may provide other decimal floating types.

[10e] 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.

[10f] 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>( p ) digits</th>
<th>( e_{\text{max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary32</td>
<td>32</td>
<td>1023</td>
</tr>
<tr>
<td>binary64</td>
<td>64</td>
<td>16383</td>
</tr>
<tr>
<td>binary128</td>
<td>128</td>
<td>65535</td>
</tr>
<tr>
<td>decimal64</td>
<td>22</td>
<td>6144</td>
</tr>
<tr>
<td>decimal128</td>
<td>40</td>
<td>24576</td>
</tr>
</tbody>
</table>

[10g] Types designated \_Float32x\_, \_Float64x\_, \_Float128x\_, \_Decima1\_64x\_, and \_Decima1\_128x 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: \_Decima1\_64x\_, which may have the same set of values as \_Decima1\_128\_, and may provide \_Decima1\_128x\_.

[10h] The standard floating types, interchange floating types, and extended floating types are collectively called the real floating types.

Replace 6.2.5#11:
[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.

For the standard real types float, double, and long double, the interchange floating types _FloatN, and the extended floating types _FloatNx, there are complex types designated respectively as float _Complex, double _Complex, long double _Complex, _FloatN _Complex, and _FloatNx _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:

_DECIMAL32
_DECIMAL64
_DECIMAL128

with:

_FLOATN, where N is 16, 32, 64, or ≥ 128 and a multiple of 32
_FLOAT32x
_FLOAT64x
_FLOAT128x

_DECIMALN, where N ≥ 32 and a multiple of 32
_DECIMAL64x
_DECIMAL128x

In the list of type specifiers in 6.7.2, replace:

_DECIMAL32
_DECIMAL64
_DECIMAL128

with:

_FLOATN, where N is 16, 32, 64, or ≥ 128 and a multiple of 32
_FLOAT32x
_FLOAT64x
_FLOAT128x

_DECIMALN, where N ≥ 32 and a multiple of 32
_DECIMAL64x
_DECIMAL128x

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

— _DECIMAL32
— _DECIMAL64
— _DECIMAL128

with:

— _FLOATN, where N is 16, 32, 64, or ≥ 128 and a multiple of 32
— _Float32x
— _Float64x
— _Float128x
— _DecimalN, where $N \geq 32$ and a multiple of 32
— _Decimal16x
— _Decimal128x
— _FloatN _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 _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), _Decimal16x, 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 _FloatN, and the extended floating types _FloatNx, there are imaginary types designated respectively as float _Imaginary, double _Imaginary, long double _Imaginary, _FloatN _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 Part 2 of Technical Specification 18661 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. Wherever `FLT_RADIX` appears in a description of a function that has versions that operate on decimal floating types, it is noted that for the decimal floating-point versions the value used is implicitly 10, rather than `FLT_RADIX`.

− number of digits in the coefficient

| 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)

| DEC32_MAX | $9.999999E96DF$ |
| DEC64_MAX | $9.999999999999999E384DD$ |
| DEC128_MAX | $9.999999999999999999999999999999999E6144DL$ |

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

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

— minimum normalized positive decimal floating-point number

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

— minimum positive subnormal decimal floating-point number

| DEC32_TRUE_MIN | 0.000001E-95DF |
| DEC64_TRUE_MIN | 0.000000000000001E-383DD |
| DEC128_TRUE_MIN | 0.000000000000000000001E-6143DL |

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 (including decimal floating types) and extended floating types in terms of the model presented in 5.2.4.2.2. The prefix FLT<sub>N</sub> indicates a binary interchange floating type of width <sub>N</sub>. The prefix FLT<sub>NX</sub> indicates a binary extended floating type that extends a basic format of width <sub>N</sub>. The prefix DEC<sub>N</sub> indicates a decimal floating type of width <sub>N</sub>. The prefix DEC<sub>NX</sub> indicates a decimal extended floating type that extends a basic format of width <sub>N</sub>. The type parameters <sub>p</sub>, <sub>emax</sub>, and <sub>emin</sub> 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.
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:

-1 indeterminable;

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

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

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;

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.

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\n
\texttt{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 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,

\[
\left[1 + p \log_{10} 2\right]
\]

\texttt{FLT\_DECIMAL\_DIG}

\texttt{FLT\_X\_DECIMAL\_DIG}

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

\texttt{FLT\_MANT\_DIG}

\texttt{FLT\_X\_MANT\_DIG}

- number of digits in the coefficient, $p$

\texttt{DEC\_MANT\_DIG}

\texttt{DEC\_X\_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,

\[
\left[(p - 1) \log_{10} 2\right]
\]

\texttt{FLT\_DECIMAL\_DIG}

\texttt{FLT\_X\_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,

\[
\left[(p - 1) \log_{10} 2\right]
\]

\texttt{FLT\_DIG}

\texttt{FLT\_X\_DIG}

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

\texttt{FLT\_MIN\_EXP}

\texttt{FLT\_X\_MIN\_EXP}

\texttt{DEC\_MIN\_EXP}

\texttt{DEC\_X\_MIN\_EXP}

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

\texttt{FLT\_MIN\_10\_EXP}

\texttt{FLT\_X\_MIN\_10\_EXP}

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

\texttt{FLT\_MAX\_EXP}

\texttt{FLT\_X\_MAX\_EXP}
DEC\_MAX\_EXP
DEC\_MAX\_EXP

— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, \[ \log_{10}(1 - 2^{-p}2^{emax}) \]

FLT\_MAX\_10\_EXP
FLT\_MAX\_10\_EXP

— maximum representable finite floating-point number, \((1 - b^{-p})b^{emax}\)

FLT\_MAX
FLT\_MAX
DEC\_MAX
DEC\_MAX

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

FLT\_EPSILON
FLT\_EPSILON
DEC\_EPSILON
DEC\_EPSILON

— minimum normalized positive floating-point number, \(b^{emin-1}\)

FLT\_MIN
FLT\_MIN
DEC\_MIN
DEC\_MIN

— minimum positive subnormal floating-point number, \(b^{emin-p}\)

FLT\_TRUE\_MIN
FLT\_TRUE\_MIN
DEC\_TRUE\_MIN
DEC\_TRUE\_MIN

With the following change, \texttt{DECIMAL\_DIG} characterizes conversions of supported IEC 60559 encodings, which may be wider than supported floating types.

35 \textbf{Change to C11 + TS18661-1 + TS18661-2:}

In 5.2.4.2.2#11, change the bullet defining \texttt{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 \textbf{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.

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

In 6.3.1.8#1, replace the following items after “This pattern is called the *usual arithmetic conversions*”:

5  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.

10 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`.

15   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`.

20 with:

   If one operand has decimal floating type, the other operand shall not have standard floating, complex, or imaginary type, nor shall it have a floating type of radix 2.

   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.

25   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.

30   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.

35   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:

```
f l F L df dd dL DF DD DL
```

10 to:

```
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`.

15 with:

[2a] A `floating-suffix` `df`, `dd`, `dl`, `DF`, `DD`, `dN`, `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 `strtd32`, `strtd64`, or `strtd128` function for the same numeric string.

with:

The quantum exponent is specified to be the same as for the corresponding `strtdN` or `strtdNx` function for the same numeric string.

10 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 `<stdlib.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.

11 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.

...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`, `dN`, and `dNx` suffixes, which are corresponding functions whose parameters, return values, or both are of types `float`, `long double`, `_FloatN`, `_FloatNx`, `_DecimalN`, and `_DecimalNx`, respectively.

Add after 7.12#1d:

1. 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.

11.1 Macros

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

Replace 7.12#3a:

3. 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
SNANFX
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

FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128

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

with:

[7b] The macros

FP_FAST_FMAFN
FP_FAST_FMADDN
FP_FAST_FMAFNX
FP_FAST_FMADNX

are, respectively, _FloatN, _DecimalN, _FloatN, and _DecimalN analogues of
FP_FAST_FMA.

Replace 7.12#7c:

[7c] The macros

FP_FAST_D32ADDD64
FP_FAST_D32ADDD128
FP_FAST_D64ADDD128
FP_FAST_D32SUBD64
FP_FAST_D32SUBD128
FP_FAST_D64SUBD128
FP_FAST_D32MULD64
FP_FAST_D32MULD128
FP_FAST_D64MULD128
FP_FAST_D32DIVD64
FP_FAST_D32DIVD128
FP_FAST_D64DIVD128
FP_FAST_D32FMADD64
FP_FAST_D32FMADD128
FP_FAST_D64FMADD128
FP_FAST_D32SQRTD64
FP_FAST_D32SQRTD128
FP_FAST_D64SQRTD128

are decimal analogues of FP_FAST_FADD, FP_FAST_FADDL, FP_FAST_DADDL, etc.

with:

[7c] The macros in the following lists are interchange and extended floating type analogues of
FP_FAST_FADD, FP_FAST_FADDL, FP_FAST_DADDL, etc.

[7d] For M < N, the macros

FP_FAST_FMADDFN
FP_FAST_FMSUBFN
FP_FAST_FMMULFN
FP_FAST_FMDIVFN
FP_FAST_FMFMAFN
FP_FAST_FMSQRTFN
FP_FAST_DMAADDN
FP_FAST_DMSUBDN
FP_FAST_DMULDN
FP_FAST_DDIVDN
FP_FAST_DMFMAVN
FP_FAST_DMSQRTDN

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

```
FP_FAST_FMADDFNX
FP_FAST_FMSUBFNX
FP_FAST_FMMULFNX
FP_FAST_FMDIVFNX
FP_FAST_FMSQRTFNX
FP_FAST_DMADDDNX
FP_FAST_DMSUBDNX
FP_FAST_DMULDNX
FP_FAST_DMDVIDNX
FP_FAST_DFMADNX
FP_FAST_DMSQRTDXN
```

classify 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

```
FP_FAST_FMADDNX
FP_FAST_FMSUBNX
FP_FAST_FMMULNX
FP_FAST_FMDIVNX
FP_FAST_FMSQRTNX
FP_FAST_DMADDNX
FP_FAST_DMSUBDNX
FP_FAST_DMULDNX
FP_FAST_DMDVNDX
FP_FAST_DFMADNX
FP_FAST_DMSQRTDNNX
```

classify 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

```
FP_FAST_FMADDNX
FP_FAST_FMSUBNX
FP_FAST_FMMULNX
FP_FAST_FMDIVNX
FP_FAST_FMSQRTNX
FP_FAST_DMADDNX
FP_FAST_DMSUBDNX
FP_FAST_DMULDNX
FP_FAST_DMDVNDX
FP_FAST_DFMADNX
FP_FAST_DMSQRTDNNX
```

classify 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 $M$.

11.2 Function prototypes

Change to C11 + TS18661-1 + TS18661-2:
Add the following list of function prototypes to the synopsis of the respective subclauses:

7.12.4 Trigonometric functions

_\_Float64N acosfN(_\_Float64N x);  
_\_Float64N acosfN(_\_Float64N x); 
_\_Decimal64N acosdN(_\_Decimal64N x); 
_\_Decimal64N acosdN(_\_Decimal64N x); 

_\_Float64N asinfN(_\_Float64N x);  
_\_Float64N asinfN(_\_Float64N x); 
_\_Decimal64N asindN(_\_Decimal64N x); 
_\_Decimal64N asindN(_\_Decimal64N x); 

_\_Float64N atanfN(_\_Float64N x);  
_\_Float64N atanfN(_\_Float64N x); 
_\_Decimal64N atandN(_\_Decimal64N x); 
_\_Decimal64N atandN(_\_Decimal64N x); 

_\_Float64N atan2fN(_\_Float64N y, _\_Float64N x); 
_\_Float64N atan2fN(_\_Float64N y, _\_Float64N x); 
_\_Decimal64N atan2dN(_\_Decimal64N y, _\_Decimal64N x); 
_\_Decimal64N atan2dN(_\_Decimal64N y, _\_Decimal64N x); 

_\_Float64N cosfN(_\_Float64N x);  
_\_Float64N cosfN(_\_Float64N x); 
_\_Decimal64N cosdN(_\_Decimal64N x); 
_\_Decimal64N cosdN(_\_Decimal64N x); 

_\_Float64N sinfN(_\_Float64N x);  
_\_Float64N sinfN(_\_Float64N x); 
_\_Decimal64N sindN(_\_Decimal64N x); 
_\_Decimal64N sindN(_\_Decimal64N x); 

_\_Float64N tanfN(_\_Float64N x);  
_\_Float64N tanfN(_\_Float64N x); 
_\_Decimal64N tandN(_\_Decimal64N x); 
_\_Decimal64N tandN(_\_Decimal64N x); 

7.12.5 Hyperbolic functions

_\_Float64N acoshfN(_\_Float64N x);  
_\_Float64N acoshfN(_\_Float64N x); 
_\_Decimal64N acoshdN(_\_Decimal64N x); 
_\_Decimal64N acoshdN(_\_Decimal64N x); 

_\_Float64N asinhfN(_\_Float64N x);  
_\_Float64N asinhfN(_\_Float64N x); 
_\_Decimal64N asinhdN(_\_Decimal64N x); 
_\_Decimal64N asinhdN(_\_Decimal64N x); 

_\_Float64N atanhfN(_\_Float64N x);  
_\_Float64N atanhfN(_\_Float64N x); 
_\_Decimal64N atanhdN(_\_Decimal64N x); 
_\_Decimal64N atanhdN(_\_Decimal64N x);
7.12.6 Exponential and logarithmic functions

_FloatN expfN(_FloatN x);
_FloatNx expfNx(_Float Nx x);
.DecimalN expdN(_DecimalN x);
.DecimalNx expdNx(_DecimalNx x);

_FloatN exp2fN(_FloatN x);
_FloatNx exp2fNx(_Float Nx x);
.DecimalN exp2dN(_DecimalN x);
.DecimalNx exp2dNx(_DecimalNx x);

_FloatN expmlfN(_FloatN x);
_FloatNx expmlfNx(_Float Nx x);
.DecimalN expmldN(_DecimalN x);
.DecimalNx expmldNx(_DecimalNx x);

_FloatN frexpfN(_FloatN value, int *exp);
_FloatNx frexpfNx(_FloatN value, int *exp);
.DecimalN frexpN(_DecimalN value, int *exp);
.DecimalNx frexpNx(_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(_Float Nx value, int exp);
.DecimalN ldexpdN(_DecimalN value, int exp);
.DecimalNx ldexpdNx(_DecimalNx value, int exp);

long int lllogbfN(_FloatN x);
long int lllogbfNx(_FloatNx x);
long int lllogbdN(_DecimalN x);
long int lllogbdNx(_DecimalNx x);

_FloatN logfN(_FloatN x);
_FloatNx logfNx(_FloatNx x);
.DecimalN logdN(_DecimalN x);
7.12.7 Power and absolute-value functions

```c
_FloatN log10fN(_FloatN x);
_FloatN log10dN(_FloatN x);
_DecimalN log10dN(_DecimalN x);

_FloatN log10fX(_FloatN x);
_FloatN log10fX(_FloatN x);
_DecimalN log10dN(_DecimalN x);

_FloatN log1pfN(_FloatN x);
_FloatN log1pfN(_FloatN x);
_DecimalN log1pdN(_DecimalN x);

_FloatN log1pfX(_FloatN x);
_FloatN log1pfX(_FloatN x);
_DecimalN log1pdN(_DecimalN x);

_FloatN log2fN(_FloatN x);
_FloatN log2fX(_FloatN x);
_DecimalN log2dN(_DecimalN x);

_FloatN log2fX(_FloatN x);
_FloatN log2dN(_DecimalN x);
_DecimalN log2dN(_DecimalN x);

_FloatN logbfN(_FloatN x);
_FloatN logbfX(_FloatN x);
_DecimalN logbdN(_DecimalN x);

_FloatN logbdN(_DecimalN x);

_FloatN logbfX(_FloatN x);
_FloatN logbfX(_FloatN x);
_DecimalN logbdN(_DecimalN x);

_FloatN logbdN(_DecimalN x);

_FloatN modffN(_FloatN x, _FloatN *iptr);
_FloatN modffN(_FloatN x, _FloatN *iptr);
_DecimalN modfdN(_DecimalN x, _DecimalN *iptr);

_DecimalN modfdN(_DecimalN x, _DecimalN *iptr);

_FloatN scalbnfN(_FloatN value, int exp);
_FloatN scalbnfN(_FloatN value, int exp);
_DecimalN scalbndN(_DecimalN value, int exp);

_DecimalN scalbndN(_DecimalN value, int exp);

_FloatN scalbnfN(_FloatN value, long int exp);
_FloatN scalbnfN(_FloatN value, long int exp);
_DecimalN scalbndN(_DecimalN value, long int exp);

_DecimalN scalbndN(_DecimalN value, long int exp);

_FloatN cbrtfN(_FloatN x);
_FloatN cbrtfN(_FloatN x);
_DecimalN cbrtdN(_DecimalN x);

_DecimalN cbrtdN(_DecimalN x);

_FloatN cbrtfX(_FloatN x);
_FloatN cbrtfX(_FloatN x);
_DecimalN cbrtdX(_DecimalN x);

_DecimalN cbrtdX(_DecimalN x);

_FloatN fabsfN(_FloatN x);
_FloatN fabsfN(_FloatN x);
_DecimalN fabsdN(_DecimalN x);

_DecimalN fabsdN(_DecimalN x);

_FloatN hypotfN(_FloatN x, _FloatN y);
_FloatN hypotfN(_FloatN x, _FloatN y);
_DecimalN hypotdN(_DecimalN x, _DecimalN y);

_DecimalN hypotdN(_DecimalN x, _DecimalN y);

_FloatN powfN(_FloatN x, _FloatN y);
_FloatN powfN(_FloatN x, _FloatN y);
```
7.12.8 Error and gamma functions

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

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

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

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

7.12.9 Nearest integer functions

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

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

_FloatN nearbyintfN(_FloatN x);
_FloatN nearbyintfNx(_FloatN x);
_DecimalN nearbyintdN(_DecimalN x);
_DecimalN nearbyintdNx(_DecimalN x);

_FloatN rintfN(_FloatN x);
_FloatN rintfNx(_FloatN x);
_DecimalN rintdN(_DecimalN x);
_DecimalN rintdNx(_DecimalN x);

long int lrintfN(_FloatN x);
long int lrintfNx(_FloatN x);
long int lrintdN(_DecimalN x);
long int lrintdNx(_DecimalN x);
long long int llrintfN(_FloatN x);
long long int llrintfNx(_FloatN x);
long long int llrintdN(_DecimalN x);
long long int llrintdNx(_DecimalN x);

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

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

long long int llroundfN(_FloatN x);
long long int llroundfNx(_FloatN x);
long long int llrounddN(_DecimalN x);
long long int llrounddNx(_DecimalN x);

FLOATN roundevenfN(_FloatN x);
_FLOATN roundevenfNx(_FloatNx x);
_DECIMALN roundevendN(_DecimalN x);
_DECIMALN roundevendNx(_DecimalNx x);

_FLOATN truncfN(_FloatN x);
_FLOATN truncfNx(_FloatNx x);
_DECIMALN truncdN(_DecimalN x);
_DECIMALN truncdNx(_DecimalNx x);

intmax_t fromfpfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpfNx(_FloatNx x, int round, unsigned int width);
intmax_t fromfpdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpdNx(_DecimalNx x, int round, unsigned int width);

uintmax_t ufompfpfN(_FloatN x, int round, unsigned int width);
uintmax_t ufompfpfNx(_FloatNx x, int round, unsigned int width);
uintmax_t ufompfpdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufompfpdNx(_DecimalNx x, int round, unsigned int width);

intmax_t fromfpfxfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpfxfNx(_FloatNx x, int round, unsigned int width);
intmax_t fromfpfxdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpfxdNx(_DecimalNx x, int round, unsigned int width);

uintmax_t ufompfpfxfN(_FloatN x, int round, unsigned int width);
uintmax_t ufompfpfxfNx(_FloatNx x, int round, unsigned int width);
uintmax_t ufompfpfxdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufompfpfxdNx(_DecimalNx x, int round, unsigned int width);

7.12.10 Remainder functions

_FLOATN fmodfN(_FloatN x, _FloatN y);
_FLOATN fmodfNx(_FloatNx x, _FloatNx y);
_DECIMALN fmoddN(_DecimalN x, _DecimalN y);
_DECIMALN fmoddNx(_DecimalNx x, _DecimalNx y);
_FLOATN remainderfN(_FloatN x, _FloatN y);
7.12.11 Manipulation functions

_FloatN remainderfNx(_FloatN x, _FloatN y);
_FloatN remainderdN(_DecimalN x, _DecimalN y);
_FloatN remainderdNx(_DecimalN x, _DecimalN y);
_FloatN remquoN(_FloatN x, _FloatN y, int *quo);
_FloatN remquoNx(_FloatN x, _FloatN y, int *quo);

_FloatN copysignN(_FloatN x, _FloatN y);
_FloatN copysignN(x, _FloatN y);
_FloatN copysigndN(_DecimalN x, _DecimalN y);
_FloatN copysigndN(x, _DecimalN y);
_FloatN nanfN(const char *tagp);
_FloatN nanfNn(const char *tagp);
_FloatN nandN(const char *tagp);
_FloatN nandNn(const char *tagp);
_FloatN nextafterN(_FloatN x, _FloatN y);
_FloatN nextafterN(x, _FloatN y);
_FloatN nextafterdN(_DecimalN x, _DecimalN y);
_FloatN nextafterdNn(_DecimalN x, _DecimalN y);
_FloatN nextupfN(_FloatN x);
_FloatN nextupN(_FloatN x);
_FloatN nextupfNn(_FloatN x);
_FloatN nextupNn(_FloatN x);
_FloatN nextdownfN(_FloatN x);
_FloatN nextdownfNn(_FloatN x);
_FloatN nextdownN(_DecimalN x);
_FloatN nextdownNn(_DecimalN x);
_FloatN nextdowndN(_DecimalN x);
_FloatN nextdowndNn(_DecimalN x);

int canonicalizeN(_FloatN * cx, const _FloatN * x);
int canonicalizeNn(_FloatN * cx, const _FloatN * x);
int canonicalizedN(_DecimalN * cx, const _DecimalN * x);
int canonicalizedNn(_DecimalN * cx, const _DecimalN * x);

_DeclimalN quantizedN(_DecimalN x, _DecimalN y);
_DeclimalN quantizedNn(_DecimalN x, _DecimalN y);
_Bool samequantumN(_DecimalN x, _DecimalN y);
_Bool samequantumNn(_DecimalN x, _DecimalN y);

_DeclimalN quantumN(_DecimalN x);
_DeclimalN quantumNn(_DecimalN x);
long long int llquantexpdN(_DecimalN x);
long long int llquantexpdNn(_DecimalN x);
7.12.12 Maximum, minimum, and positive difference functions

```c
void decodebind(unsigned char * restrict encptr, const _Decimal N * restrict xptr);
void decodedecdN(_DecimalN * restrict xptr, const unsigned char * restrict encptr);
void encodebind(unsigned char * restrict encptr, const _DecimalN * restrict xptr);
void decodebindN(_DecimalN * restrict xptr, const unsigned char * restrict encptr);
```

7.12.13 Floating multiply

```c
_Fl0atN fdimfN(_Fl0atN x, _Fl0atN y);
_Fl0atNx fdimfxN(_FloatNx x, _FloatNx y);
Deci0alN fdimdN(_DecimalN x, _DecimalN y);
_DecimalN fdimdNx(_DecimalNx x, _DecimalNx y);
```

```c
_Fl0atN fmxfN(_Fl0atN x, _Fl0atN y);
_Fl0atNx fmxfNxN(_FloatNx x, _FloatNx y);
Deci0alN fmxdN(_DecimalN x, _DecimalN y);
_DecimalN fmxdNx(_DecimalNx x, _DecimalNx y);
```

```c
_Fl0atN fminfN(_Fl0atN x, _Fl0atN y);
_Fl0atNx fminfxN(_FloatNx x, _FloatNx y);
Deci0alN fmindN(_DecimalN x, _DecimalN y);
_DecimalN fmindNx(_DecimalNx x, _DecimalNx y);
```

```c
_Fl0atN fmaxmagnF(_Fl0atN x, _Fl0atN y);
_Fl0atNx fmaxmagnFxN(_FloatNx x, _FloatNx y);
Deci0alN fmaxmagnD(_DecimalN x, _DecimalN y);
_DecimalN fmaxmagnDx(_DecimalNx x, _DecimalNx y);
```

7.12.14 Functions that round result to narrower format

```c
_Fl0atM fMaddfN(_Fl0atN x, _Fl0atN y); // M < N
_Fl0atM fMaddfNx(_FloatNx x, _FloatNx y); // M <= N
_FloatM fMaddfN(_FloatN x, _FloatN y); // M < N
_FloatM fMaddfN(_FloatM x, _FloatM y); // M <= N
_DecimalM dMaddN(_DecimalN x, _DecimalN y); // M < N
_DecimalM dMaddNx(_DecimalNx x, _DecimalNx y); // M <= N
_DecimalM dMaddN(_DecimalN x, _DecimalN y); // M < N
_DecimalM dMaddNx(_DecimalNx x, _DecimalNx y); // M <= N
_FloatM fMsufN(_FloatN x, _FloatN y); // M < N
_FloatM fMsufN(_FloatM x, _FloatM y); // M <= N
_FloatM fMsufN(_FloatN x, _FloatN y); // M < N
```

© ISO/IEC 2013 – All rights reserved
```c
_FloatM fmSubfN(_FloatN x, _FloatN y); // M < N
_DECIMALM dMSubD(_DECIMALN x, _DECIMALN y); // M < N
_DECIMALM dMSubDx(_DECIMALN x, _DECIMALN y); // M <= N
_DECIMALM dMSubDxN(_DECIMALN x, _DECIMALN y); // M < N

_FloatM fmMulfN(_FloatN x, _FloatN y); // M < N
_FloatM fmMulfN(_FloatN x, _FloatN y); // M <= N
_FloatM fmMulfN(_FloatN x, _FloatN y); // M < N
_DECIMALM dMMulD(_DECIMALN x, _DECIMALN y); // M < N
_DECIMALM dMMulDx(_DECIMALN x, _DECIMALN y); // M <= N
_DECIMALM dMMulDxN(_DECIMALN x, _DECIMALN y); // M < N

_FloatM fMdivfN(_FloatN x, _FloatN y); // M < N
_FloatM fMdivfN(_FloatN x, _FloatN y); // M <= N
_FloatM fMdivfN(_FloatN x, _FloatN y); // M < N
_DECIMALM dMDivD(_DECIMALN x, _DECIMALN y); // M < N
_DECIMALM dMDivDx(_DECIMALN x, _DECIMALN y); // M <= N
_DECIMALM dMDivDxN(_DECIMALN x, _DECIMALN y); // M < N

_FloatM fMsqrtfN(_FloatN x); // M < N
_FloatM fMsqrtfN(_FloatN x); // M <= N
_FloatM fMsqrtfN(_FloatN x); // M < N
_DECIMALM dMsqrtD(_DECIMALN x); // M <= N
_DECIMALM dMsqrtDx(_DECIMALN x); // M < N
_DECIMALM dMsqrtDxN(_DECIMALN x); // M < N

_FloatM fMfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMfmafN(_FloatN x, _FloatN y, _FloatN z); // M <= N
_FloatM fMfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_DECIMALM dMfmadN(_DECIMALN x, _DECIMALN y, _DECIMALN z); // M < N
_DECIMALM dMfmadN(_DECIMALN x, _DECIMALN y, _DECIMALN z); // M < N
_DECIMALM dMfmadN(_DECIMALN x, _DECIMALN y, _DECIMALN z); // M < N
```

F.10.12 Total order functions

```c
int totalorderfN(_FloatN x, _FloatN y);
int totalorderfNx(_FloatN x, _FloatN y);
int totalorderfN(_DecimalN x, _DecimalN y);
int totalorderfNx(_DecimalN x, _DecimalN y);
int totalordermagnfN(_FloatN x, _FloatN y);
int totalordermagnfN(_FloatN x, _FloatN y);
int totalordermagn(_DecimalN x, _DecimalN y);
int totalordermagn(_DecimalN x, _DecimalN y);
```
F.10.13 Payload functions

```c
_FloatN getpayloadfN(const _FloatN *x);
_FloatN getpayloadfNx(const _FloatNx *x);
_DecimalN getpayloadN(const _DecimalN *x);
_DecimalN getpayloadNx(const _DecimalNx *x);

int setpayloadfN(_FloatN *res, _FloatN pl);
int setpayloadfNx(_FloatNx *res, _FloatNx pl);
int setpayloadN(_DecimalN *res, _DecimalN pl);
int setpayloadNx(_DecimalNx *res, _DecimalNx pl);
int setpayloadfNsigfN(_FloatN *res, _FloatN pl);
int setpayloadfNsigfNx(_FloatNx *res, _FloatNx pl);
int setpayloadfNsigdN(_DecimalN *res, _DecimalN pl);
int setpayloadfNsigdNx(_DecimalNx *res, _DecimalNx pl);
```

11.3 Encoding conversion functions

The functions in this subclause, together with the numerical conversion functions for encodings in clause 12, support the non-arithmetic interchange formats specified by IEC 60559.

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

After 7.12.11.7, add:

7.12.11.7a The `encodefN` functions

**Synopsis**

```c
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void encodefN(unsigned char * restrict encptr, const _FloatN * restrict xptr);
```

**Description**

[2] The `encodefN` functions convert `xptr` into an IEC 60559 binary 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**

```c
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#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 binary\(N\) 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 the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall provide both `dMencdecdN` and `dMencbindN` functions for each \(M\) and \(N\) equal the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.

7.12.11.7c.1 The `fMencfN` functions

Synopsis

[1] 
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void fMencfN(unsigned char * restrict encMptr, const unsigned char * restrict encNptr);
```

Description

[2] These functions convert between IEC 60559 binary interchange formats. These functions interpret the \(N/8\) element array pointed to by `encNptr` 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 `encMptr`. 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 `dMencdecdN` and `dMencbindN` functions

Synopsis

[1] 
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void dMencdecdN(unsigned char * restrict encMptr, const unsigned char * restrict encNptr);
void dMencbindN(unsigned char * restrict encMptr, const unsigned char * restrict encNptr);
```

Description

[2] These functions convert between IEC 60559 decimal interchange formats that use the same encoding scheme. The `dMencdecdN` functions convert between formats using the encoding scheme based on decimal encoding of the significand. The `dMencbindN` 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 \texttt{encNptr} 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 \texttt{encMptr}. 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.

#### 12 Numeric conversion functions in <\texttt{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 \texttt{strtod} function in <\texttt{stdlib.h}>. Conversions to character sequences are provided by functions analogous to the \texttt{strfromd} function in <\texttt{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

[3a] For each interchange or extended floating type that the implementation provides, \texttt{<stdlib.h>} shall declare the associated functions. Conversely, for each such type that the implementation does not provide, \texttt{<stdlib.h>} shall not declare the associated functions unless specified otherwise.

After 7.22.1.2b, insert:

7.22.1.2c The \texttt{strfromfN}, \texttt{strfromfNx}, \texttt{strfromdN}, and \texttt{strfromdNx} functions

### Synopsis

[1] \texttt{#define __STDC_WANT_IEC_60559_TYPES_EXT__}

\begin{verbatim}
#include <stdlib.h>

int strfromfN(char * restrict s, size_t n, const char * restrict format, _FloatN fp);
int strfromfNx(char * restrict s, size_t n, const char * restrict format, _FloatNx fp);
int strfromdN(char * restrict s, size_t n, const char * restrict format, _DecimalN fp);
int strfromdNx(char * restrict s, size_t n, const char * restrict format, _DecimalNx fp);
\end{verbatim}

### Description

[2] The \texttt{strfromfN} and \texttt{strfromfNx} functions are similar to the \texttt{strfromd} function, except they convert to the types \_FloatN and \_FloatNx, respectively. The \texttt{strfromdN} and \texttt{strfromdNx} functions are similar to the \texttt{strfromd64} function, except they convert from the types \_DecimalN and \_DecimalNx, respectively.

### Returns

[3] The \texttt{strfromfN} and \texttt{strfromfNx} functions return values similar to the \texttt{strfromd} function. The \texttt{strfromdN} and \texttt{strfromdNx} functions return values similar to the \texttt{strfromd64} function.

After 7.22.1.3a, insert:
7.22.1.3b The `strtofN`, `strtofNx`, `strtodN`, and `strtodNx` functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
_FloatN strtofN(const char * restrict nptr, char ** restrict endptr);
_FloatNx strtofNx(const char * restrict nptr, char ** restrict endptr);
_DecimalN strtodN(const char * restrict nptr, char ** restrict endptr);
_DecimalNx strtodNx(const char * restrict nptr, char ** restrict endptr);

Description

[2] The `strtofN` and `strtofNx` functions are similar to the `strtod` function, except they convert to the types `_FloatN` and `_FloatNx`, respectively. The `strtodN` and `strtodNx` functions are similar to the `strtod64` function, except they convert to the types `_DecimalN` and `_DecimalNx`, respectively.

Returns

[3] The `strtofN` and `strtofNx` functions return values similar to the `strtod` function, except in the types `_FloatN` and `_FloatNx`, respectively. The `strtodN` and `strtodNx` functions return values similar to the `strtod64` function, except in the types `_DecimalN` and `_DecimalNx`, respectively.

7.22.1.3c String-to-encoding functions

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

7.22.1.3c.1 The `strtoencfN` functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#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.
7.22.1.3c.2 The strtoencdecN and strtoencbindN functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#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.

7.22.1.3d String-from-encoding functions

20 [1] An implementation shall declare the strfromencfN function for each N equal the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall declare both the strfromencdecN and strfromencbindN functions for each N equal the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.

7.22.1.3d.1 The strfromencfN functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#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 binaryN encoding. The order of bytes in the arrays is implementation defined.

Returns

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

7.22.1.3d.2 The strfromencdecN and strfromencbindN functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
int strfromencdecN(char * restrict s, size_t n, const char * restrict format, const unsigned char * restrict encptr);
int strfromencbindNx(char * restrict s, size_t n, const char * restrict format, const unsigned char * restrict encptr);
format, const unsigned char * restrict encptr);

Description

[2] The `strfromencdecN` functions are similar to the `strfromdN` functions except the input is the value of the N/8 element array pointed to by `encptr`, interpreted as an IEC 60559 decimal encoding in the coding scheme based on decimal encoding of the significand. The `strfromencbindN` functions are similar to the `strfromdN` functions except the input is the value of the N/8 element array pointed to by `encptr`, interpreted as an IEC 60559 decimal 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 `strfromencdecN` and `strfromencbindN` functions return the same values as corresponding `strfromdN` functions.

13 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`, `_FloatN`, and `_FloatNx`.

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

```c
_FloatN complex cacosfN(_FloatN complex z);
_FloatN complex cacosfNx(_FloatNx complex z);
_FloatN complex casinfN(_FloatN complex z);
_FloatN complex casinfNx(_FloatNx complex z);
```
7.3.6 Hyperbolic functions

- \_FloatN\ complex\ \texttt{cacosf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{cacosfN}(\_FloatN\ complex\ \texttt{z});

- \_FloatN\ complex\ \texttt{ccoshf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{ccoshfN}(\_FloatN\ complex\ \texttt{z});

- \_FloatN\ complex\ \texttt{casinhf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{casinhfN}(\_FloatN\ complex\ \texttt{z});

- \_FloatN\ complex\ \texttt{catanhf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{catanhfN}(\_FloatN\ complex\ \texttt{z});

7.3.7 Exponential and logarithmic functions

- \_FloatN\ complex\ \texttt{cexpf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{cexpfN}(\_FloatN\ complex\ \texttt{z});

- \_FloatN\ complex\ \texttt{clogf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{clogfN}(\_FloatN\ complex\ \texttt{z});

7.3.8 Power and absolute value functions

- \_FloatN\ complex\ \texttt{cabsf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{cabsfN}(\_FloatN\ complex\ \texttt{z});

- \_FloatN\ complex\ \texttt{cpowf}(\_FloatN\ complex\ \texttt{z}, \_FloatN\ complex\ \texttt{y});
- \_FloatN\ complex\ \texttt{cpowfN}(\_FloatN\ complex\ \texttt{z}, \_FloatN\ complex\ \texttt{y});

- \_FloatN\ complex\ \texttt{csqrtf}(\_FloatN\ complex\ \texttt{z});
- \_FloatN\ complex\ \texttt{csqrtfN}(\_FloatN\ complex\ \texttt{z});

7.3.9 Manipulation functions

- \_FloatN\ complex\ \texttt{cargf}(\_FloatN\ complex\ \texttt{z});
ISO/IEC TS 18661-3  Working Group Draft – December 17, 2013  WG 14 N1784

14 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:

In 7.25, replace paragraphs [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, 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 \( M \geq 64 \) or _DecimalNx where \( N \geq 32 \), 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 \( M \geq 128 \), or _FloatNx where \( N \geq 64 \), 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) of these arguments, the type determined is long double.

— Otherwise, if 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 __STDC_WANT_IEC_60559_TYPES_EXT__

#if defined(__STDC_WANT_IEC_60559_TYPES_EXT__) 
#define cbrt(X) _Generic((X), 
                    _Float128: cbrtf128(X), 
                    _Float64: cbrtf64(X), 
                    _Float32: cbrtf32(X), 
                    _Float64x: cbrtf64x(X), 
                    _Float32x: cbrtf32x(X), 
                    long double: cbttl(X), 
                    default: _Roundwise_cbrt(X), 
                    float: cbrtf(X) 
                   )
#else  
#define cbrt(X) _Generic((X), 
                    long double: cbttl(X), 
                    default: _Roundwise_cbrt(X), 
                    float: cbrtf(X) 
                   )
#endif
```

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

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

```c
#define __STDC_WANT_IEC_60559_TYPES_EXT__
```

In 7.25#7, append to the declarations:

```c
#if __STDC_IEC_60559_TYPES__ >= 201ymmL  
_Float32x f32x;  
_Float64 f64;  
_Float128 f128;  
_Float64 complex f64c;
#endif
```

In 7.25#7, append to the table:
\begin{verbatim}
\cos(f64x)  \quad \text{ccosf64x}(f64x)
\pow(dc, f128)  \quad \text{cpowf128}(dc, f128)
\fmax(f64, d)  \quad \text{fmaxf64}(f64, d)
\fmax(d, f32x)  \quad \text{fmax}(d, f32x), \text{the function, if the set of values of } _\text{Float32x} \text{ is a subset of (or equivalent to) the set of values of } \text{double}, \text{or}
\fmaxf32x(d, f32x), \text{if the set of values of } \text{double} \text{ is a proper subset of the set of values of } _\text{Float32x}, \text{or}
\text{undefined}, \text{if neither of the sets of values of } \text{double} \text{ and } _\text{Float32x} \text{ is a subset of the other (and the sets are not equivalent)}
\end{verbatim}
Bibliography


