ISO/IEC JTC 1/SC 22/WG 14 N1758

Date: yyyy-mm-dd

Reference number of document: ISO/IEC TS 18661

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

Secretariat: ANSI

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

Warning

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

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

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

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

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

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

Violators may be prosecuted.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>v</td>
</tr>
<tr>
<td>Background</td>
<td>v</td>
</tr>
<tr>
<td>IEC 60559 floating-point standard</td>
<td>v</td>
</tr>
<tr>
<td>C support for IEC 60559</td>
<td>vi</td>
</tr>
<tr>
<td>Purpose</td>
<td>vii</td>
</tr>
<tr>
<td>Additional background on formats</td>
<td>vii</td>
</tr>
<tr>
<td>1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>2 Conformance</td>
<td>1</td>
</tr>
<tr>
<td>3 Normative references</td>
<td>1</td>
</tr>
<tr>
<td>4 Terms and definitions</td>
<td>1</td>
</tr>
<tr>
<td>5 C standard conformance</td>
<td>2</td>
</tr>
<tr>
<td>5.1 Freestanding implementations</td>
<td>2</td>
</tr>
<tr>
<td>5.2 Predefined macros</td>
<td>2</td>
</tr>
<tr>
<td>5.3 Standard headers</td>
<td>2</td>
</tr>
<tr>
<td>6 Types</td>
<td>7</td>
</tr>
<tr>
<td>7 Characteristics</td>
<td>12</td>
</tr>
<tr>
<td>8 Conversions</td>
<td>16</td>
</tr>
<tr>
<td>9 Constants</td>
<td>17</td>
</tr>
<tr>
<td>10 Non-arithmetic interchange formats</td>
<td>18</td>
</tr>
<tr>
<td>11 Mathematics <code>&lt;math.h&gt;</code></td>
<td>18</td>
</tr>
<tr>
<td>11.1 Macros</td>
<td>19</td>
</tr>
<tr>
<td>11.2 Function prototypes</td>
<td>22</td>
</tr>
<tr>
<td>11.3 Encoding conversion functions</td>
<td>30</td>
</tr>
<tr>
<td>12 Numeric conversion functions in <code>&lt;stdlib.h&gt;</code></td>
<td>32</td>
</tr>
<tr>
<td>13 Complex arithmetic <code>&lt;complex.h&gt;</code></td>
<td>35</td>
</tr>
<tr>
<td>14 Type-generic macros <code>&lt;tgmath.h&gt;</code></td>
<td>37</td>
</tr>
<tr>
<td>Bibliography</td>
<td>41</td>
</tr>
</tbody>
</table>
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: Supplemental functions
— Part 5: Supplemental 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 ≥ 128 and a multiple of 32, and
  - for decimal, k ≥ 32 and a multiple of 32
- extended formats, for each basic format, with minimum range and precision specified

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

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

Extended formats are intended for intermediate computation, not input or output data. The extra precision often allows the computation of extended results which when converted to a narrower output format differ from the ideal results by little more than a unit in the last place. Also, the extra range often avoids any intermediate overflow or underflow that might occur if the computation were done in the format of the data. The essential 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 float and double formats must be IEC 60559 binary32 and binary64.

Part 2 of Technical Specification 18661 defines types _Decimal32, _Decimal64, and _Decimal128 with IEC 60559 formats decimal32, decimal64, and decimal128. Although IEC 60559 does not require arithmetic support (other than conversions) for its decimal32 interchange format, Part 2 of Technical Specification 18661 has full arithmetic and library support for _Decimal32, just like for _Decimal64 and _Decimal128.

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

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 double, _Float64 (the type for the binary64 interchange format), and maybe _Float32x (the type for the binary32-extended format). This redundancy is intended to support the different programming models appropriate for the types with arithmetic interchange formats and extended formats and C standard floating types.

This document also supports the IEC 60559 non-arithmetic interchange formats with functions that convert among encodings and between encodings and character sequences, for all interchange formats.
Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 3: Interchange and extended types

1 Scope

This 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.ymmL.

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:yyy, 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:yyy, 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:

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

to:

```c
__STDC_IEC_60559_DFP__ The integer constant 201ymmL, intended to indicate support of the decimal floating types _Decimal32, _Decimal64, and _Decimal128, with decimal floating-point arithmetic according to IEC 60559.
```

In 6.10.8.3#1, add:

```c
__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:

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLTN_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>FLTN_DECIMAL_DIG</td>
<td></td>
</tr>
<tr>
<td>FLTN_DIG</td>
<td></td>
</tr>
<tr>
<td>FLTN_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>FLTN_MIN</td>
<td></td>
</tr>
<tr>
<td>FLTN_MAX</td>
<td></td>
</tr>
<tr>
<td>FLTN_MIN_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLTN_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLTN_EPSILON</td>
<td></td>
</tr>
<tr>
<td>FLTN_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>FLTN_MAX</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLTNx_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>FLTNx_DECIMAL_DIG</td>
<td></td>
</tr>
<tr>
<td>FLTNx_DIG</td>
<td></td>
</tr>
<tr>
<td>FLTNx_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>FLTNx_MIN</td>
<td></td>
</tr>
<tr>
<td>FLTNx_MAX</td>
<td></td>
</tr>
<tr>
<td>FLTNx_MIN_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLTNx_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLTNx_EPSILON</td>
<td></td>
</tr>
<tr>
<td>FLTNx_MIN</td>
<td></td>
</tr>
<tr>
<td>FLTNx_MAX</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _DecimalN, where N ≠ 32, 64, and 128:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECN_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>DECN_DECIMAL_DIG</td>
<td></td>
</tr>
<tr>
<td>DECN_DIG</td>
<td></td>
</tr>
<tr>
<td>DECN_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>DECN_MAX_EXP</td>
<td></td>
</tr>
<tr>
<td>DECN_MIN</td>
<td></td>
</tr>
<tr>
<td>DECN_MAX</td>
<td></td>
</tr>
<tr>
<td>DECN_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>DECN_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>DECN_EPSILON</td>
<td></td>
</tr>
<tr>
<td>DECN_MIN</td>
<td></td>
</tr>
<tr>
<td>DECN_MAX</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _DecimalNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECNx_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>DECNx_DECIMAL_DIG</td>
<td></td>
</tr>
<tr>
<td>DECNx_DIG</td>
<td></td>
</tr>
<tr>
<td>DECNx_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>DECNx_MAX_EXP</td>
<td></td>
</tr>
<tr>
<td>DECNx_MIN</td>
<td></td>
</tr>
<tr>
<td>DECNx_MAX</td>
<td></td>
</tr>
<tr>
<td>DECNx_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>DECNx_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>DECNx_EPSILON</td>
<td></td>
</tr>
<tr>
<td>DECNx_MIN</td>
<td></td>
</tr>
<tr>
<td>DECNx_MAX</td>
<td></td>
</tr>
</tbody>
</table>

20 After 7.12#1c, insert the paragraph:

[1d] The following identifiers are defined or declared only if __STDC_WANT_IEC_60559_TYPES_EXT__ is defined as a macro at the point in the source file where <math.h> is first included:
for supported types _FloatN:

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_FN</td>
<td>modffN</td>
</tr>
<tr>
<td>SNANF</td>
<td>scalbnfN</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>tgammafN</td>
</tr>
<tr>
<td>tanhfN</td>
<td>ceilfN</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>llogbfN</td>
<td>roundfN</td>
</tr>
<tr>
<td>ldexpfN</td>
<td>lroundfN</td>
</tr>
<tr>
<td>logfN</td>
<td>lroundfN</td>
</tr>
<tr>
<td>log10fN</td>
<td>truncfN</td>
</tr>
<tr>
<td>log1pfN</td>
<td>fromfpfN</td>
</tr>
<tr>
<td>log2fN</td>
<td>ufromfpfN</td>
</tr>
</tbody>
</table>
| for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_FNX</td>
<td>logbfNx</td>
</tr>
<tr>
<td>SNANF</td>
<td>modffNx</td>
</tr>
<tr>
<td>FP_FAST_FMAFN</td>
<td>scalbnfNx</td>
</tr>
<tr>
<td>acosfN</td>
<td>scalblnfN</td>
</tr>
<tr>
<td>asinfN</td>
<td>cbrtfN</td>
</tr>
<tr>
<td>atanfN</td>
<td>fabsfN</td>
</tr>
<tr>
<td>atan2fN</td>
<td>hypotfN</td>
</tr>
<tr>
<td>cosfN</td>
<td>powfN</td>
</tr>
<tr>
<td>sinfN</td>
<td>sqrtfN</td>
</tr>
<tr>
<td>tanfN</td>
<td>erfN</td>
</tr>
<tr>
<td>acoshfN</td>
<td>erfcfN</td>
</tr>
<tr>
<td>asinhfN</td>
<td>lgammafN</td>
</tr>
<tr>
<td>tanhfN</td>
<td>tgammafN</td>
</tr>
<tr>
<td>expfN</td>
<td>ceilfN</td>
</tr>
<tr>
<td>exp2fN</td>
<td>floorfN</td>
</tr>
<tr>
<td>expm1fN</td>
<td>nearbyintfN</td>
</tr>
<tr>
<td>frexpfn</td>
<td>rintfN</td>
</tr>
<tr>
<td>ilogbfN</td>
<td>lrintfN</td>
</tr>
<tr>
<td>llogbfN</td>
<td>llrintfN</td>
</tr>
<tr>
<td>ldexpfN</td>
<td>roundfN</td>
</tr>
<tr>
<td>logfN</td>
<td>lroundfN</td>
</tr>
<tr>
<td>log10fN</td>
<td>truncfN</td>
</tr>
<tr>
<td>log1pfN</td>
<td>fromfpfN</td>
</tr>
<tr>
<td>log2fN</td>
<td>ufromfpfN</td>
</tr>
<tr>
<td>for supported types _FloatNx:</td>
<td></td>
</tr>
</tbody>
</table>
for supported types _FloatM and _FloatN where \( M < N \):

\[
\begin{align*}
&MaddfN \\
&MsubfN \\
&MdivfN \\
&MsqrtfN \\
&MfmafN
\end{align*}
\]

for supported types _FloatM and _FloatNx where \( M \leq N \):

\[
\begin{align*}
5 & MaddfNx \\
5 & MsubfNx \\
5 & MdivfNx \\
5 & MsqrtfNx \\
5 & MfmafNx
\end{align*}
\]

for supported types _FloatMx and _FloatN where \( M < N \):

\[
\begin{align*}
&MaddfN \\
&MsubfN \\
&MdivfN \\
&MsqrtfN \\
&MfmafN
\end{align*}
\]

for supported types _FloatMx and _FloatNx where \( M < N \):

\[
\begin{align*}
&MaddfNx \\
&MsubfNx \\
&MdivfNx \\
&MsqrtfNx \\
&MfmafNx
\end{align*}
\]

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

\[
\begin{align*}
&FMencfN
\end{align*}
\]

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

\[
\begin{align*}
\text{HUGE\_VAL\_D\_N} & \text{ scalblndN} \\
\text{SNAND} & \text{ cbrtdN} \\
\text{FP\_FAST\_FMAD} & \text{ fabsdN} \\
\text{acosdN} & \text{ hypotdN} \\
\text{asindN} & \text{ powdN} \\
\text{atan2dN} & \text{ sqrtddN} \\
\text{cosdN} & \text{ erfcdN} \\
\text{sindN} & \text{ lgammadN} \\
\text{tandN} & \text{ tgammaN} \\
\text{acosdhN} & \text{ ceilldN} \\
\text{asinhdN} & \text{ floordN} \\
\text{atanhdN} & \text{ nearbyintdN} \\
\text{expdN} & \text{ rintdN} \\
\text{exp2dN} & \text{ lrintdN} \\
\text{expmlldN} & \text{ llrintdN} \\
\text{fexpdN} & \text{ rounddN} \\
\text{ilogbdN} & \text{ lrounddN} \\
\text{ilogbNd} & \text{ lroundNd} \\
\text{ldexpdN} & \text{ truncdN} \\
\text{logN} & \text{ roundevendN} \\
\text{log10dN} & \text{ fromfpdN} \\
\text{log1pdN} & \text{ ufrofpdN} \\
\text{log2dN} & \text{ fromfpdxN} \\
\text{logbN} & \text{ ufrofpdxN} \\
\text{modfN} & \text{ fmodN} \\
\text{scalbndN} & \text{ remainderdN}
\end{align*}
\]
for supported types _DecimalNX:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_D NX</td>
<td>scalbnDx</td>
<td>fmodDx</td>
</tr>
<tr>
<td>SNAND /</td>
<td>scalblndx</td>
<td>remainderDx</td>
</tr>
<tr>
<td>FP_FAST_FMAD /</td>
<td>cbtdx</td>
<td>remquodX</td>
</tr>
<tr>
<td>acosdX</td>
<td>fabsdx</td>
<td>copysignDx</td>
</tr>
<tr>
<td>asindX</td>
<td>hypotdx</td>
<td>nandX</td>
</tr>
<tr>
<td>atandX</td>
<td>powdx</td>
<td>nextafterDx</td>
</tr>
<tr>
<td>atan2dX</td>
<td>sqrtDx</td>
<td>nextupDx</td>
</tr>
<tr>
<td>cosdX</td>
<td>erfdx</td>
<td>nextdownDx</td>
</tr>
<tr>
<td>sindX</td>
<td>erfcdx</td>
<td>canonicalizedDx</td>
</tr>
<tr>
<td>tandX</td>
<td>lgammadx</td>
<td>quantizedDx</td>
</tr>
<tr>
<td>acoshdX</td>
<td>tgammaDx</td>
<td>samequantumedX</td>
</tr>
<tr>
<td>asinhdX</td>
<td>ceilDx</td>
<td>quantumedX</td>
</tr>
<tr>
<td>atanhdX</td>
<td>floorDx</td>
<td>llquantexpDx</td>
</tr>
<tr>
<td>expdX</td>
<td>nearbyintDx</td>
<td>fdimDx</td>
</tr>
<tr>
<td>exp2dX</td>
<td>rintDx</td>
<td>fmaxDx</td>
</tr>
<tr>
<td>expmlDx</td>
<td>lrintDx</td>
<td>fminDx</td>
</tr>
<tr>
<td>frexpDx</td>
<td>llrintDx</td>
<td>fmaxmagDx</td>
</tr>
<tr>
<td>logbdX</td>
<td>roundDx</td>
<td>fminmagDx</td>
</tr>
<tr>
<td>log10dX</td>
<td>truncDx</td>
<td>totalorderDx</td>
</tr>
<tr>
<td>log1pdX</td>
<td>fromfpDx</td>
<td>getpayloadDx</td>
</tr>
<tr>
<td>log2dX</td>
<td>ufromfpDx</td>
<td>setpayloadDx</td>
</tr>
<tr>
<td>logbdX</td>
<td>fromfpDx</td>
<td>setpayloadsgDx</td>
</tr>
<tr>
<td>modfDx</td>
<td>ufromfpDx</td>
<td></td>
</tr>
</tbody>
</table>

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_FAST_D / ADDDN</td>
<td>FP_FAST_D / MSQRTDN</td>
<td>dMmulD</td>
</tr>
<tr>
<td>FP_FAST_D / SUBDN</td>
<td>FP_FAST_D / FMADN</td>
<td>dMdivD</td>
</tr>
<tr>
<td>FP_FAST_D / MULDN</td>
<td>dMaddD</td>
<td>dMsqrtD</td>
</tr>
<tr>
<td>FP_FAST_D / DIVDN</td>
<td>dMsubD</td>
<td>dMfmadD</td>
</tr>
</tbody>
</table>

for supported types _DecimalM and _DecimalN where M ≤ N:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_FAST_D / ADDDNX</td>
<td>FP_FAST_D / MSQRTDNX</td>
<td>dMmulDX</td>
</tr>
<tr>
<td>FP_FAST_D / SUBDNX</td>
<td>FP_FAST_D / FMADNX</td>
<td>dMdivDX</td>
</tr>
<tr>
<td>FP_FAST_D / MULDNX</td>
<td>dMaddDX</td>
<td>dMsqrtDX</td>
</tr>
<tr>
<td>FP_FAST_D / DIVDNX</td>
<td>dMsubDX</td>
<td>dMfmadNX</td>
</tr>
</tbody>
</table>

for supported types _DecimalMx and _DecimalN where M < N:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_FAST_D / MXADDN</td>
<td>FP_FAST_D / MXSQRTDN</td>
<td>dMxmulD</td>
</tr>
<tr>
<td>FP_FAST_D / MXSUBN</td>
<td>FP_FAST_D / MXFMADN</td>
<td>dMxdivD</td>
</tr>
<tr>
<td>FP_FAST_D / MXMULDN</td>
<td>dMxaddD</td>
<td>dMxsqrtD</td>
</tr>
<tr>
<td>FP_FAST_D / MXDIVDN</td>
<td>dMxsubD</td>
<td>dMxfmadD</td>
</tr>
</tbody>
</table>
for supported types _DecimalMx and _DecimalNx where \( M < N \):

\[
\begin{align*}
\text{FP}_\text{FAST}_\text{DM}_\text{XADD}D\text{N} & \quad \text{FP}_\text{FAST}_\text{DM}_\text{XSQRTD}D\text{N} & \quad \text{DM}_\text{Xmul}D\text{N} \\
\text{FP}_\text{FAST}_\text{DM}_\text{XSUB}D\text{N} & \quad \text{FP}_\text{FAST}_\text{DM}_\text{XFMD}D\text{N} & \quad \text{DM}_\text{Xdiv}D\text{N} \\
\text{FP}_\text{FAST}_\text{DM}_\text{XMD}D\text{N} & \quad \text{DM}_\text{Xadd}D\text{N} & \quad \text{DM}_\text{Xsq}rtD\text{N} \\
\text{FP}_\text{FAST}_\text{DM}_\text{XMDD}D\text{N} & \quad \text{DM}_\text{Xsub}D\text{N} & \quad \text{DM}_\text{Xfma}D\text{N}
\end{align*}
\]

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

\[
\begin{align*}
\text{dM}\text{ncd}eD\text{N} & \quad \text{dM}\text{ncb}iD\text{N}
\end{align*}
\]

After 7.22#1b, insert the paragraph:

10 [1c] The following identifiers are declared only if \__STDC_WANT_IEC_60559_TYPES_EXT\__ is defined as a macro at the point in the source file where \<stdlib.h\> is first included:

for supported types _FloatN:

\[
\begin{align*}
\text{strfrom}\text{f}\text{N} & \quad \text{strto}\text{f}\text{N}
\end{align*}
\]

for supported types _FloatNx:

\[
\begin{align*}
\text{strfrom}\text{f}\text{N}x & \quad \text{strto}\text{f}\text{N}x
\end{align*}
\]

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

\[
\begin{align*}
\text{strfrom}\text{d}\text{N} & \quad \text{strto}\text{d}\text{N}
\end{align*}
\]

for supported types _DecimalNx:

\[
\begin{align*}
\text{strfrom}\text{f}\text{N}x & \quad \text{strto}\text{d}\text{N}x
\end{align*}
\]

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

\[
\begin{align*}
\text{strfrom}\text{en}\text{c}D\text{N} & \quad \text{strto}\text{en}\text{c}D\text{N}
\end{align*}
\]

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

\[
\begin{align*}
\text{strfrom}\text{enc}\text{d}eD\text{N} & \quad \text{strto}\text{enc}\text{d}eD\text{N} \\
\text{strfrom}\text{enc}\text{b}iD\text{N} & \quad \text{strto}\text{enc}\text{b}iD\text{N}
\end{align*}
\]

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. Tables 1 and 2 give parameters for the IEC 60559 interchange formats.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>binary16</th>
<th>binary32</th>
<th>binary64</th>
<th>binary128</th>
<th>binaryN ((N \geq 128))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N), storage width in bits</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>multiple of 32</td>
</tr>
<tr>
<td>(p), precision in bits</td>
<td>11</td>
<td>24</td>
<td>53</td>
<td>113</td>
<td>(N - \text{round}(4 \times \log_2(N)) + 13)</td>
</tr>
<tr>
<td>(e_{\text{max}}, \text{maximum exponent } e)</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
<td>(2^{(e_{\text{max}} - 1)} - 1)</td>
</tr>
</tbody>
</table>

**Encoding parameters**

| \(bias, \ E-e\) | 15 | 127 | 1023 | 16383 | \(e_{\text{max}}\) |
| sign bit | 1 | 1 | 1 | 1 | 1 |
| \(w\), exponent field width in bits | 5 | 8 | 11 | 15 | \(\text{round}(4 \times \log_2(N)) - 13\) |
| \(t\), trailing significand field width in bits | 10 | 23 | 52 | 112 | \(N - w - 1\) |
| \(N\), storage width in bits | 16 | 32 | 64 | 128 | \(1 + w + t\) |

The function \(\text{round}()\) in Table 1 rounds to the nearest integer. For example, binary256 would have \(p = 237\) and \(e_{\text{max}} = 262143\).

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

**Encoding parameters**

| \(bias, \ E-e\) | 101 | 398 | 6176 | \(e_{\text{max}} + p - 2\) |
| sign bit | 1 | 1 | 1 | 1 |
| \(w\), exponent field width in bits | 11 | 13 | 17 | \(N/16 + 9\) |
| \(t\), trailing significand field width in bits | 20 | 50 | 110 | \(15 \times N/16 - 10\) |
| \(N\), storage width in bits | 32 | 64 | 128 | \(1 + 5 + w + t\) |

For example, decimal256 would have \(p = 70\) and \(e_{\text{max}} = 1572864\).

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

```c
_decimalN, where N ≥ 32 and a multiple of 32
```

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

\[10\text{c} \] An implementation that defines \_STDC\_IEC\_60559\_BFP and \_STDC\_IEC\_60559\_TYPES shall provide \_Float32 and \_Float64 as interchange floating types with the same representation and alignment requirements as float and double, respectively. If the implementation’s long double type supports an IEC 60559 interchange format of width \(N > 64\), then the implementation shall also provide the type \_Float\(N\) as an interchange floating type with the same representation and alignment requirements as long double. The implementation may provide other binary interchange floating types.

\[10\text{d} \] An implementation that defines \_STDC\_IEC\_60559\_DFP shall provide the decimal floating types \_Decimal32, \_Decimal64, and \_Decimal128. If the implementation also defines \_STDC\_IEC\_60559\_TYPES, it may provide other decimal floating types.

\[10\text{e} \] 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 (7.12.11c), 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 binary16 format, at least as a non-arithmetic interchange format.

\[10\text{f} \] 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. Table 3 below gives the minimum values of these parameters:

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

\[10\text{g} \] Types designated \_Float32x, \_Float64x, \_Float128x, \_Decimal64x, and \_Decimal128x support the corresponding IEC 60559 extended formats and are collectively called the extended floating types. Extended floating types are not compatible with any other types. An implementation that defines \_STDC\_IEC\_60559\_BFP and \_STDC\_IEC\_60559\_TYPES shall provide \_Float32x, which may have the same set of values as double, and may provide any of the other two binary extended floating types. An implementation that defines \_STDC\_IEC\_60559\_DFP and \_STDC\_IEC\_60559\_TYPES shall provide: \_Decimal64x, which may have the same set of values as \_Decimal128, and may provide \_Decimal128x.

\[10\text{h} \] 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.

with:

[11] 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
— _Decimal64x
— _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), _Decimal64x, and _Decimal128x shall not be used if the implementation does not support the corresponding types (see 6.10.8.3).

Replace 6.5#8a:

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

with:

[8a] Expressions 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] Except for assignment and cast (which remove all extra range and precision), the values of the decimal floating operands and results of expressions that are subject to the usual arithmetic conversions, and the values of floating constants of decimal floating type, are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of DEC_EVAL_METHOD: [ISSUE: wording here is a clarification of the analogous specification in 5.2.4.2.2#9 - how to fix 5.2.4.2.2?]

- 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

<table>
<thead>
<tr>
<th>DEC32_MANT_DIG</th>
<th>DEC64_MANT_DIG</th>
<th>DEC128_MANT_DIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>16</td>
<td>34</td>
</tr>
</tbody>
</table>
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 indicates a binary interchange floating type of width N. The prefix FLT NX indicates a binary extended floating type that extends a basic format of width N. The prefix DEC N indicates a decimal floating type of width N. The prefix DEC NX indicates a decimal extended floating type that extends a basic format of width N. The type parameters p, e max, and e min for extended floating types are for the extended floating type itself, not for the basic format that it extends. For each interchange or extended floating type that the implementation provides, <float.h> shall define the associated macros in the following lists. Conversely, for each such type that the implementation does not provide, <float.h> shall not define the associated macros in the following lists.
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 the types `_Decimal32`, `_Decimal64`, and `_Decimal128`:

-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.

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

- radix of exponent representation, \( b \) (= 2 for binary, 10 for decimal)

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

- number of 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,

\[
\text{ceiling}(1 + p \log_{10} 2)
\]

- number of bits in the floating-point significand, \( p \)

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

- number of digits in the coefficient, \( p \)

- 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, ceiling(1 + \( p \log_{10} 2\))

- 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, floor((\( p - 1 \)) log_{10} 2)
— minimum negative integer such that the radix raised to one less than that power is a normalized floating-point number, $e_{\text{min}}$

\[
\text{FLT}_N\_\text{MIN}\_\text{EXP} \\
\text{DEC}_N\_\text{MIN}\_\text{EXP}
\]

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

\[
\text{FLT}_N\_\text{MIN}_\text{10}\_\text{EXP} \\
\text{DEC}_N\_\text{MIN}_\text{10}\_\text{EXP}
\]

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

\[
\text{FLT}_N\_\text{MAX}\_\text{EXP} \\
\text{DEC}_N\_\text{MAX}\_\text{EXP}
\]

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

\[
\text{FLT}_N\_\text{MAX}_\text{10}\_\text{EXP} \\
\text{DEC}_N\_\text{MAX}_\text{10}\_\text{EXP}
\]

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

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

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

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

— minimum normalized positive floating-point number, $b^{e_{\text{min}}-1}$

\[
\text{FLT}_N\_\text{MIN} \\
\text{DEC}_N\_\text{MIN}
\]

— minimum positive subnormal floating-point number, $b^{e_{\text{min}}-p}$

\[
\text{FLT}_N\_\text{TRUE}\_\text{MIN} \\
\text{DEC}_N\_\text{TRUE}\_\text{MIN}
\]
With the following change, `DECIMAL_DIG` characterizes conversions of supported IEC 60559 encodings, which may be wider than supported floating types.

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

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

5. number of decimal digits, \( n \), such that any floating-point number in the widest supported floating type with ...

to:

5. number of decimal digits, \( n \), such that any floating-point number in the widest of the supported floating types and the supported IEC 60559 encodings with ...

### 8 Conversions

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

**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:"

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

First, if the type of either operand is `_Decimal128`, the other operand is converted to `_Decimal128`.

Otherwise, if the type of either operand is `_Decimal64`, the other operand is converted to `_Decimal64`.

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

If there are no decimal floating types in the operands:

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

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

Otherwise, if the corresponding real type of either operand is `float`, the other operand is converted, without change of type domain, to a type whose corresponding real type is `float.62` 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.
Otherwise, if both operands are floating types and the sets of values of their corresponding real types are equivalent, then the following rules are applied:

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

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

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

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

Otherwise, if one operand has a floating type, the other operand is converted to the corresponding real type of the operand of floating type.

9 Constants

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

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

Change floating-suffix in 6.4.4.2 from:

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

to:

floating-suffix: one of
  f l F L fN FN fNx FNx dN DN dNx DNx

Replace 6.4.4.2#2a:

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

with:

[2a] A floating-suffix dN, DN, dNx, or DNx shall not be used in a hexadecimal-floating-constant.

Replace 6.4.4.2#4a:

[4a] If a floating constant is suffixed by df or DF, it has type _Decimal128. 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 dN or DN, it has type _DecimalN. If suffixed by dNx or DNx, it has type _DecimalNx.
Replace the second sentence of 6.4.4.2#5a:

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

with:

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

10 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 Annes 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 type `float`, `long double`, `_FloatN`, `_FloatN`, `_DecimalN`, and `_DecimalN`, respectively.

Add after 7.12#1d:

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

11.1 Macros

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

Replace 7.12#3a:

[3a] The macro

    HUGE_VAL_D64

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

    HUGE_VAL_D32
    HUGE_VAL_D128

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

with:

[3a] The macros

    HUGE_VAL_F
    HUGE_VAL_D
    HUGE_VAL_FNX
    Huge_VAL_DNX

expand to constant expressions of types `_Float`, `_Decimal`, `_FloatN`, and `_DecimalN`, respectively, representing positive infinity.

Replace 7.12#5b:

[5b] The decimal signaling NaN macros

    SNAND32
    SNAND64
    SNAND128

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

with:

[5b] The signaling NaN macros

    SNANFN
    SNANDN
    SNANFNX
    SNANDNX

expand to constant expressions of types `_FloatN`, `_DecimalN`, `_FloatNx`, and `_DecimalNx`, respectively, representing a signaling NaN. If a signaling NaN macro is used for initializing an object of the same type that has static or thread-local storage duration, the object is initialized with a signaling NaN value.
Replace 7.12#7b:

[7b] The macros

```
FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128
```

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

with:

[7b] The macros

```
FP\_FAST\_FMAFN
FP\_FAST\_FMAF6N
FP\_FAST\_FMAFNX
FP\_FAST\_FMAFNXN
```

are, respectively, \_FloatN, \_DecimalN, \_FloatNx, and \_DecimalNx analogues of \texttt{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 \texttt{FP\_FAST\_FADD}, \texttt{FP\_FAST\_FADDL}, \texttt{FP\_FAST\_DADDL}, etc.

with:

[7c] The macros in the following lists are interchange and extended floating type analogues of \texttt{FP\_FAST\_FADD}, \texttt{FP\_FAST\_FADDL}, \texttt{FP\_FAST\_DADDL}, etc.
[7d] For \( M < N \), the macros

\[
\begin{align*}
&FP\_FAST\_FMADDFN \\
&FP\_FAST\_FMSUBFN \\
&FP\_FAST\_FMULFN \\
&FP\_FAST\_FMDIVFN \\
&FP\_FAST\_FMSQRTFN \\
&FP\_FAST\_DMADDN \\
&FP\_FAST\_DMSUBN \\
&FP\_FAST\_DMMULDN \\
&FP\_FAST\_DMDIVDN \\
&FP\_FAST\_DMFMADN \\
&FP\_FAST\_DMSQRTDN \\
\end{align*}
\]

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

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

\[
\begin{align*}
&FP\_FAST\_FMADDNX \\
&FP\_FAST\_FMSUBNX \\
&FP\_FAST\_FMULNX \\
&FP\_FAST\_FMDIVNX \\
&FP\_FAST\_FMSQRTNX \\
&FP\_FAST\_DMADDNX \\
&FP\_FAST\_DMSUBNX \\
&FP\_FAST\_DMMULDNX \\
&FP\_FAST\_DMDIVDNX \\
&FP\_FAST\_DMFMADNX \\
&FP\_FAST\_DMSQRTDNX \\
\end{align*}
\]

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

[7f] For \( M < N \), the macros

\[
\begin{align*}
&FP\_FAST\_FXMADDN \\
&FP\_FAST\_FXSUBN \\
&FP\_FAST\_FXMULN \\
&FP\_FAST\_FXDIVN \\
&FP\_FAST\_FXSQRTN \\
&FP\_FAST\_FXMADDN \\
&FP\_FAST\_FXSUBN \\
&FP\_FAST\_FXMULDN \\
&FP\_FAST\_FXDIVDN \\
&FP\_FAST\_FXFMADN \\
&FP\_FAST\_FXSQRTDN \\
\end{align*}
\]

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

\[
\begin{align*}
\text{FP\_FAST\_FM\_ADDF\_N} \\
\text{FP\_FAST\_FM\_SUBF\_N} \\
\text{FP\_FAST\_FM\_MULF\_N} \\
\text{FP\_FAST\_FM\_DIVF\_N} \\
\text{FP\_FAST\_FM\_SQRF\_N} \\
\text{FP\_FAST\_DM\_ADDD\_N} \\
\text{FP\_FAST\_DM\_SUBD\_N} \\
\text{FP\_FAST\_DM\_MULD\_N} \\
\text{FP\_FAST\_DM\_DIVD\_N} \\
\text{FP\_FAST\_DM\_XMAD\_N} \\
\text{FP\_FAST\_DM\_MSQRTD\_N}
\end{align*}
\]

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

### 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

\[
\begin{align*}
\_\text{FloatN acosf N\_FloatN \(x\)}; \\
\_\text{FloatN asinf N\_FloatN \(x\)}; \\
\_\text{DecimalN acosd N\_DecimalN \(x\)}; \\
\_\text{DecimalN asind N\_DecimalN \(x\)}; \\
\_\text{FloatN atanf N\_FloatN \(x\)}; \\
\_\text{FloatN atand N\_DecimalN \(x\)}; \\
\_\text{FloatN atan2f N\_FloatN y N\_FloatN \(x\)}; \\
\_\text{FloatN atan2d N\_DecimalN y N\_DecimalN \(x\)}; \\
\_\text{FloatN cosf N\_FloatN \(x\)}; \\
\_\text{FloatN csinf N\_FloatN \(x\)}; \\
\_\text{DecimalN csind N\_DecimalN \(x\)}; \\
\_\text{FloatN tanf N\_FloatN \(x\)};
\end{align*}
\]
7.12.5 Hyperbolic functions

_FloatN acoshF(_FloatN x);
_FloatN acoshF(_FloatN x);
DecimalN acoshD(_DecimalN x);
DecimalN acoshD(_DecimalN x);

_FloatN asinhF(_FloatN x);
_FloatN asinhF(_FloatN x);
DecimalN asinhD(_DecimalN x);
DecimalN asinhD(_DecimalN x);

_FloatN atanhF(_FloatN x);
_FloatN atanhF(_FloatN x);
DecimalN atanhD(_DecimalN x);
DecimalN atanhD(_DecimalN x);

_FloatN coshf(_FloatN x);
_FloatN coshf(_FloatN x);
DecimalN coshD(_DecimalN x);
DecimalN scoshD(_DecimalN x);

_FloatN sinhF(_FloatN x);
_FloatN sinhF(_FloatN x);
DecimalN sinhD(_DecimalN x);
DecimalN sinhD(_DecimalN x);

_FloatN tanhf(_FloatN x);
_FloatN tanhf(_FloatN x);
DecimalN tanhd(_DecimalN x);
DecimalN tanhd(_DecimalN x);

7.12.6 Exponential and logarithmic functions

_FloatN expf(_FloatN x);
_FloatN expf(_FloatN x);
DecimalN expd(_DecimalN x);
DecimalN expd(_DecimalN x);

_FloatN exp2fN(_FloatN x);
_FloatN exp2fN(_FloatN x);
DecimalN exp2dN(_DecimalN x);
DecimalN exp2dN(_DecimalN x);

_FloatN expmlfN(_FloatN x);
_FloatN expmlfN(_FloatN x);
DecimalN expmldN(_DecimalN x);
DecimalN expmldN(_DecimalN x);

_FloatN frexpF(_FloatN value, int *exp);
_FloatN frexpF(_FloatN value, int *exp);
_DecimalN frexpfdN(_DecimalN value, int *exp);
_decimalNx frexpfdN(_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 ldexpfN(_FloatNx value, int exp);
_decimalN ldexpdN(_DecimalN value, int exp);
_decimalNx ldexpdN(_DecimalNx value, int exp);

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

_FloatN logfN(_FloatN x);
_FloatNx logfNx(_FloatNx x);
_decimalN logdN(_DecimalN x);
_decimalNx logdNx(_DecimalNx x);

_FloatN log10fN(_FloatN x);
_FloatNx log10fNx(_FloatNx x);
_decimalN log10dN(_DecimalN x);
_decimalNx log10dNx(_DecimalNx x);

_FloatN log1pfN(_FloatN x);
_FloatNx log1pfNx(_FloatNx x);
_decimalN log1pdN(_DecimalN x);
_decimalNx log1pdNx(_DecimalNx x);

_FloatN log2fN(_FloatN x);
_FloatNx log2fNx(_FloatNx x);
_decimalN log2dN(_DecimalN x);
_decimalNx log2dNx(_DecimalNx x);

_FloatN logbfN(_FloatN x);
_FloatNx logbfNx(_FloatNx x);
_decimalN logbdN(_DecimalN x);
_decimalNx logbdNx(_DecimalNx x);

_FloatN modffN(_FloatN x, _FloatN *iptr);
_FloatNx modffNx(_FloatNx x, _FloatNx *iptr);
_decimalN modfdN(_DecimalN x, _DecimalN *iptr);
_decimalNx modfdNx(_DecimalNx x, _DecimalNx *iptr);

_FloatN scalbnfN(_FloatN value, int exp);
_FloatNx scalbnfN(_FloatNx value, int exp);
_decimalN scalbdN(_DecimalN value, int exp);
_decimalNx scalbdN(_DecimalNx value, int exp);

_FloatN scalblnfN(_FloatN value, long int exp);
_FloatNx scalblnfN(_FloatNx value, long int exp);
_decimalN scalblndN(_DecimalN value, long int exp);
_decimalNx scalblndN(_DecimalNx value, long int exp);
7.12.7 Power and absolute-value functions

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

_FloatN fabsfN(_FloatN x);
_FloatN fabsfNx(_FloatNx x);
_DecimalN fabsdN(_DecimalN x);
_DecimalN fabsdNx(_DecimalNx x);

_FloatN hypotfN(_FloatN x, _FloatN y);
_FloatN hypotfNx(_FloatNx x, _FloatNx y);
_DecimalN hypotdN(_DecimalN x, _DecimalN y);
_DecimalN hypotdNx(_DecimalNx x, _DecimalNx y);

_FloatN powfN(_FloatN x, _FloatN y);
_FloatN powfNx(_FloatNx x, _FloatNx y);
_DecimalN powdN(_DecimalN x, _DecimalN y);
_DecimalN powdNx(_DecimalNx x, _DecimalNx y);

_FloatN sqrtfN(_FloatN x);
_FloatN sqrtfNx(_FloatNx x);
_DecimalN sqrtdN(_DecimalN x);
_DecimalN sqrtdNx(_DecimalNx x);

7.12.8 Error and gamma functions

_FloatN erfN(_FloatN x);
_FloatN erfNx(_FloatNx x);
_DecimalN erfdN(_DecimalN x);
_DecimalN erfdNx(_DecimalNx x);

_FloatN erfcfN(_FloatN x);
_FloatN erfcfNx(_FloatNx x);
_DecimalN erfcfN(_DecimalN x);
_DecimalN erfcfNx(_DecimalNx x);

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

_FloatN tgammafN(_FloatN x);
_FloatN tgammafNx(_FloatNx x);
_DecimalN tgammafN(_DecimalN x);
_DecimalN tgammafNx(_DecimalNx x);

7.12.9 Nearest integer functions

_FloatN ceilfN(_FloatN x);
_FloatN ceilfNx(_FloatNx x);
_DecimalN ceilfN(_DecimalN x);
_DecimalN ceilfNx(_DecimalNx x);
_DecimalNx ceil(Nx(_DecimalN x));

_FloatNx floorf(N _FloatN x);
_FloatNx floorfNx(_FloatN x);  
_DecimalNx floordN(_DecimalN x);
_DecimalNx floordNx(_DecimalN x);

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

_FloatNx rintfN(_FloatN x);
_FloatNx rintfNx(_FloatN x);
_DecimalNx rintdN(_DecimalN x);
_DecimalNx 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);

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

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

_FloatNx truncfN(_FloatN x);
_FloatNx truncfNx(_FloatN x);
_DecimalNx truncaN(_DecimalN x);
_DecimalNx truncaNx(_DecimalN x);

intmax_t fromfpN(_FloatN x, int round, unsigned int width);
intmax_t fromfpNx(_FloatN x, int round, unsigned int width);
intmax_t fromfpdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpdNx(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfndN(_DecimalN x, int round, unsigned int width);

intmax_t fromfpfN(_FloatN x, int round, unsigned int width);
intmax_t fromfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfndN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfndN(_DecimalN x, int round, unsigned int width);

7.12.10 Remainder functions

_FloatN fmodfN(_FloatN x, _FloatN y);
_FloatN fmodfN(_FloatN x, _FloatN y);
_FloatN fmoddN(_DecimalN x, _DecimalN y);
_FloatN fmoddN(_DecimalN x, _DecimalN y);

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

_FloatN remquofN(_FloatN x, _FloatN y, int *quo);
_FloatN remquofN(_FloatN x, _FloatN y, int *quo);

7.12.11 Manipulation functions

_FloatN copysignfN(_FloatN x, _FloatN y);
_FloatN copysignfN(_FloatN x, _FloatN y);
_FloatN copysigndN(_DecimalN x, _DecimalN y);
_FloatN copysigndN(_DecimalN x, _DecimalN y);

_FloatN nanf(const char *tagp);
_FloatN nanf(const char *tagp);
_FloatN nandN(const char *tagp);
_FloatN nandN(const char *tagp);

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

_FloatN nextupfN(_FloatN x);
_FloatN nextupfN(_FloatN x);
_FloatN nextupdN(_DecimalN x);
_FloatN nextupdN(_DecimalN x);

_FloatN nextdownfN(_FloatN x);
_FloatN nextdownfN(_FloatN x);
_FloatN nextdowndN(_DecimalN x);
_FloatN nextdowndN(_DecimalN x);
ISO/IEC TS 18661
Working Group Draft – September 16, 2013
WG 14 N1758

int canonicalizeN(_FloatN * cx, const _FloatN * x);
int canonicalizeNx(_FloatNx * cx, const _FloatNx * x);
int canonicalizedN(_DecimalN * cx, const _DecimalN * x);
int canonicalizedNx(_DecimalNx * cx, const _DecimalNx * x);

_DecimalN quantizedN(_DecimalN x, _DecimalN y);
_DecimalNx quantizedNx(_DecimalNx x, _DecimalNx y);

_Bool samequantumN(_DecimalN x, _DecimalN y);
_Bool samequantumNx(_DecimalNx x, _DecimalNx y);

_DecimalN quantumN(_DecimalN x);
_DecimalNx quantumNx(_DecimalNx x);

long long llquantexpdN(_DecimalN x);
long long llquantexpNx(_DecimalNx x);

void encodedecdN(unsigned char * restrict encptr, const _DecimalN * restrict xptr);
void decodedecdN(_DecimalN * restrict xptr, const unsigned char * restrict encptr);
void encodebindN(unsigned char * restrict encptr, const _DecimalN * restrict xptr);
void decodebindN(_DecimalN * restrict xptr, const unsigned char * restrict encptr);

7.12.12 Maximum, minimum, and positive difference functions

_FloatN fdimfN(_FloatN x, _FloatN y);
_FloatNx fdimfNx(_FloatNx x, _FloatNx y);
_DecimalN fdimdN(_DecimalN x, _DecimalN y);
_DecimalNx fdimdNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaxfN(_FloatN x, _FloatN y);
_FloatNx fmaxfNx(_FloatNx x, _FloatNx y);
_DecimalN fmaxdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaxdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminfN(_FloatN x, _FloatN y);
_FloatNx fminfNx(_FloatNx x, _FloatNx y);
_DecimalN fmindN(_DecimalN x, _DecimalN y);
_DecimalNx fmindNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaxmagfN(_FloatN x, _FloatN y);
_FloatNx fmaxmagNx(_FloatNx x, _FloatNx y);
_DecimalN fmaxmagdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaxmagdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminmagfN(_FloatN x, _FloatN y);
_FloatNx fminmagNx(_FloatNx x, _FloatNx y);
_DecimalN fminmagdN(_DecimalN x, _DecimalN y);
_DecimalNx fminmagdNx(_DecimalNx x, _DecimalNx y);

7.12.13 Floating multiply-add

_FloatN fmafN(_FloatN x, _FloatN y, _FloatN z);
_FloatNx fmafNx(_FloatNx x, _FloatNx y, _FloatNx z);
7.12.14 Functions that round result to narrower format

_FloatM fmaddN(_DecimalN x, _DecimalN y, _DecimalN z);
_DecimalN fmaddN(_DecimalN x, _DecimalN y, _DecimalN z);

_FloatM fMaddN(_FloatN x, _FloatN y); // M < N
_FloatM fMaddN(_FloatN x, _FloatN y); // M <= N
_FloatM fMxaddN(_FloatN x, _FloatN y); // M < N
_FloatM fMxaddN(_FloatN x, _FloatN y); // M <= N
_DecimalN dMaddN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMaddN(_DecimalN x, _DecimalN y); // M <= N
_DecimalN dMxaddN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMxaddN(_DecimalN x, _DecimalN y); // M <= N

_FloatM fMsubN(_FloatN x, _FloatN y); // M < N
_FloatM fMsubN(_FloatN x, _FloatN y); // M <= N
_FloatM fMxsubN(_FloatN x, _FloatN y); // M < N
_FloatM fMxsubN(_FloatN x, _FloatN y); // M <= N
_DecimalN dMsubN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMsubN(_DecimalN x, _DecimalN y); // M <= N
_DecimalN dMxsubN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMxsubN(_DecimalN x, _DecimalN y); // M <= N

_FloatM fMulfN(_FloatN x, _FloatN y); // M < N
_FloatM fMulfN(_FloatN x, _FloatN y); // M <= N
_FloatM fMxmulfN(_FloatN x, _FloatN y); // M < N
_FloatM fMxmulfN(_FloatN x, _FloatN y); // M <= N
_DecimalN dMulfN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMulfN(_DecimalN x, _DecimalN y); // M <= N
_DecimalN dMxmulfN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMxmulfN(_DecimalN x, _DecimalN y); // M <= N

_FloatM fMdivfN(_FloatN x, _FloatN y); // M < N
_FloatM fMdivfN(_FloatN x, _FloatN y); // M <= N
_FloatM fMxdivfN(_FloatN x, _FloatN y); // M < N
_FloatM fMxdivfN(_FloatN x, _FloatN y); // M <= N
_DecimalN dMdivfN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMdivfN(_DecimalN x, _DecimalN y); // M <= N
_DecimalN dMxdivfN(_DecimalN x, _DecimalN y); // M < N
_DecimalN dMxdivfN(_DecimalN x, _DecimalN y); // M <= N

_FloatM fMsqrtfN(_FloatN x); // M < N
_FloatM fMsqrtfN(_FloatN x); // M <= N
_FloatM fMsqrtfN(_FloatN x); // M < N
_FloatM fMsqrtfN(_FloatN x); // M <= N
_DecimalN dMsqrtfN(_DecimalN x); // M < N
_DecimalN dMsqrtfN(_DecimalN x); // M <= N
_DecimalN dMxdivfN(_DecimalN x); // M < N
_DecimalN dMxdivfN(_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 fMxfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMxfmafN(_FloatN x, _FloatN y, _FloatN z); // M <= N
_DecimalN dMfmadN(_DecimalN x, _DecimalN y, _DecimalN z); // M < N
_DecimalN dMfmadN(_DecimalN x, _DecimalN y, _DecimalN z); // M <= N
// M <= N
_DecimalM x dMx_fma_N(_DecimalN x, _DecimalN y, _DecimalN z);
// M < N
_DecimalM x dMx_fma_Nx(_DecimalN x, _DecimalN y, _DecimalN z);
// M < N

F.10.12 Total order functions

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

F.10.13 Payload functions

_FloatN getpayloadfN(const _FloatN *x);
_FloatN getpayloadfNx(const _FloatN *x);
_DecimalN getpayloaddN(const _DecimalN *x);
_DecimalN getpayloaddNx(const _DecimalN *x);

int setpayloadfN(_FloatN *res, _FloatN pl);
int setpayloadfNx(_FloatN *res, _FloatN pl);
int setpayloaddN(_DecimalN *res, _DecimalN pl);
int setpayloaddNx(_DecimalN *res, _DecimalN pl);

int setpayloadsigfN(_FloatN *res, _FloatN pl);
int setpayloadsigfNx(_FloatN *res, _FloatN pl);
int setpayloadsigdN(_DecimalN *res, _DecimalN pl);
int setpayloadsigdNx(_DecimalN *res, _DecimalN 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

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT_
#include <math.h>
void encodefN(unsigned char * restrict encptr, const _FloatN * restrict xptr);
Description

[2] The `encodeN` functions convert `xptr` into an IEC 60559 binary `N` 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 `decodeN` functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
void decodefN (_FloatN * restrict xptr, const unsigned char * restrict encptr);

Description

[2] The `decodeN` 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 `dMencodedN` and `dMencodedN` 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.
7.12.11.7c.2 The dMencodedN and dMencodebN functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
void dMencodedN(unsigned char * restrict encMptr, const unsigned char * restrict encNptr);
void dMencodebN(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 dMencodedN functions convert between formats using the encoding scheme
based on decimal encoding of the significand. The dMencodebN 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 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.

12 Numeric conversion functions in <stdlib.h>

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

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

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

After 7.22.1#1, insert

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

After 7.22.1.2b, insert:

7.22.1.2c The strfromfN, strfromfNx, strfromdN, and strfromdNx functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#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);

Description

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

Returns

[3] The strfromfN and strfromfNx functions return values similar to the strfromd function. The strfromdN and strfromdNx functions return values similar to the 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 from 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 strtoencodecN and strtoencbindN 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

```c
#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 6059 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

```c
#define __STDC_WANT_IEC_6059_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 `strtfN` functions, except they store an IEC 6059 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

[1] An implementation shall declare the `strfromencfN` function for each `N` equal the width of a supported IEC 6059 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 6059 arithmetic or non-arithmetic decimal interchange format.

7.22.1.3d.1 The `strfromencfN` functions

Synopsis

```c
#define __STDC_WANT_IEC_6059_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 strfromencbindN(char * restrict s, size_t n, const char * restrict 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 decimalN 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 decimalN 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 \( \text{float, long double, } _{\text{float}}N, \) and \( _{\text{float}}Nx. \)

Add after 7.3.1#3:

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

- \(_{\text{float}}N \text{ complex } \text{cacosfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{cacosfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{casinfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{casinfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{catanfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{catanfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{ccosfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{ccosfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{casinfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{casinfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{ctanfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{ctanfNx}(\_\text{float}Nx \text{ complex } z);\)

### 7.3.6 Hyperbolic functions

- \(_{\text{float}}N \text{ complex } \text{cacosfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{cacosfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{casinhfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{casinhfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{catanhfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{catanhfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{ccoshfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{ccoshfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{casinhfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{casinhfNx}(\_\text{float}Nx \text{ complex } z);\)
- \(_{\text{float}}N \text{ complex } \text{ctanhfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{ctanhfNx}(\_\text{float}Nx \text{ complex } z);\)

### 7.3.7 Exponential and logarithmic functions

- \(_{\text{float}}N \text{ complex } \text{cexpfN}(\_\text{float}N \text{ complex } z);\)
- \(_{\text{float}}Nx \text{ complex } \text{cexpfNx}(\_\text{float}Nx \text{ complex } z);\)
7.3.8 Power and absolute value functions

```c
_FloatN complex clogfN(_FloatN complex z);
_FloatN complex clogNx(_FloatN complex z);
_FloatN complex cabsfN(_FloatN complex z);
_FloatN complex cabsNx(_FloatN complex z);
_FloatN complex cpowfN(_FloatN complex z, _FloatN complex y);
_FloatN complex cpowfNx(_FloatN complex z, _FloatN complex y);
_FloatN complex csqrtfN(_FloatN complex z);
_FloatN complex csqrtfNx(_FloatN complex z);
```

7.3.9 Manipulation functions

```c
_FloatN complex cargfN(_FloatN complex z);
_FloatN complex cargfNx(_FloatN complex z);
_FloatN complex cimagfN(_FloatN complex z);
_FloatN complex cimagfNx(_FloatN complex z);
_FloatN complex CMPLXfN(_FloatN x, _FloatN y);
_FloatN complex CMPLXfNx(_FloatN x, _FloatN y);
_FloatN complex conjfN(_FloatN complex z);
_FloatN complex conjNx(_FloatN complex z);
_FloatN complex cprojfN(_FloatN complex z);
_FloatN complex cprojNx(_FloatN complex z);
_FloatN complex crealfN(_FloatN complex z);
_FloatN complex crealfNx(_FloatN complex z);
```

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

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

— Otherwise, the type determined is float.

In the second bullet 7.25#3c, attach a footnote to the wording:

the type determined is the widest

where the footnote is:

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

In 7.25#6, replace:

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

with:

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

After 7.25#6c, add the paragraph:

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

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

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

```c
#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:

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
```

In 7.25\#7, append to the declarations:

```
#define __STDC_WANT_IEC_60559WithTypeExt__ >= 201
_Float32x f32x;
_Float64 f64;
_Float128 f128;
_Float64 complex f64c;
#endif
```

In 7.25\#7, append to the table:

| \cos(f64x) | \ccosf64x(f64xc) |
| \pow(dc, f128) | \cpowf128(dc, f128) |
| \fmax(f64, d) | \fmaxf64(f64, d) |
| \fmax(d, f32x) | \fmax(d, f32x), the function, if the set of values of \_Float32x is a subset of (or equivalent to) the set of values of \double, or \fmaxf32x(d, f32x), if the set of values of \double is a proper subset of the set of values of \_Float32x, or undefined, if neither of the sets of values of \double and \_Float32x is a subset of the other (and the sets are not equivalent) |
| \pow(f32x, n) | \powf32x(f32x, n) |
Bibliography


