TECHNICAL
SPECIFICATION

ISO/IEC TS 18661-4a

WG 14 N2401

CFP Working Draft
For C2X integration
2019-06-23

Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —

Part 4:
Supplementary functions

Technologies de l'information — Langages de programmation, leurs environnements et interfaces du logiciel système — Extensions à virgule flottante pour C —

Partie 4: Fonctions supplémentaires

Reference number
ISO/IEC TS 18661-4:2015(E)

Published in Switzerland

Foreword

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

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

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

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

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

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments, and system software interfaces*.

ISO/IEC TS 18661 consists of the following parts, under the general title *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C*:

— *Part 1: Binary floating-point arithmetic*

— *Part 2: Decimal floating-point arithmetic*

— *Part 3: Interchange and extended types*

— *Part 4: Supplementary functions*

The following part is under preparation:

— *Part 5: Supplementary attributes*

ISO/IEC TS 18661-1 updates ISO/IEC 9899:2011, *Information technology — 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*.

ISO/IEC TS 18661-2 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*.

ISO/IEC TS 18661-3, ISO/IEC TS 18661-4, and ISO/IEC TS 18661-5 specify extensions to ISO/IEC 9899:2011 for features recommended in ISO/IEC/IEEE 60559:201

Introduction

Background

IEC 60559 floating-point standard

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

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

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

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

4 Provide direct support for

   a. Execution-time diagnosis of anomalies

   b. Smoother handling of exceptions

   c. Interval arithmetic at a reasonable cost

5 Provide for development of

   a. Standard elementary functions such as exp and cos

   b. Very high precision (multiword) arithmetic

   c. Coupling of numerical and symbolic algebraic computation

6 Enable rather than preclude further refinements and extensions.

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

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

— *operations* – basic arithmetic operations (addition, multiplication, etc.) on the format data to compose a well-defined, closed arithmetic system; also specified conversions between floating-point formats and decimal character sequences, and a few auxiliary operations

— *context* – status flags for detecting exceptional conditions (invalid operation, division by zero, overflow, underflow, and inexact) and controls for choosing different rounding methods

The ISO/IEC/IEEE 60559:2011 international standard is equivalent to the IEEE 754-2008 standard for floating-point arithmetic, which is a major revision to IEEE 754-1985.

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

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

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

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

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

The revised standard, like its predecessor, defines its model of floating-point arithmetic in the abstract. It neither defines the way in which operations are expressed (which might vary depending on the computer language or other interface being used), nor does it define the concrete representation (specific layout in storage, or in a processor's register, for example) of data or context, except that it does define specific encodings that are to be used for the exchange of floating-point data between different implementations that conform to the specification.

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

C support for IEC 60559

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

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

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

ISO/IEC 9899:2011 (C11) includes refinements to the C99 floating-point specification, though it is still based on IEC 60559:1989. C11 upgraded annex G from “informative” to “conditionally normative”.

ISO/IEC TR 24732:2009 introduced partial C support for the decimal floating-point arithmetic in ISO/IEC/IEEE 60559:2011. ISO/IEC 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 ISO/IEC/IEEE 60559:2011 decimal formats, though it does not include all of the operations required by ISO/IEC/IEEE 60559:2011.

Purpose

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

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

ISO/IEC TS 18661-2 enhances ISO/IEC TR 24732 to cover all the requirements, plus some basic recommendations, of ISO/IEC/IEEE 60559:2011 for decimal floating-point arithmetic. C implementations intending to provide an extension for decimal floating-point arithmetic supporting ISO/IEC/IEEE 60559:2011 are expected to conform to ISO/IEC TS 18661-2.

ISO/IEC TS 18661-3 (Interchange and extended types), ISO/IEC TS 18661-4 (Supplementary functions), and ISO/IEC TS 18661-5 (Supplementary attributes) cover recommended features of ISO/IEC/IEEE 60559:2011. C implementations intending to provide extensions for these features are expected to conform to the corresponding parts.

Additional background on supplementary functions

This document uses the term supplementary functions to refer to functions that provide operations recommended, but not required, by IEC 60559.

ISO/IEC/IEEE 60559:2011 specifies and recommends a more extensive set of mathematical operations than C11 provides. The IEC 60559 specification is generally consistent with C11, though it adds requirements for symmetry and antisymmetry. This part of ISO/IEC TS 18661 extends the specification in Library subclause 7.12 Mathematics to include the complete set of IEC 60559 mathematical operations. For implementations conforming to annex F, it also requires full IEC 60559 semantics, including symmetry and antisymmetry properties.

IEC 60559 requires correct rounding for its required operations (squareRoot, fusedMultiplyAdd, etc.), and recommends correct rounding for its recommended mathematical operations. This part of ISO/IEC TS 18661 reserves identifiers, with **`cr`** prefixes, for C functions corresponding to correctly rounded versions of the IEC 60559 mathematical operations, which may be provided at the option of the implementation. For example, the identifier **`crexp`** is reserved for a correctly rounded version of the **`exp`** function.

IEC 60559 also specifies and recommends reduction operations, which operate on vector operands. These operations, which compute sums and products, may associate in any order and may evaluate in any wider format. Hence, unlike other IEC 60559 operations, they do not have unique specified results. This part of ISO/IEC TS 18661 extends the specification in Library subclause 7.12 Mathematics to include functions corresponding to the IEC 60559 reduction operations. For implementations conforming to annex F, it also requires the IEC 60559 specified behavior for floating-point exceptions.

Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —

Part 4: **Supplementary functions**

1 Scope

This part of ISO/IEC TS 18661 extends programming language C to include functions specified and recommended in ISO/IEC/IEEE 60559:2011.

2 Conformance

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

a) it meets the requirements for a conforming implementation of C11 with all the changes to C11 as specified in parts 1-4 of ISO/IEC TS 18661;

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

c) it defines **`__STDC_IEC_60559_FUNCS__`** to **`201506L`**.

3 Normative references

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

ISO/IEC 9899:2011, *Information technology — Programming languages — C*

ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic*

ISO/IEC TS 18661-1:2014, *Information technology — Programming languages, their environments and system software interfaces — Floating-point extensions for C — Part 1: Binary floating-point arithmetic*

ISO/IEC TS 18661-2:2015, *Information technology — Programming languages, their environments and system software interfaces — Floating-point extensions for C — Part 2: Decimal floating-point arithmetic*

ISO/IEC TS 18661-3:2015, *Information technology — Programming languages, their environments and system software interfaces — Floating-point extensions for C — Part 3: Interchange and extended types*

4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2011, ISO/IEC/IEEE 60559:2011, ISO/IEC TS 18661-1:2014, ISO/IEC TS 18661-2:2015, ISO/IEC TS 18661-3:2015, and the following apply.

4.1

C11
standard ISO/IEC 9899:2011, *Information technology — Programming languages C*, including *Technical Corrigendum 1* (ISO/IEC 9899:2011/Cor. 1:2012)

5 C standard conformance

5.1 Freestanding implementations

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

5.2 Predefined macros

6 Operation binding

The following change to C2X-20190607 shows how functions in C11 and in this Part of Technical Specification 18661 provide operations recommended in IEC 60559.

Change to C2X-20190607:

After F.3#22, add:

[23] The C functions in the following table provide operations recommended by IEC 60559 and similar operations. Correct rounding, which IEC 60559 specifies for its operations, is not required for the C functions in the table. See also 7.31.8.

IEC 60559 operation	C function	Clause
exp	**exp**	7.12.6.1, F.10.3.1
expm1	**expm1**	7.12.6.3, F.10.3.3
exp2	**exp2**	7.12.6.2, F.10.3.2
exp2m1	**exp2m1**	7.12.6.14, F.10.3.14
exp10	**exp10**	7.12.6.15, F.10.3.15
exp10m1	**exp10m1**	7.12.6.16, F.10.3.16
log	**log**	7.12.6.7, F.10.3.7
log2	**log2**	7.12.6.10, F.10.3.10
log10	**log10**	7.12.6.8, F.10.3.8
logp1	**log1p**, **logp1**	7.12.6.9, F.10.3.9
log2p1	**log2p1**	7.12.6.17, F.10.3.17
log10p1	**log10p1**	7.12.6.18, F.10.3.18
hypot	**hypot**	7.12.7.3, F.10.4.3
rSqrt	**rsqrt**	7.12.7.6, F.10.4.6
compound	**compoundn**	7.12.7.7, F.10.4.7
rootn	**rootn**	7.12.7.8, F.10.4.8
pown	**pown**	7.12.7.9, F.10.4.9
pow	**pow**	7.12.7.4, F.10.4.4

powr	`powr`	7.12.7.10, F.10.4.10
sin	`sin`	7.12.4.6, F.10.1.6
cos	`cos`	7.12.4.5, F.10.1.5
tan	`tan`	7.12.4.7, F.10.1.7
sinPi	`sinpi`	7.12.4.13, F.10.1.13
cosPi	`cospi`	7.12.4.12, F.10.1.12
	`tanpi`	7.12.4.14, F.10.1.14
	`asinpi`	7.12.4.9, F.10.1.9
	`acospi`	7.12.4.8, F.10.1.8
atanPi	`atanpi`	7.12.4.10, F.10.1.10
atan2Pi	`atan2pi`	7.12.4.11, F.10.1.11
asin	`asin`	7.12.4.2, F.10.1.2
acos	`acos`	7.12.4.1, F.10.1.1
atan	`atan`	7.12.4.3, F.10.1.3
atan2	`atan2`	7.12.4.4, F.10.1.4
sinh	`sinh`	7.12.5.5, F.10.2.5
cosh	`cosh`	7.12.5.4, F.10.2.4
tanh	`tanh`	7.12.5.6, F.10.2.6
asinh	`asinh`	7.12.5.2, F.10.2.2
acosh	`acosh`	7.12.5.1, F.10.2.1
atanh	`atanh`	7.12.5.3, F.10.2.3

7 Mathematical functions in `<math.h>`

This clause specifies changes to C2X-20190607 to include functions that support mathematical operations recommended by IEC 60559. The changes reserve names for correctly rounded versions of the functions. IEC 60559 recommends support for the correctly rounded functions. The changes also support the symmetry and antisymmetry properties that IEC 60559 specifies for mathematical functions.

Changes to C2X-20190607:

After 7.12.4.7, insert the following:

7.12.4.8 The `acospi` functions

Synopsis

[1]
```
#include <math.h>
double acospi(double x);
float acospif(float x);
long double acospil(long double x);
_Decimal32 acospid32(_Decimal32 x);
_Decimal64 acospid64(_Decimal64 x);
_Decimal128 acospid128(_Decimal128 x);
```

Description

[2] The **acospi** functions compute the principal value of the arc cosine of **x**, divided by π, thus measuring the angle in half-revolutions. A domain error occurs for arguments not in the interval [−1, +1].

Returns

[3] The **acospi** functions return arccos(**x**) / π, in the interval [0, 1].

7.12.4.9 The asinpi functions

Synopsis

[1]
```
#include <math.h>
double asinpi(double x);
float asinpif(float x);
long double asinpil(long double x);
_Decimal32 asinpid32(_Decimal32 x);
_Decimal64 asinpid64(_Decimal64 x);
_Decimal128 asinpid128(_Decimal128 x);
```

Description

[2] The **asinpi** functions compute the principal value of the arc sine of **x**, divided by π, thus measuring the angle in half-revolutions. A domain error occurs for arguments not in the interval [−1, +1]. A range error occurs if the magnitude of nonzero **x** is too small.

Returns

[3] The **asinpi** functions return arcsin(**x**) / π, in the interval [−1/2, +1/2].

7.12.4.10 The atanpi functions

Synopsis

[1]
```
#include <math.h>
double atanpi(double x);
float atanpif(float x);
long double atanpil(long double x);
_Decimal32 atanpid32(_Decimal32 x);
_Decimal64 atanpid64(_Decimal64 x);
_Decimal128 atanpid128(_Decimal128 x);
```

Description

[2] The **atanpi** functions compute the principal value of the arc tangent of **x**, divided by π, thus measuring the angle in half-revolutions. A range error occurs if the magnitude of nonzero **x** is too small.

Returns

[3] The **atanpi** functions return arctan(**x**) / π, in the interval [−1/2, +1/2].

7.12.4.11 The atan2pi functions

Synopsis

[1]
```
#include <math.h>
double atan2pi(double y, double x);
float atan2pif(float y, float x);
long double atan2pil(long double y, long double x);
_Decimal32 atan2pid32(_Decimal32 y, _Decimal32 x);
_Decimal64 atan2pid64(_Decimal64 y, _Decimal64 x);
_Decimal128 atan2pid128(_Decimal128 y, _Decimal128 x);
```

Description

[2] The **atan2pi** functions compute the angle, measured in half-revolutions, subtended at the origin by the point (**x**, **y**) and the positive *x*-axis. Thus, **atan2pi** computes arctan(**y**/**x**) / π, in the range [−1, +1]. A domain error may occur if both arguments are zero. A range error occurs if **x** is positive and the magnitude of nonzero **y**/**x** is too small.

Returns

[3] The **atan2pi** functions return the computed angle, in the interval [−1, +1].

7.12.4.12 The cospi functions

Synopsis

[1]
```
#include <math.h>
double cospi(double x);
float cospif(float x);
long double cospil(long double x);
_Decimal32 cospid32(_Decimal32 x);
_Decimal64 cospid64(_Decimal64 x);
_Decimal128 cospid128(_Decimal128 x);
```

Description

[2] The **cospi** functions compute the cosine of $\pi \times$ **x**, thus regarding **x** as a measurement in half-revolutions.

Returns

[3] The **cospi** functions return cos($\pi \times$ **x**).

7.12.4.13 The `sinpi` functions

Synopsis

[1]
```
#include <math.h>
double sinpi(double x);
float sinpif(float x);
long double sinpil(long double x);
_Decimal32 sinpid32(_Decimal32 x);
_Decimal64 sinpid64(_Decimal64 x);
_Decimal128 sinpid128(_Decimal128 x);
```

Description

[2] The **`sinpi`** functions compute the sine of $\pi \times$ **`x`**, thus regarding **`x`** as a measurement in half-revolutions.

Returns

[3] The **`sinpi`** functions return sin($\pi \times$ **`x`**).

7.12.4.14 The `tanpi` functions

Synopsis

[1]
```
#include <math.h>
double tanpi(double x);
float tanpif(float x);
long double tanpil(long double x);
_Decimal32 tanpid32(_Decimal32 x);
_Decimal64 tanpid64(_Decimal64 x);
_Decimal128 tanpid128(_Decimal128 x);
```

Description

[2] The **`tanpi`** functions compute the tangent of $\pi \times$ **`x`**, thus regarding **`x`** as a measurement in half-revolutions. A pole error may occur for arguments $n + 1/2$, for integers n.

Returns

[3] The **`tanpi`** functions return tan($\pi \times$ **`x`**).

In 7.12.6.10, replace the subclause title:

7.12.6.10 The `log1p` functions

with:

7.12.6.10 The `log1p` and `logp1` functions

In 7.12.6.10#1, append to the Synopsis:

```
double logp1(double x);
float logp1f(float x);
long double logp1l(long double x);
_Decimal32 logp1d32(_Decimal32 x);
_Decimal64 logp1d64(_Decimal64 x);
_Decimal128 logp1d128(_Decimal128 x);
```

In 7.12.6.10#2, replace the first sentence:

> The **log1p** functions compute the base-*e* (natural) logarithm of 1 plus the argument.

with:

> The **log1p** functions are equivalent to the **logp1** functions. These functions compute the base-*e* (natural) logarithm of 1 plus the argument.

Replace 7.12.6.10#3:

> [3] The **log1p** functions return $\log_e(1 + \mathbf{x})$.

with:

> [3] The **log1p** and **logp1** functions return $\log_e(1 + \mathbf{x})$.

After 7.12.6.2, insert the following:

7.12.6.2a The exp2m1 functions

Synopsis

[1]
```
#include <math.h>
double exp2m1(double x);
float exp2m1f(float x);
long double exp2m1l(long double x);
_Decimal32 exp2m1d32(_Decimal32 x);
_Decimal64 exp2m1d64(_Decimal64 x);
_Decimal128 exp2m1d128(_Decimal128 x);
```

Description

[2] The **exp2m1** functions compute the base-2 exponential of the argument, minus 1. A range error occurs if finite **x** is too large or if the magnitude of nonzero **x** is too small.

Returns

[3] The **exp2m1** functions return $2^{\mathbf{x}} - 1$.

After 7.12.6.1, insert the following:

7.12.6.1a The `exp10` functions

Synopsis

[1]
```
#include <math.h>
double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);
_Decimal32 exp10d32(_Decimal32 x);
_Decimal64 exp10d64(_Decimal64 x);
_Decimal128 exp10d128(_Decimal128 x);
```

Description

[2] The **`exp10`** functions compute the base-10 exponential of the argument. A range error occurs if the magnitude of finite **`x`** is too large.

Returns

[3] The **`exp10`** functions return 10^x.

7.12.6.1b The `exp10m1` functions

Synopsis

[1]
```
#include <math.h>
double exp10m1(double x);
float exp10m1f(float x);
long double exp10m1l(long double x);
_Decimal32 exp10m1d32(_Decimal32 x);
_Decimal64 exp10m1d64(_Decimal64 x);
_Decimal128 exp10m1d128(_Decimal128 x);
```

Description

[2] The **`exp10m1`** functions compute the base-10 exponential of the argument, minus 1. A range error occurs if finite **`x`** is too large or if the magnitude of nonzero **`x`** is too small.

Returns

[3] The **`exp10m1`** functions return $10^x - 1$.

After 7.12.6.11, insert the following:

7.12.6.11a The `log2p1` functions

Synopsis

[1]
```
#include <math.h>
double log2p1(double x);
float log2p1f(float x);
long double log2p1l(long double x);
_Decimal32 log2p1d32(_Decimal32 x);
_Decimal64 log2p1d64(_Decimal64 x);
_Decimal128 log2p1d128(_Decimal128 x);
```

Description

[2] The **`log2p1`** functions compute the base-2 logarithm of 1 plus the argument. A domain error occurs if the argument is less than −1. A pole error may occur if the argument equals −1.

Returns

[3] The **`log2p1`** functions return $\log_2(1 + \mathbf{x})$.

After 7.12.6.9, insert the following:

7.12.6.9a The `log10p1` functions

Synopsis

[1]
```
#include <math.h>
double log10p1(double x);
float log10p1f(float x);
long double log10p1l(long double x);
_Decimal32 log10p1d32(_Decimal32 x);
_Decimal64 log10p1d64(_Decimal64 x);
_Decimal128 log10p1d128(_Decimal128 x);
```

Description

[2] The **`log10p1`** functions compute the base-10 logarithm of 1 plus the argument. A domain error occurs if the argument is less than −1. A pole error may occur if the argument equals −1. A range error occurs if the magnitude of nonzero **`x`** is too small.

Returns

[3] The **`log10p1`** functions return $\log_{10}(1 + \mathbf{x})$.

After 7.12.7.1, insert the following:

7.12.7.1a The compoundn functions

Synopsis

[1]
```
#include <math.h>
#include <stdint.h>
double compoundn(double x, intmax_t n);
float compoundnf(float x, intmax_t n);
long double compoundnl(long double x, intmax_t n);
_Decimal32 compoundnd32(_Decimal32 x, intmax_t n);
_Decimal64 compoundnd64(_Decimal64 x, intmax_t n);
_Decimal128 compoundnd128(_Decimal128 x, intmax_t n);
```

Description

[2] The **compoundn** functions compute 1 plus **x**, raised to the power **n**. A domain error occurs if **x** < −1. A range error may occur if **n** is too large, depending on **x**. A pole error may occur if **x** equals −1 and **n** < 0.

Returns

[3] The functions return $(1 + \mathbf{x})^{\mathbf{n}}$.

After 7.12.7.4, insert the following:

7.12.7.4a The pown functions

Synopsis

[1]
```
#include <math.h>
#include <stdint.h>
double pown(double x, intmax_t n);
float pownf(float x, intmax_t n);
long double pownl(long double x, intmax_t n);
_Decimal32 pownd32(_Decimal32 x, intmax_t n);
_Decimal64 pownd64(_Decimal64 x, intmax_t n);
_Decimal128 pownd128(_Decimal128 x, intmax_t n);
```

Description

[2] The **pown** functions compute **x** raised to the **n**th power. A range error may occur. A pole error may occur if **x** equals zero and **n** < 0.

Returns

[3] The **pown** functions return $\mathbf{x}^{\mathbf{n}}$.

7.12.7.4b The `powr` functions

Synopsis

[1]
```
#include <math.h>
double powr(double x, double y);
float powrf(float x, float y);
long double powrl(long double x, long double y);
_Decimal32 powrd32(_Decimal32 x, _Decimal32 y);
_Decimal64 powrd64(_Decimal64 x, _Decimal64 y);
_Decimal128 powrd128(_Decimal128 x, _Decimal128 y);
```

Description

[2] The **`powr`** functions compute **`x`** raised to the power **`y`** as exp(**`y`** × log(**`x`**)). A domain error occurs if **`x`** < 0 or if **`x`** and **`y`** are both zero. A range error may occur. A pole error may occur if **`x`** equals zero and finite **`y`** < 0.

Returns

[3] The **`powr`** functions return $\mathbf{x}^{\mathbf{y}}$.

7.12.7.4c The `rootn` functions

Synopsis

[1]
```
#include <math.h>
#include <stdint.h>
double rootn(double x, intmax_t n);
float rootnf(float x, intmax_t n);
long double rootnl(long double x, intmax_t n);
_Decimal32 rootnd32(_Decimal32 x, intmax_t n);
_Decimal64 rootnd64(_Decimal64 x, intmax_t n);
_Decimal128 rootnd128(_Decimal128 x, intmax_t n);
```

Description

[2] The **`rootn`** functions compute the principal **`n`**th root of **`x`**. A domain error occurs if **`n`** is 0 or if **`x`** < 0 and **`n`** is even. A range error may occur if **`n`** is −1. A pole error may occur if **`x`** equals zero and **`n`** < 0.

Returns

[3] The **`rootn`** functions return $\mathbf{x}^{1/\mathbf{n}}$.

7.12.7.4d The `rsqrt` functions

Synopsis

[1]
```
#include <math.h>
double rsqrt(double x);
float rsqrtf(float x);
long double rsqrtl(long double x);
_Decimal32 rsqrtd32(_Decimal32 x);
_Decimal64 rsqrtd64(_Decimal64 x);
_Decimal128 rsqrtd128(_Decimal128 x);
```

Description

[2] The **`rsqrt`** functions compute the reciprocal of the square root of the argument. A domain error occurs if the argument is less than zero. A pole error may occur if the argument equals zero.

Returns

[3] The **`rsqrt`** functions return 1 / √**`x`**.

After 7.31.8#3, insert:

[4] The function names

`crexp`	**`crrsqrt`**	**`cracospi`**
`crexpm1`	**`crcompoundn`**	**`cratanpi`**
`crexp2`	**`crrootn`**	**`cratan2pi`**
`crexp2m1`	**`crpown`**	**`crasin`**
`crexp10`	**`crpow`**	**`cracos`**
`crexp10m1`	**`crpowr`**	**`cratan`**
`crlog`	**`crsin`**	**`cratan2`**
`crlog2`	**`crcos`**	**`crsinh`**
`crlog10`	**`crtan`**	**`crcosh`**
`crlog1p`	**`crsinpi`**	**`crtanh`**
`crlogp1`	**`crcospi`**	**`crasinh`**
`crlog2p1`	**`crtanpi`**	**`cracosh`**
`crlog10p1`	**`crasinpi`**	**`cratanh`**
`crhypot`		

and the same names suffixed with **`f`**, **`l`**, **`d32`**, **`d64`**, or **`d128`** may be added to the **`<math.h>`** header. The **`cr`** prefix is intended to indicate a correctly rounded version of the function.

After F.10#2, insert:

[2a] For each single-argument function *f* in **`<math.h>`** whose mathematical counterpart is symmetric (even), $f(x)$ is $f(-x)$ for all rounding modes and for all x in the (valid) domain of the function. For each single-argument function *f* in **`<math.h>`** whose mathematical counterpart is antisymmetric (odd), $f(-x)$ is $-f(x)$ for the IEC 60559 rounding modes roundTiesToEven, roundTiesToAway, and roundTowardZero, and for all x in the (valid) domain of the function. The **`atan2`** and **`atan2pi`** functions are odd in their first argument.

After F.10.1.7, insert the following:

F.10.1.8 The `acospi` functions

— `acospi(+1)` returns +0.
— `acospi(`x`)` returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.9 The `asinpi` functions

— `asinpi(`±0`)` returns ±0.
— `asinpi(`x`)` returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.10 The `atanpi` functions

— `atanpi(`±0`)` returns ±0.
— `atanpi(`±∞`)` returns ±1/2.

F.10.1.11 The `atan2pi` functions

— `atan2pi(`±0, −0`)` returns ±1.
— `atan2pi(`±0, +0`)` returns ±0.
— `atan2pi(`±0, x`)` returns ±1 for $x < 0$.
— `atan2pi(`±0, x`)` returns ±0 for $x > 0$.
— `atan2pi(`y, ±0`)` returns −1/2 for $y < 0$.
— `atan2pi(`y, ±0`)` returns +1/2 for $y > 0$.
— `atan2pi(`±y, −∞`)` returns ±1 for finite $y > 0$.
— `atan2pi(`±y, +∞`)` returns ±0 for finite $y > 0$.
— `atan2pi(`±∞, x`)` returns ±1/2 for finite x.
— `atan2pi(`±∞, −∞`)` returns ±3/4.
— `atan2pi(`±∞, +∞`)` returns ±1/4.

F.10.1.12 The `cospi` functions

— `cospi(`±0`)` returns 1.
— `cospi(`$n + 1/2$`)` returns +0, for integers n.
— `cospi(`±∞`)` returns a NaN and raises the "invalid" floating-point exception.

F.10.1.13 The `sinpi` functions

— `sinpi(`±0`)` returns ±0.
— `sinpi(`±n`)` returns ±0, for positive integers n.
— `sinpi(`±∞`)` returns a NaN and raises the "invalid" floating-point exception.

F.10.1.14 The `tanpi` functions

— `tanpi(`±0`)` returns ±0.
— `tanpi(`n`)` returns +0, for positive even and negative odd integers n.
— `tanpi(`n`)` returns −0, for positive odd and negative even integers n.
— `tanpi(`$n + 1/2$`)` returns +∞ and raises the "divide-by-zero" floating-point exception, for even integers n.

— **`tanpi`**`(`$n + 1/2$`)` returns −∞ and raises the "divide-by-zero" floating-point exception, for odd integers n.
— **`tanpi`**`(±∞)` returns a NaN and raises the "invalid" floating-point exception.

After F.10.3.2, insert the following:

F.10.3.2a The `exp2m1` functions

— **`exp2m1`**`(±0)` returns ±0.
— **`exp2m1`**`(−∞)` returns −1.
— **`exp2m1`**`(+∞)` returns +∞.

After F.10.3.1, insert the following:

F.10.3.1a The `exp10` functions

— **`exp10`**`(±0)` returns 1.
— **`exp10`**`(−∞)` returns +0.
— **`exp10`**`(+∞)` returns +∞.

F.10.3.1b The `exp10m1` functions

— **`exp10m1`**`(±0)` returns ±0.
— **`exp10m1`**`(−∞)` returns −1.
— **`exp10m1`**`(+∞)` returns +∞.

In F.10.3.10, replace the subclause title:

F.10.3.10 The `log1p` functions

with:

F.10.3.10 The `log1p` and `logp1` functions

In F.10.3.10, in each bullet, replace log1p with logp1, and, after the bullets, append:

The **`log1p`** functions are equivalent to the **`logp1`** functions.

After F.10.3.11, insert the following:

F.10.3.11a The `log2p1` functions

— **`log2p1`**`(±0)` returns ±0.
— **`log2p1`**`(−1)` returns −∞ and raises the "divide-by-zero" floating-point exception.
— **`log2p1`**`(`x`)` returns a NaN and raises the "invalid" floating-point exception for $x < -1$.
— **`log2p1`**`(+∞)` returns +∞.

After F.10.3.9, insert the following:

F.10.3.9a The `log10p1` functions

— **`log10p1`**`(±0)` returns ±0.

— **`log10p1(−1)`** returns −∞ and raises the “divide-by-zero” floating-point exception.
— **`log10p1(`**x**`)`** returns a NaN and raises the “invalid” floating-point exception for $x < -1$.
— **`log10p1(+∞)`** returns +∞.

After F.10.4.4, insert the following:

F.10.4.4a The `pown` functions

— **`pown(`**x**`, 0)`** returns 1 for all x not a signaling NaN.
— **`pown(±0, `**n**`)`** returns ±∞ and raises the “divide-by-zero” floating-point exception for odd $n < 0$.
— **`pown(±0, `**n**`)`** returns +∞ and raises the “divide-by-zero” floating-point exception for even $n < 0$.
— **`pown(±0, `**n**`)`** returns +0 for even $n > 0$.
— **`pown(±0, `**n**`)`** returns ±0 for odd $n > 0$.
— **`pown(±∞, `**n**`)`** is equivalent to **`pown(±0, `**$-n$**`)`** for n not 0, except that the “divide-by-zero” floating-point exception is not raised.

F.10.4.4b The `powr` functions

— **`powr(`**x**`, ±0)`** returns 1 for finite $x > 0$.
— **`powr(±0, `**y**`)`** returns +∞ and raises the “divide-by-zero” floating-point exception for finite $y < 0$.
— **`powr(±0, −∞)`** returns +∞.
— **`powr(±0, `**y**`)`** returns +0 for $y > 0$.
— **`powr(+1, `**y**`)`** returns 1 for finite y.
— **`powr(`**x**`, `**y**`)`** returns a NaN and raises the “invalid” floating-point exception for $x < 0$.
— **`powr(±0, ±0)`** returns a NaN and raises the “invalid” floating-point exception.
— **`powr(+∞, ±0)`** returns a NaN and raises the “invalid” floating-point exception.
— **`powr(1, ±∞)`** returns a NaN and raises the “invalid” floating-point exception.

F.10.4.4c The `rootn` functions

— **`rootn(±0, `**n**`)`** returns ±∞ and raises the “divide-by-zero” floating-point exception for odd $n < 0$.
— **`rootn(±0, `**n**`)`** returns +∞ and raises the “divide-by-zero” floating-point exception for even $n < 0$.
— **`rootn(±0, `**n**`)`** returns +0 for even $n > 0$.
— **`rootn(±0, `**n**`)`** returns ±0 for odd $n > 0$.
— **`rootn(+∞, `**n**`)`** returns +∞ for $n > 0$.
— **`rootn(−∞, `**n**`)`** returns −∞ for odd $n > 0$.
— **`rootn(−∞, `**n**`)`** returns a NaN and raises the “invalid” floating-point exception for even $n > 0$.
— **`rootn(+∞, `**n**`)`** returns +0 for $n < 0$.
— **`rootn(−∞, `**n**`)`** returns −0 for odd $n < 0$.
— **`rootn(−∞, `**n**`)`** returns a NaN and raises the “invalid” floating-point exception for even $n < 0$.
— **`rootn(`**x**`, 0)`** returns a NaN and raises the “invalid” floating-point exception for all x (including NaN).

— `rootn(`x`, `n`)` returns a NaN and raises the "invalid" floating-point exception for $x < 0$ and n even.

F.10.4.4d The `rsqrt` functions

— `rsqrt(±0)` returns ±∞ and raises the "divide-by-zero" floating-point exception.
— `rsqrt(`x`)` returns a NaN and raises the "invalid" floating-point exception for $x < 0$.
— `rsqrt(+∞)` returns +0.

After F.10.4.1, insert the following:

F.10.4.1a The `compoundn` functions

— `compoundn(`x`, 0)` returns 1 for $x \geq -1$ or x a NaN.
— `compoundn(`x`, `n`)` returns a NaN and raises the "invalid" floating-point exception for $x < -1$.
— `compoundn(−1, `n`)` returns +∞ and raises the divide-by-zero floating-point exception for $n < 0$.
— `compoundn(−1, `n`)` returns +0 for $n > 0$.

In the Preferred Quantum Exponents table in 5.2.4.2.3#7, insert before the final row:

`compoundn`	floor(**n** × min(0, Q(**x**)))
`pown`	floor(**n** × Q(**x**))
`powr`	floor(**y** × Q(**x**))
`rootn`	floor(Q(**x**)/**n**)
`rsqrt`	−floor(Q(**x**)/2)

8 Reduction functions for `<math.h>`

[NOTE This clause was not approved for inclusion in C2X.]

9 Future directions for `<complex.h>`

This clause extends the list of function names reserved for future library directions under **`<complex.h>`** to include complex versions of math functions that this part of Technical Specification 18661 adds to C11.

Change to C2X-20190607:

In 7.31.1#1, add the following to the list of function names:

`cexp2m1`	`crsqrt`	`casinpi`
`cexp10`	`ccompoundn`	`catanpi`
`cexp10m1`	`crootn`	`ccospi`
`clogp1`	`cpown`	`csinpi`
`clog2p1`	`cpowr`	`ctanpi`
`clog10p1`	`cacospi`	

10 Type-generic macros **<tgmath.h>**

The following changes to C2X-20190607 enhance the specification of type-generic macros in **<tgmath.h>** to apply to math functions that this Part of Technical Specification 18661 adds to C11.

Change to C2X-20190607:

In 7.25#9, add the following to the list of type-generic macros:

exp2m1	**rsqrt**	**asinpi**
exp10	**compoundn**	**atanpi**
exp10m1	**rootn**	**atan2pi**
logp1	**pown**	**cospi**
log2p1	**powr**	**sinpi**
log10p1	**acospi**	**tanpi**

11 Constant rounding modes **<fenv.h>**

As IEC 60559 operations, the **<math.h>** functions introduced in this part of ISO/IEC TS 18661 are subject to IEC 60559 constant rounding-direction attributes. The following changes to C2X-20190607 add these new functions to the set of functions affected by constant rounding modes in **<fenv.h>**.

Changes to C2X-20190607:

In 7.6.2#4, replace the table:

Header	Function families
<math.h>	**acos**, **asin**, **atan**, **atan2**
<math.h>	**cos**, **sin**, **tan**
<math.h>	**acosh**, **asinh**, **atanh**
<math.h>	**cosh**, **sinh**, **tanh**
<math.h>	**exp**, **exp2**, **expm1**
<math.h>	**log**, **log10**, **log1p**, **log2**
<math.h>	**scalbn**, **scalbln**, **ldexp**
<math.h>	**cbrt**, **hypot**, **pow**, **sqrt**
<math.h>	**erf**, **erfc**
<math.h>	**lgamma**, **tgamma**
<math.h>	**rint**, **nearbyint**, **lrint**, **llrint**
<math.h>	**fdim**
<math.h>	**fma**
<math.h>	**fadd**, **daddl**, **fsub**, **dsubl**, **fmul**, **dmull**, **fdiv**, **ddivl**, **ffma**, **dfmal**, **fsqrt**, **dsqrtl**
<stdlib.h>	**atof**, **strfrom**, **strto**
<wchar.h>	**wcsto**
<stdio.h>	**printf** and **scanf** families
<wchar.h>	**wprintf** and **wscanf** families

with:

Header	Function families
`<math.h>`	`acos`, `acospi`, `asin`, `asinpi`, `atan`, `atan2`, `atan2pi`, `atanpi`
`<math.h>`	`cos`, `cospi`, `sin`, `sinpi`, `tan`, `tanpi`
`<math.h>`	`acosh`, `asinh`, `atanh`
`<math.h>`	`cosh`, `sinh`, `tanh`
`<math.h>`	`exp`, `exp10`, `exp10m1`, `exp2`, `exp2m1`, `expm1`
`<math.h>`	`log`, `log10`, `log10p1`, `log1p`, `log2`, `log2p1`, `logp1`
`<math.h>`	`ldexp`, `scalbln`, `scalbn`
`<math.h>`	`cbrt`, `compoundn`, `hypot`, `pow`, `pown`, `powr`, `rootn`, `rsqrt`, `sqrt`
`<math.h>`	`erf`, `erfc`
`<math.h>`	`lgamma`, `tgamma`
`<math.h>`	`llrint`, `lrint`, `nearbyint`, `rint`
`<math.h>`	`fdim`
`<math.h>`	`fma`
`<math.h>`	`daddl`, `ddivl`, `dfmal`, `dmull`, `dsqrtl`, `dsubl`, `fadd`, `fdiv`, `ffma`, `fmul`, `fsqrt`, `fsub`
`<stdlib.h>`	`atof`, `strfrom`, `strto`
`<wchar.h>`	`wcsto`
`<stdio.h>`	`printf` and `scanf` families
`<wchar.h>`	`wprintf` and `wscanf` families

In 7.6.3#2, replace the table:

Header	Function families
`<math.h>`	`acos`, `asin`, `atan`, `atan2`
`<math.h>`	`cos`, `sin`, `tan`
`<math.h>`	`acosh`, `asinh`, `atanh`
`<math.h>`	`cosh`, `sinh`, `tanh`
`<math.h>`	`exp`, `exp2`, `expm1`
`<math.h>`	`log`, `log10`, `log1p`, `log2`
`<math.h>`	`scalbn`, `scalbln`, `ldexp`
`<math.h>`	`cbrt`, `hypot`, `pow`, `sqrt`
`<math.h>`	`erf`, `erfc`
`<math.h>`	`lgamma`, `tgamma`
`<math.h>`	`rint`, `nearbyint`, `lrint`, `llrint`
`<math.h>`	`quantize`
`<math.h>`	`fdim`
`<math.h>`	`fmad`
`<math.h>`	`d32add`, `d64add`, `d32sub`, `d64sub`, `d32mul`, `d64mul`, `d32div`, `d64div`, `d32fma`, `d64fma`, `d32sqrt`, `d64sqrt`
`<stdlib.h>`	`strfrom`, `strto`
`<wchar.h>`	`wcsto`

<stdio.h>	**printf** and **scanf** families
<wchar.h>	**wprintf** and **wscanf** families

with:

Header	Function families
<math.h>	**acos**, **acospi**, **asin**, **asinpi**, **atan**, **atan2**, **atan2pi**, **atanpi**
<math.h>	**cos**, **cospi**, **sin**, **sinpi**, **tan**, **tanpi**
<math.h>	**acosh**, **asinh**, **atanh**
<math.h>	**cosh**, **sinh**, **tanh**
<math.h>	**exp**, **exp10**, **exp10m1**, **exp2**, **exp2m1**, **expm1**
<math.h>	**log**, **log10**, **log10p1**, **log1p**, **log2**, **log2p1**, **logp1**
<math.h>	**ldexp**, **scalbln**, **scalbn**
<math.h>	**cbrt**, **compoundn**, **hypot**, **pow**, **pown**, **powr**, **rootn**, **rsqrt**, **sqrt**
<math.h>	**erf**, **erfc**
<math.h>	**lgamma**, **tgamma**
<math.h>	**llrint**, **lrint**, **nearbyint**, **rint**
<math.h>	**quantize**
<math.h>	**fdim**
<math.h>	**fmad**
<math.h>	**d32add, d64add, d32sub, d64sub, d32mul, d64mul, d32div, d64div, d32fma, d64fma, d32sqrt, d64sqrt**
<stdlib.h>	**strfrom**, **strto**
<wchar.h>	**wcsto**
<stdio.h>	**printf** and **scanf** families
<wchar.h>	**wprintf** and **wscanf** families

Bibliography

[1] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems, second edition*

[2] IEEE 754−1985, *IEEE Standard for Binary Floating-Point Arithmetic*

[3] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*

[4] IEEE 854−1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*

[5] ISO/IEC 9899:2011/Cor.1:2012, *Information technology — Programming languages — C / Technical Corrigendum 1*