

ISO/IEC JTC 1/SC 22/WG 14 N1680

Formatted: Font color: Light Blue

Date: yyyy-mm-dd

Reference number of document: ISO/IEC TS 18661

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

Secretariat: ANSI

5

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

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

Warning

15

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

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

Document type: Technical Specification
Document subtype:
Document stage: (20) Preparation
Document language: E

Copyright notice

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

Page

Introduction v

Background v

IEC 60559 floating-point standard v

5 C support for IEC 60559vi

Purposevii

More about formatsvii

1 Scope..... 1

2 Conformance 1

10 3 Normative references 1

4 Terms and definitions 2

5 C standard conformance 2

5.1 Freestanding implementations 2

5.2 Predefined macros 2

15 5.3 Standard headers 2

6 Types 3

7 Characteristics..... 5

8 Conversions..... 7

9 Constants 8

20 10 Expressions 9

11 Mathematics <math.h>..... 9

12 Numeric conversion functions <stdlib.h>.....18

Bibliography.....20

25

Foreword

5 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.

10 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.

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

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

— *Part 1: Binary floating-point arithmetic*

— *Part 2: Decimal floating-point arithmetic*

— *Part 3: Interchange and extended types*

20 — *Part 4: Supplemental functions*

— *Part 5: Supplemental attributes*

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

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

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

Introduction

Background

IEC 60559 floating-point standard

5 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:

- 10 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.
- 15 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
 - 20 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
 - 25 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:

- 30 • *formats* - for binary floating-point data, including representations for Not-a-Number (NaN) and signed infinities and zeros
- *operations* – basic arithmetic operations (addition, multiplication, etc.) on the format data to compose a well-defined, closed arithmetic system (It also specified conversions between floating-point formats and decimal character sequences, and a few auxiliary operations.)
- 35 • *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.

5 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.

10 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 inquiries 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.

15 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.

20 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.

25 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.

30 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.

35 **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.

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

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

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

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

Purpose

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

- 15 Part 1 provides suggested 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 suggested changes in Part 1.

20 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 (this document), 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.

25 More about 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)
- 30 • 64 and 128 bit basic decimal formats
- interchange formats, whose precision and range are determined by the width k , where
 - for binary, $k = 16, 32, 64,$ and $k \geq 128$ and a multiple of 32, and
 - for decimal, $k \geq 32$ and a multiple of 32
- extended formats, for each basic format, with minimum range and precision specified

35 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.

40 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.

45 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 this Technical Specification.

5 | 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.

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

15 | The C language provides just three "generic" floating types (`float`, `double`, and `long double`), which Annex F of the C Standard requires 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 interchange and extended formats that allow portable use of the formats as envisioned in IEC 60559. It covers these aspects of types with IEC 60559 interchange and extended formats:

- 20 | names
- characteristics
- conversions
- constants
- function suffixes
- 25 | character sequence conversion interfaces

30 | 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 interchange and extended formats and C generic floating types.

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

1 Scope

5 This document, Part 3 of Technical Specification 18661, extends programming language C to include nomenclature for types with the interchange and extended floating-point formats specified in ISO/IEC/IEEE 60559:2011.

10 This document proposes nomenclature for all applicable types in Parts 1 and 2 of Technical Specification 18661 and for any other types with IEC 60559 interchange or extended formats supported by the implementation.

2 Conformance

An implementation conforms to Part 3 of Technical Specification 18661 if

- a) It conforms for Part 1 or Part 2 (or both) of Technical Specification 18661;
- 15 b) It meets the requirements for a conforming implementation of C11 with all the suggested changes to C11 as specified in Part 3 of Technical Specification 18661; and
- c) It defines `__STDC_IEC_60559_TYPES__` to 201~~ymm~~L.

3 Normative references

20 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*

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

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

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

Suggested changes proposed by Part 3 of Technical Specification 18661 are relative to ISO/IEC 9899:2011 (C11). The actual specification is given by a synthesis with the suggested changes from Parts 1 or 2 or both, depending on which Parts the implementation supports.

35 **ISSUE 1: Will the approach of the previous paragraph lead to a specification that is intelligible? Is there a better approach?**

4 Terms and definitions

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

4.1

C11

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

5 C standard conformance

5.1 Freestanding implementations

The following suggested change to C11 expands the conformance requirements for freestanding implements so that they might conform to this Part of Technical Specification 18661

Suggested change to C11:

Replace the third sentence of 4#6:

A conforming freestanding implementation shall accept any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, and `<stdnoreturn.h>`.

with:

A conforming freestanding implementation shall accept any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers `<fenv.h>`, `<float.h>`, `<iso646.h>`, `<limits.h>`, `<math.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, and `<stdnoreturn.h>` and the numeric conversion functions (7.22.1) of the standard header `<stdlib.h>`.

5.2 Predefined macros

Suggested change to C11:

In 6.10.8.3#1, add:

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

5.3 Standard headers

The library functions, macros, and types defined in this Part of Technical Specification 18661 are defined by their respective headers if the macro `__STDC_WANT_IEC_18661_EXT3__` is defined at the point in the source file where the appropriate header is first included.

ISSUE 2: In Part 1 the content in 5.3 above is at the end of 5.1 - which should be fixed.

6 Types

This clause recommends changes to C11 to include the interchange and extended types specified in IEC 60559.

ISSUE 3: Do we need to specify nomenclature to support decimal extended types?

5 Suggested change to C11:

Change the first sentence of 6.2.5#10 from:

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

to:

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

10 After 6.2.5#10, insert:

[10a] 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 1 – Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary N ($N \geq 128$)
N , storage width in bits	16	32	64	128	multiple of 32
p , precision in bits	11	24	53	113	$N - \text{round}(4 \times \log_2(N)) + 13$
$emax$, maximum exponent e	15	127	1023	16383	$2^{(N-p-1)} - 1$
<i>Encoding parameters</i>					
$bias$, $E-e$	15	127	1023	16383	$emax$
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 `round()` in Table 1 rounds to the nearest integer.

For example, `binary256` would have $p = 237$ and $emax = 262143$.

Table 2 – Decimal interchange format parameters

Parameter	decimal32	decimal64	decimal128	decimal N ($N \geq 32$)
N , storage width in bits	32	64	128	multiple of 32
p , precision in digits	7	16	34	$9 \times N/32 - 2$
$emax$, maximum exponent e	96	384	6144	$3 \times 2^{(N/16 + 3)}$
<i>Encoding parameters</i>				
$bias$, $E - e$	101	398	6176	$emax + p - 2$
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 $emax = 1572864$.

Types designated

`_Float N` , where N is 16, 32, 64, or ≥ 128 and a multiple of 32

`_Decimal N` , where $N \geq 32$ and a multiple of 32

support the corresponding IEC 60559 interchange formats and are collectively called the *data-interchange types*. Each data-interchange type has the IEC 60559 interchange format corresponding to its width and radix. Data-interchange types that are supported by all applicable floating-point operations are collectively called the *interchange floating types*. An implementation that defines `_STDC_IEC_60559_BFP` shall provide `_Float32` and `_Float64` as interchange floating types equivalent to `float` and `double`, shall provide `_Float16` as a data-interchange type, and may provide `_Float16` as an interchange floating type. If the implementation's `long double` type has an IEC 60559 interchange format of width N , then the implementation shall provide the type `_Float N` as an interchange floating type. An implementation may provide any of the other data-interchange types and may provide any of them as interchange floating types.

[10b] For each of its basic formats, IEC 60559 specifies as 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 3 – Extended format parameters for floating-point numbers

	Extended formats associated with:

Parameter	binary32	binary64	binary128	decimal64	decimal128
p digits \geq	32	64	128	22	40
$emax \geq$	1023	16383	65535	6144	24576

Types designated `_Float32x`, `_Float64x`, `_Float128x`, `_Decimal64x`, and `_Decimal128x` support the corresponding IEC 60559 extended formats and are collectively called the *extended floating types*. An implementation that defines `__STDC_IEC_60559_BFP__` shall provide `_Float32x`, which may be equivalent to `double`, and may provide any of the other two binary extended floating types. An implementation that defines `__STDC_IEC_60559_DFP__` shall provide: `_Decimal64x`, which may be equivalent to `_Decimal128`, and may provide `_Decimal128x`.

[10c] The generic floating types, interchange floating types, and extended floating types are collectively called the *real floating types*.

10 Replace 6.2.5#11:

[11] There are three *complex types*, designated as `float _Complex`, `double _Complex`, and `long double _Complex`.43) (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*.

15 with:

[11] For the generic real types `float`, `double`, and `long double` and the interchange floating types `_FloatN`, there are *complex types* designated respectively as `float _Complex`, `double _Complex`, `long double _Complex`, and `_FloatN _Complex`. 43) (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*.

20 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.

25 with:

[2] For the generic real types `float`, `double`, and `long double` and the interchange floating types `_FloatN`, there are *imaginary types* designated respectively as `float _Imaginary`, `double _Imaginary`, `long double _Imaginary`, and `_FloatN _Imaginary`. The imaginary types (along with the real floating and complex types) are floating types.

30 7 Characteristics

This clause suggests new `<float.h>` macros, analogous to the macros for generic floating types, that characterize the data-interchange types and the extended floating types. It also suggests macros to indicate which data-interchange types are provided as interchange floating types.

Suggested change to C11:

35 Add the following after 5.2.4.2.2:

Macros in `<float.h>` provide characteristics of interchange types and extended floating types, as specified in IEC 60559, for all the data-interchange types and extended floating types supported by the implementation. The prefixes `FLT N _` and `DEC N _` are used for binary and decimal data-interchange types of width N . The prefixes `FLT N X_` and `DEC N X_` are used for binary and decimal extended floating types that extend a basic format of width N . For extended floating types, the type parameters p , e_{max} , and e_{min} are for the extended floating type itself, not for the basic format that it extends.

The macros

```

FLT $N$ _IS_ARITH
DEC $N$ _IS_ARITH

```

are defined if the implementation supports the data-interchange types `_Float N` and `_Decimal N` , respectively, as interchange floating types. If defined the macros have the integer value 1.

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

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

```

FLT $N$ _MANT_DIG
FLT $N$ X_MANT_DIG

```

— number of digits in the coefficient, p

```

DEC $N$ _MANT_DIG
DEC $N$ X_MANT_DIG

```

— number of decimal digits, n , such that any floating-point number with p binary digits can be rounded to a floating-point number with n decimal digits and back again without change to the value, $\text{ceiling}(1 + p \log_{10} 2)$

```

FLT $N$ _DECIMAL_DIG
FLT $N$ X_DECIMAL_DIG

```

— number of decimal digits, q , such that any floating-point number with q decimal digits can be rounded into a floating-point number with p binary digits and back again without change to the q decimal digits, $\text{floor}((p - 1) \log_{10} 2)$

```

FLT $N$ _DIG
FLT $N$ X_DIG

```

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

```

FLT $N$ _MIN_EXP
FLT $N$ X_MIN_EXP
DEC $N$ _MIN_EXP
DEC $N$ X_MIN_EXP

```

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

```

FLT $N$ _MIN_10_EXP
FLT $N$ X_MIN_10_EXP

```

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

5
`FLT_N_MAX_EXP`
`FLT_X_MAX_EXP`
`DEC_N_MAX_EXP`
`DEC_X_MAX_EXP`

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

10
`FLT_N_MAX_10_EXP`
`FLT_X_MAX_10_EXP`

The values given in the following list shall be replaced by constant expressions:

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

15
`FLT_N_MAX`
`FLT_X_MAX`
`DEC_N_MAX`
`DEC_X_MAX`

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

20
`FLT_N_EPSILON`
`FLT_X_EPSILON`
`DEC_N_EPSILON`
`DEC_X_EPSILON`

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

30
`FLT_N_MIN`
`FLT_X_MIN`
`DEC_N_MIN`
`DEC_X_MIN`

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

35
`FLT_N_TRUE_MIN`
`FLT_X_TRUE_MIN`
`DEC_N_TRUE_MIN`
`DEC_X_TRUE_MIN`

8 Conversions

- 40 The following suggested change to C11 supports the IEC 60559 restrictions against operands whose types do not have the same radix or such that neither type is a subset of (or equivalent to) the other.

Suggested change to C11:

In 6.3.1.8#1, replace the first 3 items after “This pattern is called the *usual arithmetic conversions*.” with:

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

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

Otherwise, if both operands are floating types and the sets of values of their corresponding real types are equivalent, no further conversion is needed.

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.

The following suggested change to C11 provides conversions between data-interchange types and other data-interchange types and real floating types.

After 6.3.2.3, add the subclause:

6.3.2.3a Data-interchange types

[1] Any supported data-interchange type can be converted to and from any supported data-interchange type and any real floating type, with rounding to IEC 60559 formats as specified in IEC 60559.

9 Constants

The following suggested changes to C11 provide suffixes that designate constants of data-interchange types and extended floating types.

Suggested changes to C11:

Change *floating-suffix* in 6.4.4.2 from:

floating-suffix: one of
f l F L

to:

floating-suffix: one of
f l F L
fN FN - for each supported binary data-interchange type `_FloatN`
fNx FNx - for each supported binary extended floating type `_FloatNx`
dN DN - for each supported decimal data-interchange type `_DecimalN`
dNx DNx - for each supported decimal extended floating type `_DecimalNx`

Add the following paragraph after 6.4.4.2#2:

Constraints

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

Add the following paragraph after 6.4.4.2#4:

[4a] If a floating constant is suffixed by 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`.

Add the following paragraph after 6.4.4.2#5:

[5a] Decimal floating-point constants that have the same numerical value but different quantum exponents have distinguishable internal representations. The quantum exponent is specified to be the same as for the corresponding `strtodN` or `strtodNx` function for the same numeric string.

5 10 Expressions

The following suggested change to C11 is intended to allow data-interchange types to be converted to and from real floating types and data-interchange types, by assignment, cast, argument passing, and function return.

Suggested changes to C11:

10 At the end of 6.5.16.1#1, append the bullet:

— the left operand has atomic, qualified, or unqualified real floating type or data-interchange type, and the right has real floating type or data-interchange type;

11 Mathematics <math.h>

15 This clause suggests changes to C11 to include functions and macros for interchange and extended floating types. Binary interchange floating types and binary extended floating types are supported by functions and macros corresponding to all those specified for generic floating types (`float`, `double`, and `long double`) in C11 and Part 1 of Technical Specification 18661. Decimal interchange floating types and decimal extended floating types are supported by functions and macros corresponding to all those for the decimal types in Part 2 of Technical Specification 18661. Data-interchange types (including ones that are not interchange floating types) are supported by the classification macros in C11 and Parts 1 and 2 of Technical Specification 18661, and by the `totalorder` and `totalordermag` functions in Parts 1 and 2.

25 The list of elementary functions specified in the mathematics library is extended to handle interchange floating types and extended floating types. These include functions specified in C11 (7.12.4, 7.12.5, 7.12.6, 7.12.7, 7.12.8, 7.12.9, 7.12.10, 7.12.11, 7.12.12, and 7.12.13) and in Part 1 of Technical Specification 18661 (14.1, 14.2, 14.3, 14.4, 14.5, 14.8, 14.9, and 14.0). Macros analogous to the `HUGE_VAL`, `INFINITY`, `NAN`, and the `SNAN` macros are defined for data-interchange types and extended floating types. With the exception of the floating-point functions listed in 11.2, which have accuracy as specified in IEC 60559, the accuracy of floating-point results is implementation-defined. The implementation may state that the accuracy is unknown. All comparison macros specified in C11 (7.12.14) and in Part 1 of Technical Specification 18661 (14.6) are extended to handle interchange floating types and extended floating types. All classification macros specified in C11 (7.12.3) and in Part 1 of Technical Specification 18661 (14.7) are extended to handle data-interchange types and extended floating types.

Suggested changes to C11:

In 7.12#1, change the second sentence from:

35 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:

40 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**, **_FloatNx**, **_DecimalN**, and **_DecimalNx**. The

implementation provides all the functions for its supported interchange floating types and extended floating types.

Add after 7.12 paragraph 2.

[2a] The types

```
5      dpdencodingdN_t
      bidencodingdN_t
```

represent values of decimal data-interchange types in one of the two alternative encodings required by the IEC 60559 standard: the Densely Packed Decimal (DPD) encoding or the Binary Integer Decimal (BID) encoding. These types are used by the decimal re-encoding functions (12.4.2).

ISSUE 4: Do we need types corresponding to `float_t` and `double_t` for the interchange floating types? Is this more appropriate for Part 5 which might deal with expression evaluation methods?

Add at the end of 7.12 paragraph 3 the following macros.

[3] ... For each supported data-interchange type `_FloatN` and `_DecimalN`, the macros

```
15      HUGE_VAL_FN
      HUGE_VAL_DN
```

expand to constant expressions of types `_FloatN` and `_DecimalN`, respectively, representing positive infinity. For each supported extended floating type `_FloatNx` and `_DecimalNx`, the macro

```
20      HUGE_VAL_DNx
      HUGE_VAL_DNx
```

expand to a constant expressions of types `_FloatNx` and `_DecimalNx`, respectively, representing positive infinity.

Add at the end of 7.12 paragraph 5 the following macros.

[5b] For each supported interface type `_FloatN` and `_DecimalN`, the signaling NaN macros

```
25      SNANFN
      SNANDN
```

expand into constant expressions of type `_FloatN` and `_DecimalN`, 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. For each supported extended floating type `_FloatNx` and `_DecimalNx`, the signaling NaN macros

```
30      SNANFNx
      SNANDNx
```

expand into constant expressions of type `_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.

Add at the end of 7.12 paragraph 7 the following macros.

[7] ... For each supported interface floating type `_FloatN` and `_DecimalN`, the macros

```
40      FP_FAST_FMAFN
      FP_FAST_FMADN
```

are, respectively, `_FloatN` and `_DecimalN` analogs of `FP_FAST_FMA`. For each supported extended floating type `_FloatNx` and `_DecimalNx`, the macros

```
5      FP_FAST_FMAFNx
      FP_FAST_FMADNx
```

are, respectively, `_FloatNx` and `_DecimalNx` analogs of `FP_FAST_FMA`.

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

7.12.4 Trigonometric functions

```
10      _FloatN acosfN (_FloatN x);
      _FloatNx acosfNx(_FloatNx x);
      _DecimalN acosdN (_DecimalN x);
      _DecimalNx acosdNx(_DecimalNx x);

15      _FloatN asinfN (_FloatN x);
      _FloatNx asinfNx(_FloatNx x);
      _DecimalN asindN (_DecimalN x);
      _DecimalNx asindNx(_DecimalNx x);

20      _FloatN atanfN (_FloatN x);
      _FloatNx atanfNx(_FloatNx x);
      _DecimalN atandN (_DecimalN x);
      _DecimalNx atandNx(_DecimalNx x);

25      _FloatN atan2fN (_FloatN y, _FloatN x);
      _FloatNx atan2fNx(_FloatNx y, _FloatNx x);
      _DecimalN atan2dN (_DecimalN y, _DecimalN x);
      _DecimalNx atan2dNx(_DecimalNx y, _DecimalNx x);

30      _FloatN cosfN (_FloatN x);
      _FloatNx cosfNx(_FloatNx x);
      _DecimalN cosdN (_DecimalN x);
      _DecimalNx cosdNx(_DecimalNx x);

35      _FloatN sinfN (_FloatN x);
      _FloatNx sinfNx(_FloatNx x);
      _DecimalN sindN (_DecimalN x);
      _DecimalNx sindNx(_DecimalNx x);

40      _FloatN tanfN (_FloatN x);
      _FloatNx tanfNx(_FloatNx x);
      _DecimalN tandN (_DecimalN x);
      _DecimalNx tandNx(_DecimalNx x);
```

45 7.12.5 Hyperbolic functions

```
50      _FloatN acoshfN (_FloatN x);
      _FloatNx acoshfNx(_FloatNx x);
      _DecimalN acoshdN (_DecimalN x);
      _DecimalNx acoshdNx(_DecimalNx x);

      _FloatN asinhfN (_FloatN x);
```

```

_FloatNx asinhfNx (_FloatNx x);
_DecimalN asinhdN (_DecimalN x);
_DecimalNx asinhdNx (_DecimalNx x);

```

```

5  _FloatN atanhfN (_FloatN x);
   _FloatNx atanhfNx (_FloatNx x);
   _DecimalN atanhdN (_DecimalN x);
   _DecimalNx atanhdNx (_DecimalNx x);

```

```

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

```

```

15  _FloatN sinhN (_FloatN x);
     _FloatNx sinhNx (_FloatNx x);
     _DecimalN sinhdN (_DecimalN x);
     _DecimalNx sinhdNx (_DecimalNx x);

```

```

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

```

7.12.6 Exponential and logarithmic functions

```

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

```

```

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

```

```

35  _FloatN expm1fN (_FloatN x);
     _FloatNx expm1fNx (_FloatNx x);
     _DecimalN expm1dN (_DecimalN x);
     _DecimalNx expm1dNx (_DecimalNx x);

```

```

40  _FloatN frexpfN (_FloatN value, int *exp);
     _FloatNx frexpfNx (_FloatNx value, int *exp);
     _DecimalN frexpdN (_DecimalN value, int *exp);
     _DecimalNx frexpdNx (_DecimalNx value, int *exp);

```

```

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

```

```

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

```

```

    _FloatN ldexpfN (_FloatN value, int exp);
    _FloatNx ldexpfN (_FloatNx value, int exp);
    _DecimalN ldexpdN (_DecimalN value, int exp);
    _DecimalNx ldexpdNx (_DecimalNx value, int exp);
5
    _FloatN logfN (_FloatN x);
    _FloatNx logfNx(_FloatNx x);
    _DecimalN logdN (_DecimalN x);
    _DecimalNx logdNx(_DecimalNx x);
10
    _FloatN log10fN (_FloatN x);
    _FloatNx log10fNx(_FloatNx x);
    _DecimalN log10dN (_DecimalN x);
    _DecimalNx log10dNx(_DecimalNx x);
15
    _FloatN log1pfN (_FloatN x);
    _FloatNx log1pfNx(_FloatNx x);
    _DecimalN log1pdN (_DecimalN x);
    _DecimalNx log1pdNx(_DecimalNx x);
20
    _FloatN log2fN (_FloatN x);
    _FloatNx log2fNx(_FloatNx x);
    _DecimalN log2dN (_DecimalN x);
    _DecimalNx log2dNx(_DecimalNx x);
25
    _FloatN logbfN (_FloatN x);
    _FloatNx logbfNx(_FloatNx x);
    _DecimalN logbdN (_DecimalN x);
    _DecimalNx logbdNx(_DecimalNx x);
30
    _FloatN modffN (_FloatN x, _FloatN *iptr);
    _FloatNx modffNx(_FloatNx x, _FloatNx *iptr);
    _DecimalN modfdN (_DecimalN x, _DecimalN *iptr);
    _DecimalNx modfdNx(_DecimalNx x, _DecimalNx *iptr);
35
    _FloatN scalbnfN (_FloatN value, int exp);
    _FloatNx scalbnfN (_FloatNx value, int exp);
    _DecimalN scalbndN (_DecimalN value, int exp);
    _DecimalNx scalbndNx (_DecimalNx value, int exp);
40
    _FloatN scalblnfN (_FloatN value, long int exp);
    _FloatNx scalblnfN (_FloatNx value, long int exp);
    _DecimalN scalblndN (_DecimalN value, long int exp);
    _DecimalNx scalblndNx (_DecimalNx value, long int exp);

```

45 7.12.7 Power and absolute-value functions

```

    _FloatN cbrtfN (_FloatN x);
    _FloatNx cbrtfNx(_FloatNx x);
    _DecimalN cbrtdN (_DecimalN x);
    _DecimalNx cbrtdNx(_DecimalNx x);
50
    _FloatN fabsfN (_FloatN x);
    _FloatNx fabsfNx(_FloatNx x);
    _DecimalN fabsdN (_DecimalN x);
    _DecimalNx fabsdNx(_DecimalNx x);

```

```

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

```

```

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

```

```

_FloatN sqrtfN (_FloatN x);
_FloatNx sqrtfNx(_FloatNx x);
_DecimalN sqrtedN (_DecimalN x);
_DecimalNx sqrtedNx(_DecimalNx x);

```

7.12.8 Error and gamma functions

```

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

```

```

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

```

```

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

```

```

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

```

7.12.9 Nearest integer functions

```

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

```

```

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

```

```

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

```

```

_FloatN rintfN (_FloatN x);

```

```

    _FloatNx rintfNx(_FloatNx x);
    _DecimalN rintdN (_DecimalN x);
    _DecimalNx rintdNx(_DecimalNx x);

5    long int lrintfN (_FloatN x);
    long int lrintfNx (_FloatNx x);
    long int lrintdN (_DecimalN x);
    long int lrintdNx (_DecimalNx x);

10   long long int llrintfN (_FloatN x);
    long long int llrintfNx (_FloatNx x);
    long long int llrintdN (_DecimalN x);
    long long int llrintdNx (_DecimalNx x);

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

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

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

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

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

40   intmax_t fromfpfN (_FloatN x, int round, unsigned int width);
    intmax_t fromfpfNx (_FloatNx x, int round, unsigned int width);
    intmax_t fromfpdN (_DecimalN x, int round, unsigned int width);
    intmax_t fromfpdNx (_DecimalNx x, int round, unsigned int width);
    uintmax_t ufromfpfN (_FloatN x, int round, unsigned int width);
45   uintmax_t ufromfpfNx (_FloatNx x, int round, unsigned int width);
    uintmax_t ufromfpdN (_DecimalN x, int round, unsigned int width);
    uintmax_t ufromfpdNx (_DecimalNx x, int round, unsigned int width);

50   intmax_t fromfpxfN (_FloatN x, int round, unsigned int width);
    intmax_t fromfpxfNx (_FloatNx x, int round, unsigned int width);
    intmax_t fromfpxdN (_DecimalN x, int round, unsigned int width);
    intmax_t fromfpxdNx (_DecimalNx x, int round, unsigned int width);
    uintmax_t ufromfpxfN (_FloatN x, int round, unsigned int width);
    uintmax_t ufromfpxfNx (_FloatNx x, int round, unsigned int width);
55   uintmax_t ufromfpxdN (_DecimalN x, int round, unsigned int width);
    uintmax_t ufromfpxdNx (_DecimalNx x, int round, unsigned int width);

```

7.12.10 Remainder functions

```

_FloatN fmodfN (_FloatN x, _FloatN y);
_FloatNx fmodfNx(_FloatNx x, _FloatNx y);
_DecimalN fmoddN (_DecimalN x, _DecimalN y);
_DecimalNx fmoddNx(_DecimalNx x, _DecimalNx y);

_FloatN remainderfN (_FloatN x, _FloatN y);
_FloatNx remainderfNx(_FloatNx x, _FloatNx y);
_DecimalN remainderdN (_DecimalN x, _DecimalN y);
_DecimalNx remainderdNx(_DecimalNx x, _DecimalNx y);

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

```

7.12.11 Manipulation functions

```

_FloatN copysignfN (_FloatN x, _FloatN y);
_FloatNx copysignfNx(_FloatNx x, _FloatNx y);
_DecimalN copysigndN (_DecimalN x, _DecimalN y);
_DecimalNx copysigndNx(_DecimalNx x, _DecimalNx y);

_FloatN nanfN (const char *tagp);
_FloatNx nanfNx (const char *tagp);
_DecimalN nandN (const char *tagp);
_DecimalNx nandNx (const char *tagp);

_FloatN nextafterfN (_FloatN x, _FloatN y);
_FloatNx nextafterfNx(_FloatNx x, _FloatNx y);
_DecimalN nextafterdN (_DecimalN x, _DecimalN y);
_DecimalNx nextafterdNx(_DecimalNx x, _DecimalNx y);

```

ISSUE 5: Do we need a version of nexttoward that could be used with interchange floating types?

```

_FloatN nextupfN (_FloatN x);
_FloatNx nextupfNx(_FloatNx x);
_DecimalN nextupdN (_DecimalN x);
_DecimalNx nextupdNx(_DecimalNx x);

_FloatN nextdownfN (_FloatN x);
_FloatNx nextdownfNx(_FloatNx x);
_DecimalN nextdowndN (_DecimalN x);
_DecimalNx nextdowndNx(_DecimalNx x);

_FloatN canonicalizefN (_FloatN x);
_FloatNx canonicalizefNx(_FloatNx x);
_DecimalN canonicalizedN (_DecimalN x);
_DecimalNx canonicalizedNx(_DecimalNx x);

```

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);

```

```

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

```

```

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

```

```

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

```

7.12.13 Floating multiply-add

```

20   _FloatN fmafN (_FloatN x, _FloatN y, _FloatN z);
    _FloatNx fmafNx (_FloatNx x, _FloatNx y, _FloatNx z);
    _DecimalN fmadN (_DecimalN x, _DecimalN y, _DecimalN z);
    _DecimalNx fmadNx (_DecimalNx x, _DecimalNx y, _DecimalNx z);

```

7.12.14 Functions that round result to narrower format

```

25   _FloatM fMaddfN (_FloatN x, _FloatN y); // M < N
    _FloatM fMaddfNx (_FloatNx x, _FloatNx y); // M <= N
    _FloatNx fMxaddfN (_FloatN x, _FloatN y); // M < N
    _FloatNx fMxaddfNx (_FloatNx x, _FloatNx y); // M < N
    _DecimalM dMadddN (_DecimalN x, _DecimalN y); // M < N
    _DecimalM dMadddNx (_DecimalNx x, _DecimalNx y); // M <= N
30   _DecimalNx dMxadddN (_DecimalN x, _DecimalN y); // M < N
    _DecimalNx dMxadddNx (_DecimalNx x, _DecimalNx y); // M < N

```

```

35   _FloatM fMsubfN (_FloatN x, _FloatN y); // M < N
    _FloatM fMsubfNx (_FloatNx x, _FloatNx y); // M <= N
    _FloatNx fMxsubfN (_FloatN x, _FloatN y); // M < N
    _FloatNx fMxsubfNx (_FloatNx x, _FloatNx y); // M < N
    _DecimalM dMsubdN (_DecimalN x, _DecimalN y); // M < N
    _DecimalM dMsubdNx (_DecimalNx x, _DecimalNx y); // M <= N
40   _DecimalNx dMxsubdN (_DecimalN x, _DecimalN y); // M < N
    _DecimalNx dMxsubdNx (_DecimalNx x, _DecimalNx y); // M < N

```

```

45   _FloatM fMmulfN (_FloatN x, _FloatN y); // M < N
    _FloatM fMmulfNx (_FloatNx x, _FloatNx y); // M <= N
    _FloatNx fMxmulfN (_FloatN x, _FloatN y); // M < N
    _FloatNx fMxmulfNx (_FloatNx x, _FloatNx y); // M < N
    _DecimalM dMmuldN (_DecimalN x, _DecimalN y); // M < N
    _DecimalM dMmuldNx (_DecimalNx x, _DecimalNx y); // M <= N
50   _DecimalNx dMxmuldN (_DecimalN x, _DecimalN y); // M < N
    _DecimalNx dMxmuldNx (_DecimalNx x, _DecimalNx y); // M < N

```

```

    _FloatM fMdivfN (_FloatN x, _FloatN y); // M < N
    _FloatM fMdivfNx (_FloatNx x, _FloatNx y); // M <= N

```

```

_FloatMx fMxdivfN (_FloatN x, _FloatN y); // M < N
_FloatMx fMxdivfNx (_FloatNx x, _FloatNx y); // M < N
_DecimalM dMdivdN (_DecimalN x, _DecimalN y); // M < N
_DecimalM dMdivdNx (_DecimalNx x, _DecimalNx y); // M <= N
5  _DecimalMx dMxdivdN (_DecimalN x, _DecimalN y); // M < N
   _DecimalMx dMxdivdNx (_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMsqrtfN (_FloatN x); // M < N
_FloatM fMsqrtfNx (_FloatNx x); // M <= N
10  _FloatMx fMxsqrtfN (_FloatN x); // M < N
   _FloatMx fMxsqrtfNx (_FloatNx x); // M < N
   _DecimalM dMsqrtfN (_DecimalN x); // M < N
   _DecimalM dMsqrtfNx (_DecimalNx x); // M <= N
   _DecimalMx dMxdivdN (_DecimalN x); // M < N
15  _DecimalMx dMxdivdNx (_DecimalNx x); // M < N

_FloatM fMfmafN (_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMfmafNx (_FloatNx x, _FloatNx y, _FloatNx z); // M <= N
_FloatMx fMxfmafN (_FloatN x, _FloatN y, _FloatN z); // M < N
20  _FloatMx fMxdivfNx (_FloatNx x, _FloatNx y, _FloatNx z); // M < N
   _DecimalM dMfmadN (_DecimalN x, _DecimalN y, _DecimalN z); // M < N
   _DecimalM dMdfmadNx (_DecimalNx x, _DecimalNx y, _DecimalNx z);
   // M <= N
   _DecimalMx dMxfmadN (_DecimalN x, _DecimalN y, _DecimalN z);
25  // M < N
   _DecimalMx dMxfmadNx (_DecimalNx x, _DecimalNx y, _DecimalNx z);
   // M < N

```

F.10.13 Payload functions

```

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

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

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

```

12 Numeric conversion functions <stdlib.h>

This clause specifies functions to convert between character sequences and the data-interchange types 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 new functions that perform conversions like `snprintf_s` in <stdio.h>.

Suggested changes to C11:

After 7.22.1.4, add:

7.22.1.5 The `strtofN`, `strtofNx`, `strtodN`, and `strtodNx` functions

Synopsis

```
[1] #define __STDC_WANT_IEC_00000_EXT3__
#include <stdlib.h>
5   _FloatN strtodN (const char * restrict nptr, char ** restrict
    endptr);
    _FloatNx strtodNx (const char * restrict nptr, char ** restrict
    endptr);
    _DecimalN strtodN (const char * restrict nptr, char ** restrict
10   endptr);
    _DecimalNx strtodNx (const char * restrict nptr, char ** restrict
    endptr);
```

Description

15 The `strtodN` and `strtodNx` 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, specified in Part 2 of Technical Specification 18661, except they convert to the types `_DecimalN` and `_DecimalNx` respectively.

Returns

20 The `strtodN` and `strtodNx` 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.

7.2.2.1.6 The `strfromfN`, `strfromfNx`, `strfromdN`, and `strfromdNx` functions

Synopsis

```
[1] #define __STDC_WANT_IEC_00000_EXT3__
25 #include <stdlib.h>
    int strfromfN (char * restrict s, rsize_t n, const char * restrict
    format, _FloatN fp);
    int strfromfNx(char * restrict s, rsize_t n, const char * restrict
    format, _FloatNx fp);
30 int strfromdN (char * restrict s, rsize_t n, const char * restrict
    format, _DecimalN fp);
    int strfromdNx(char * restrict s, rsize_t n, const char * restrict
    format, _DecimalNx fp);
```

Description

35 These functions are equivalent to `snprintf_s(s, n, format, fp)` (K.3.5.3.5), except the type is implied by the function suffix and the `format` string contains no length modifier.

Returns

These functions return the value that would be returned by `snprintf_s(s, n, format, fp)`.

Bibliography

- 5
- [1] ISO/IEC 9899:2011, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*
- [2] ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic*
- [3] 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*
- [4] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems, second edition*
- 10 [5] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*
- [6] IEEE 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*
- [7] IEEE 854-1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*