

ISO/IEC JTC 1/SC 22/WG 14 N1835

Date: yyyy-mm-dd

Reference number of document: **ISO/IEC TS 18661-4**

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

Secretariat: ANSI

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

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

Contents

Page

Foreword

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

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

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

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

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

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

— *Part 1: Binary floating-point arithmetic*

— *Part 2: Decimal floating-point arithmetic*

— *Part 3: Interchange and extended types*

— *Part 4: Supplementary functions*

— *Part 5: Supplementary attributes*

Part 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*).

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

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

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

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

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

4 Provide direct support for

   a. Execution-time diagnosis of anomalies

   b. Smoother handling of exceptions

   c. Interval arithmetic at a reasonable cost

5 Provide for development of

   a. Standard elementary functions such as exp and cos

   b. Very high precision (multiword) arithmetic

   c. Coupling of numerical and symbolic algebraic computation

6 Enable rather than preclude further refinements and extensions.

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

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

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

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

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

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

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

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

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

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

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

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

C support for IEC 60559

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

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

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 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.

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

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

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

Additional background on supplementary functions

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

IEC 60559 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 Technical Specification 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 Technical Specification 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 Technical Specification 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 document, Part 4 of Technical Specification 18661, extends programming language C to include functions specified and recommended in ISO/IEC/IEEE 60559:2011.

2 Conformance

An implementation conforms to Part 4 of Technical Specification 18661 if

a) It meets the requirements for a conforming implementation of C11 with all the changes to C11 as specified in Parts 1-4 of Technical Specification 18661;

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

c) It defines **`__STDC_IEC_60559_FUNCS__`** to **`201`**`ymm`**`L`**.

3 Normative references

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

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

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

ISO/IEC/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*

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*

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

Changes specified in Part 4 of Technical Specification 18661 are relative to ISO/IEC 9899:2011, including *Technical Corrigendum 1* (ISO/IEC 9899:2011/Cor. 1:2012), together with the changes from Parts 1, 2, and 3 of Technical Specification 18661.

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*, 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 + TS18661-3 allows freestanding implementations to conform to this Part of Technical Specification18661.

5.2 Predefined macros

Change to C11 + TS18661-1 + TS18661-2 + TS18661-3:

In 6.10.8.3#1, add:

> **`__STDC_IEC_60559_FUNCS__`** The integer constant **`201ymmL`**, intended to indicate support of functions specified and recommended in IEC 60559.

5.3 Standard headers

The new identifiers added to C11 library headers by this Part of Technical Specification 18661 are defined or declared by their respective headers only if **`__STDC_WANT_IEC_60559_FUNCS_EXT__`** is defined as a macro at the point in the source file where the appropriate header is first included. The following changes to C11 + TS18661-1 + TS18661-2 + TS18661-3 list these identifiers in each applicable library subclause.

Changes to C11 + TS18661-1 + TS18661-2 + TS18661-3:

In 7.12, renumber paragraph 1e to 1h, and after paragraph 1d insert the paragraphs:

> [1e] The following identifiers are declared only if **`__STDC_WANT_IEC_60559_FUNCS_EXT__`** is defined as a macro at the point in the source file where **`<math.h>`** is first included:

exp2m1	**rootnf**	**sinpil**
exp2m1f	**rootnl**	**tanpi**
exp2m1l	**pown**	**tanpif**
exp10	**pownf**	**tanpil**
exp10f	**pownl**	**reduc_sum**
exp10l	**powr**	**reduc_sumf**
exp10m1	**powrf**	**reduc_suml**
exp10m1f	**powrl**	**reduc_sumabs**
exp10m1l	**acospi**	**reduc_sumabsf**
logp1	**acospif**	**reduc_sumabsl**
logp1f	**acospil**	**reduc_sumsq**
logp1l	**asinpi**	**reduc_sumsqf**
log2p1	**asinpif**	**reduc_sumsql**
log2p1f	**asinpil**	**reduc_sumprod**
log2p1l	**atanpi**	**reduc_sumprodf**
log10p1	**atanpif**	**reduc_sumprodl**
log10p1f	**atanpil**	**scaled_prod**
log10p1l	**atan2pi**	**scaled_prodf**
rsqrt	**atan2pif**	**scaled_prodl**
rsqrtf	**atan2pil**	**scaled_prodsum**
rsqrtl	**cospi**	**scaled_prodsumf**
compoundn	**cospif**	**scaled_prodsuml**
compoundnf	**cospil**	**scaled_proddiff**

compoundnl	**sinpi**	**scaled_proddifff**
rootn	**sinpif**	**scaled_proddiffl**

[1f] The following identifiers are declared only if **__STDC_WANT_IEC_60559_DFP_EXT__** and **__STDC_WANT_IEC_60559_FUNCS_EXT__** are defined as macros at the point in the source file where **<math.h>** is first included:

for supported types **_Decimal***N*, where N = 32, 64, and 128:

exp2m1d*N*	**pownd***N*	**tanpid***N*
exp10d*N*	**powrd***N*	**reduc_sumd***N*
exp10m1d*N*	**acospid***N*	**reduc_sumabsd***N*
logp1d*N*	**asinpid***N*	**reduc_sumsqd***N*
log2p1d*N*	**atanpid***N*	**reduc_sumprodd***N*
log10p1d*N*	**atan2pid***N*	**scaled_prodd***N*
rsqrtd*N*	**cospid***N*	**scaled_prodsumd***N*
compoundnd*N*	**sinpid***N*	**scaled_proddiffd***N*
rootnd*N*		

[1g] The following identifiers are declared only if **__STDC_WANT_IEC_60559_TYPES_EXT__** and **__STDC_WANT_IEC_60559_FUNCS_EXT__** are defined as macros at the point in the source file where **<math.h>** is first included:

for supported types **_Float***N*:

exp2m1f*N*	**pownf***N*	**tanpif***N*
exp10f*N*	**powrf***N*	**reduc_sumf***N*
exp10m1f*N*	**acospif***N*	**reduc_sumabsf***N*
logp1f*N*	**asinpif***N*	**reduc_sumsqf***N*
log2p1f*N*	**atanpif***N*	**reduc_sumprodf***N*
log10p1f*N*	**atan2pif***N*	**scaled_prodf***N*
rsqrtf*N*	**cospif***N*	**scaled_prodsumf***N*
compoundnf*N*	**sinpif***N*	**scaled_proddifff***N*
rootnf*N*		

for supported types **_Float***N***x**:

exp2m1f*N***x**	**pownf***N***x**	**tanpif*N*x**
exp10f*N***x**	**powrf***N***x**	**reduc_sumf***N***x**
exp10m1f*N***x**	**acospif***N***x**	**reduc_sumabsf***N***x**
logp1f*N***x**	**asinpif***N***x**	**reduc_sumsqf***N***x**
log2p1f*N***x**	**atanpif***N***x**	**reduc_sumprodf***N***x**
log10p1f*N***x**	**atan2pif***N***x**	**scaled_prodf***N***x**
rsqrtf*N***x**	**cospif***N***x**	**scaled_prodsumf***N***x**
compoundnf*N***x**	**sinpif***N***x**	**scaled_proddifff***N***x**
rootnf*N***x**		

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

exp2m1d*N*	**pownd***N*	**tanpid***N*
exp10d*N*	**powrd***N*	**reduc_sumd***N*
exp10m1d*N*	**acospid***N*	**reduc_sumabsd***N*
logp1d*N*	**asinpid***N*	**reduc_sumsqd***N*
log2p1d*N*	**atanpid***N*	**reduc_sumprodd***N*
log10p1d*N*	**atan2pid***N*	**scaled_prodd***N*
rsqrtd*N*	**cospid***N*	**scaled_prodsumd***N*

compoundnd*N* **sinpid***N* **scaled_proddiffd***N*
rootnd*N*

for supported types **_Decimal***N***X**:

exp2m1d*N***x** **pownd***N***x** **tanpid***N***x**
exp10d*N***x** **powrd***N***x** **reduc_sumd***N***x**
exp10m1d*N***x** **acospid***N***x** **reduc_sumabsd***N***x**
logp1d*N***x** **asinpid***N***x** **reduc_sumsqd***N***x**
log2p1d*N***x** **atanpid***N***x** **reduc_sumprodd***N***x**
log10p1d*N***x** **atan2pid***N***x** **scaled_prodd***N***x**
rsqrtd*N***x** **cospid***N***x** **scaled_prodsumd***N***x**
compoundnd*N***x** **sinpid***N***x** **scaled_proddiffd***N***x**
rootnd*N***x**

After 7.25#1c, insert the paragraph:

[1d] The following identifiers are defined as type-generic macros only if **__STDC_WANT_IEC_60559_FUNCS_EXT__** is defined as a macro at the point in the source file where **<tgmath.h>** is first included:

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

6 Operation binding

The following change to C11 + TS18661-1 + TS18661-2 + TS18661-3 shows how functions in C11 and in this Part of Technical Specification 18661 provide operations recommended in IEC 60559.

Change to C11 + TS18661-1 + TS18661-2 + TS18661-3:

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 (except for the reduction operations), is not required for the C functions in the table. See also 7.31.6a.

IEC 60559 operation	C function	Clauses - C11
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
sum	`reduc_sum`	7.12.13b.1, F.10.10b.1
dot	`reduc_sumprod`	7.12.13b.4, F.10.10b.4
sumSquare	`reduc_sumsq`	7.12.13b.3, F.10.10b.3
sumAbs	`reduc_sumabs`	7.12.13b.2, F.10.13b.2
scaledProd	`scaled_prod`	7.12.13b.5, F.10.10b.5
scaledProdSum	`scaled_prodsum`	7.12.13b.6, F.10.10b.6
scaledProdDiff	`scaled_proddiff`	7.12.13b.7, F.10.10b.7

7 Mathematical functions in `<math.h>`

This clause specifies changes to C11 + TS18661-1 + TS18661-2 + TS18661-3 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 include support for the symmetry and antisymmetry properties that IEC 60559 specifies for mathematical functions.

Changes to C11 + TS18661-1 + TS18661-2 + TS18661-3:

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);
_FloatN acospifN(_FloatN x);
_FloatNx acospifNx(_FloatNx x);
_DecimalN acospidN(_DecimalN x);
_DecimalNx acospidNx(_DecimalNx x);
```

Description

[2] The **acospi** functions compute 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);
_FloatN asinpifN(_FloatN x);
_FloatNx asinpifNx(_FloatNx x);
_DecimalN asinpidN(_DecimalN x);
_DecimalNx asinpidNx(_DecimalNx x);
```

Description

[2] The **asinpi** functions compute 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);
_FloatN atanpifN(_FloatN x);
_FloatNx atanpifNx(_FloatNx x);
_DecimalN atanpidN(_DecimalN x);
_DecimalNx atanpidNx(_DecimalNx x);
```

Description

[2] The **atanpi** functions compute 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);
_FloatN atan2pifN(_FloatN y, _FloatN x);
_FloatNx atan2pifNx(_FloatNx y, _FloatNx x);
_DecimalN atan2pidN(_DecimalN y, _DecimalN x);
_DecimalNx atan2pidNx(_DecimalNx y, _DecimalNx 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);
_FloatN cospifN(_FloatN x);
_FloatNx cospifNx(_FloatNx x);
_DecimalN cospidN(_DecimalN x);
_DecimalNx cospidNx(_DecimalNx x);
```

Description

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

Returns

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

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);
_FloatN sinpifN(_FloatN x);
_FloatNx sinpifNx(_FloatNx x);
_DecimalN sinpidN(_DecimalN x);
_DecimalNx sinpidNx(_DecimalNx x);
```

Description

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

Returns

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

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);
_FloatN tanpifN(_FloatN x);
_FloatNx tanpifNx(_FloatNx x);
_DecimalN tanpidN(_DecimalN x);
_DecimalNx tanpidNx(_DecimalNx 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 7.12.6.9, replace the subclause title:

7.12.6.9 The `log1p` functions

with:

7.12.6.9 The `log1p` and `logp1` functions

In 7.12.6.9#1, append to the Synopsis:

```
double logp1(double x);
float logp1f(float x);
long double logp1l(long double x);
_FloatN logp1fN(_FloatN x);
_FloatNx logp1fNx(_FloatNx x);
_DecimalN logp1dN(_DecimalN x);
_DecimalNx logp1dNx(_DecimalNx x);
```

In 7.12.6.9#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 7.12.6.9#3:

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

with:

[3] These functions return $\log_e$ (1 + **x**).

In F.10.3.9, replace the subclause title:

F.10.3.9 The `log1p` functions

with:

F.10.3.9 The `log1p` and `logp1` functions

After 7.12.6.13, insert the following:

7.12.6.14 The `exp2m1` functions

Synopsis

[1]
```
#include <math.h>
double exp2m1(double x);
float exp2m1f(float x);
long double exp2m1l(long double x);
_FloatN exp2m1fN(_FloatN x);
_FloatNx exp2m1fNx(_FloatNx x);
_DecimalN exp2m1dN(_DecimalN x);
_DecimalNx exp2m1dNx(_DecimalNx 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 $2^x - 1$.

7.12.6.15 The `exp10` functions

Synopsis

[1]
```
#include <math.h>
double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);
_FloatN exp10fN(_FloatN x);
_FloatNx exp10fNx(_FloatNx x);
_DecimalN exp10dN(_DecimalN x);
_DecimalNx exp10dNx(_DecimalNx 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.

7.12.6.16 The `exp10m1` functions

Synopsis

[1]
```
#include <math.h>
double exp10m1(double x);
float exp10m1f(float x);
long double exp10m1l(long double x);
_FloatN exp10m1fN(_FloatN x);
_FloatNx exp10m1fNx(_FloatNx x);
_DecimalN exp10m1dN(_DecimalN x);
_DecimalNx exp10m1dNx(_DecimalNx 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.

Returns

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

7.12.6.17 The `log2p1` functions

Synopsis

[1]
```
#include <math.h>
double log2p1(double x);
float log2p1f(float x);
long double log2p1l(long double x);
_FloatN log2p1fN(_FloatN x);
_FloatNx log2p1fNx(_FloatNx x);
_DecimalN log2p1dN(_DecimalN x);
_DecimalNx log2p1dNx(_DecimalNx 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 $\log_2(1 + \mathbf{x})$.

7.12.6.18 The `log10p1` functions

Synopsis

[1]
```
#include <math.h>
double log10p1(double x);
float log10p1f(float x);
long double log10p1l(long double x);
_FloatN log10p1fN(_FloatN x);
_FloatNx log10p1fNx(_FloatNx x);
_DecimalN log10p1dN(_DecimalN x);
_DecimalNx log10p1dNx(_DecimalNx 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 7.12.7.5, insert the following:

7.12.7.6 The `rsqrt` functions

Synopsis

[1]
```
#include <math.h>
double rsqrt(double x);
float rsqrtf(float x);
long double rsqrtl(long double x);
_FloatN rsqrtfN(_FloatN x);
_FloatNx rsqrtfNx(_FloatNx x);
_DecimalN rsqrtdN(_DecimalN x);
_DecimalNx rsqrtdNx(_DecimalNx 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 $1 / \sqrt{\mathbf{x}}$.

7.12.7.7 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);
_FloatN compoundnfN(_FloatN x, intmax_t n);
_FloatNx compoundnfNx(_FloatNx x, intmax_t n);
_DecimalN compoundndN(_DecimalN x, intmax_t n);
_DecimalNx compoundndNx(_DecimalNx 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 finite **n** is too large, depending on **x**. A pole error may occur if **x** equals −1 and **n** < 0.

Returns

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

7.12.7.8 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);
_FloatN rootnfN(_FloatN x, intmax_t n);
_FloatNx rootnfNx(_FloatNx x, intmax_t n);
_DecimalN rootndN(_DecimalN x, intmax_t n);
_DecimalNx rootndNx(_DecimalNx 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 $\mathbf{x}^{1/n}$.

7.12.7.9 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);
_FloatN pownfN(_FloatN x, intmax_t n);
_FloatNx pownfNx(_FloatNx x, intmax_t n);
_DecimalN powndN(_DecimalN x, intmax_t n);
_DecimalNx powndNx(_DecimalNx 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 $\mathbf{x}^{n}$.

7.12.7.10 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);
_FloatN powrfN(_FloatN x, _FloatN y);
_FloatNx powrfNx(_FloatNx x, _FloatNx y);
_DecimalN powrdN(_DecimalN x, _DecimalN y);
_DecimalNx powrdNx(_DecimalNx x, _DecimalNx 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 $\mathbf{x}^{\mathbf{y}}$.

After 7.31.6, insert:

7.31.6a Mathematics <math.h>

With the condition that the macro **__STDC_IEC_60559_FUNCS__** is defined, 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**, **f***N*, **f***N***x**, **d***N*, or **d***N***x** may be added to the **<math.h>** header.

In 7.31.6a, attach a footnote to the wording:

With the condition that the macro **__STDC_IEC_60559_FUNCS__** is defined, the function names

where the footnote is:

*) 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, *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, *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 for finite *x*.
— **atan2pi**(±∞, +∞) returns ±1/4 for finite *x*.

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.10.3.13, insert the following:

F.10.3.14 The `exp2m1` functions

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

F.10.3.15 The `exp10` functions

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

F.10.3.16 The `exp10m1` functions

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

F.10.3.17 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 +∞.

F.10.3.18 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.10.4.5, insert the following:

F.10.4.6 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.

F.10.4.7 The `compoundn` functions

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

F.10.4.8 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*`)` is equivalent to **`rootn`**`(±0, −`*n*`)` for *n* not 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.10.4.9 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.

F.10.4.10 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.

8 Reduction functions in `<math.h>`

This clause specifies changes to C11 + TS18661-1 + TS18661-2 + TS18661-3 to include functions that support reduction operations recommended by IEC 60559.

Changes to C11 + TS18661-1 + TS18661-2 + TS18661-3:

After 7.12.13a, insert the following:

7.12.13b Reduction functions

The functions in this subclause should be implemented so that intermediate computations do not overflow or underflow.

Functions computing sums of length **`n`** = 0 return the value +0. Functions computing products of length **`n`** = 0 return the value 1 and store the scale factor 0 in the object pointed to by **`sfptr`**.

7.12.13b.1 The `reduc_sum` functions

Synopsis

[1]
```
#include <math.h>
double reduc_sum(size_t n, const double p[static n]);
float reduc_sumf(size_t n, const float p[static n]);
long double reduc_suml(size_t n, const long double p[static n]);
_FloatN reduc_sumfN(size_t n, const _FloatN p[static n]);
_FloatNx reduc_sumfNx(size_t n, const _FloatNx p[static n]);
_DecimalN reduc_sumdN(size_t n, const _DecimalN p[static n]);
_DecimalNx reduc_sumdNx(size_t n, const _DecimalNx p[static n]);
```

Description

[2] The **reduc_sum** functions compute the sum of the **n** members of array **p**: $\Sigma_{i=0,n-1}$**p[i]**. A range error may occur.

Returns

[3] The **reduc_sum** functions return the computed sum.

7.12.13b.2 The `reduc_sumabs` functions

Synopsis

[1]
```
#include <math.h>
double reduc_sumabs(size_t n, const double p[static n]);
float reduc_sumabsf(size_t n, const float p[static n]);
long double reduc_sumabsl(size_t n, const long double p[static n]);
_FloatN reduc_sumabsfN(size_t n, const _FloatN p[static n]);
_FloatNx reduc_sumabsfNx(size_t n, const _FloatNx p[static n]);
_DecimalN reduc_sumabsdN(size_t n, const _DecimalN p[static n]);
_DecimalNx reduc_sumabsdNx(size_t n, const _DecimalNx p[static n]);
```

Description

[2] The **reduc_sumabs** functions compute the sum of the absolute values of the **n** members of array **p**: $\Sigma_{i=0,n-1}$|**p[i]**|. A range error may occur.

Returns

[3] The **reduc_sumabs** functions return the computed sum.

7.12.13b.3 The `reduc_sumsq` functions

Synopsis

[1]
```
#include <math.h>
double reduc_sumsq(size_t n, const double p[static n]);
float reduc_sumsqf(size_t n, const float p[static n]);
long double reduc_sumsql(size_t n, const long double p[static n]);
_FloatN reduc_sumsqfN(size_t n, const _FloatN p[static n]);
_FloatNx reduc_sumsqfNx(size_t n, const _FloatNx p[static n]);
_DecimalN reduc_sumsqdN(size_t n, const _DecimalN p[static n]);
_DecimalNx reduc_sumsqdNx(size_t n, const _DecimalNx p[static n]);
```

Description

[2] The **reduc_sumsq** functions compute the sum of squares of the values of the **n** members of array **p**: $\Sigma_{i=0,n-1}$ (`p[i]` × `p[i]`). A range error may occur.

Returns

[3] The **reduc_sumsq** functions return the computed sum.

7.12.13b.4 The `reduc_sumprod` functions

Synopsis

[1]
```
#include <math.h>
double reduc_sumprod(size_t n, const double p[static n],
   const double q[static n]);
float reduc_sumprodf(size_t n, const float p[static n],
   const float q[static n]);
long double reduc_sumprodl(size_t n, const long double p[static n],
   const long double q[static n]);
_FloatN reduc_sumprodfN(size_t n, const _FloatN p[static n],
   const _FloatN q[static n]);
_FloatNx reduc_sumprodfNx(size_t n, const _FloatNx p[static n],
   const _FloatNx q[static n]);
_DecimalN reduc_sumproddN(size_t n, const _DecimalN p[static n],
   const _DecimalN q[static n]);
_DecimalNx reduc_sumproddNx(size_t n, const _DecimalNx p[static n],
   const _DecimalNx q[static n]);
```

Description

[2] The **reduc_sumprod** functions compute the dot product of the sequences of members of the arrays **p** and **q**: $\Sigma_{i=0,n-1}$ (`p[i]` × `q[i]`). A range error may occur.

Returns

[3] The **reduc_sumprod** functions return the computed sum.

7.12.13b.5 The `scaled_prod` functions

Synopsis

[1]
```
#include <math.h>
double scaled_prod(size_t n, const double p[static n],
   intmax_t * restrict sfptr);
float scaled_prodf(size_t n, const float p[static n],
   intmax_t * restrict sfptr);
long double scaled_prodl(size_t n, const long double p[static n],
   intmax_t * restrict sfptr);
_FloatN scaled_prodfN(size_t n, const _FloatN p[static n],
   intmax_t * restrict sfptr);
_FloatNx scaled_prodfNx(size_t n, const _FloatNx p[static n],
   intmax_t * restrict sfptr);
_DecimalN scaled_proddN(size_t n, const _DecimalN p[static n],
   intmax_t * restrict sfptr);
_DecimalNx scaled_proddNx(size_t n, const _DecimalNx p[static n],
   intmax_t * restrict sfptr);
```


Description

[2] The **`scaled_prod`** functions compute a scaled product *pr* of the **`n`** members of the array **`p`** and a scale factor *sf*, such that $pr \times b^{sf} = \Pi_{i=0,n-1}$**`p[i]`**, where *b* is the radix of the type. These functions store the scale factor *sf* in the object pointed to by **`sfptr`**. A domain error occurs if the scale factor is outside the range of the **`intmax_t`** type. The functions should not cause a range error.

Returns

[3] The **`scaled_prod`** functions return the computed scaled product *pr*.

7.12.13b.6 The `scaled_prodsum` functions

Synopsis

[1]
```
#include <math.h>
double scaled_prodsum(size_t n, const double p[static n],
   const double q[static n], intmax_t * restrict sfptr);
float scaled_prodsumf(size_t n, const float p[static n],
   const float q[static n], intmax_t * restrict sfptr);
long double scaled_prodsuml(size_t n, const long double p[static n],
   const long double q[static n], intmax_t * restrict sfptr);
_FloatN scaled_prodsumfN(size_t n, const _FloatN p[static n],
   const _FloatN q[static n], intmax_t * restrict sfptr);
_FloatNx scaled_prodsumfNx(size_t n, const _FloatNx p[static n],
   const _FloatNx q[static n], intmax_t * restrict sfptr);
_DecimalN scaled_prodsumdN(size_t n, const _DecimalN p[static n],
   const _DecimalN q[static n], intmax_t * restrict sfptr);
_DecimalNx scaled_prodsumdNx(size_t n, const _DecimalNx p[static n],
   const _DecimalNx q[static n], intmax_t * restrict sfptr);
```

Description

[2] The **`scaled_prodsum`** functions compute a scaled product *pr* of the sums of the corresponding members of the arrays **`p`** and **`q`** and a scale factor *sf*, such that $pr \times b^{sf} = \Pi_{i=0,n-1}$(**`p[i]`** + **`q[i]`**), where *b* is the radix of the type. These functions store the scale factor *sf* in the object pointed to by **`sfptr`**. A domain error occurs if the scale factor is outside the range of the **`intmax_t`** type. These functions should not cause a range error.

Returns

[3] The **`scaled_prodsum`** functions return the computed scaled product *pr*.

7.12.13b.7 The `scaled_proddiff` functions

Synopsis

[1]
```
#include <math.h>
double scaled_proddiff(size_t n, const double p[static n],
   const double q[static n], intmax_t * restrict sfptr);
float scaled_proddifff(size_t n, const float p[static n],
   const float q[static n], intmax_t * restrict sfptr);
long double scaled_proddiffl(size_t n, const long double p[static n],
   const long double q[static n], intmax_t * restrict sfptr);
_FloatN scaled_proddifffN(size_t n, const _FloatN p[static n],
   const _FloatN q[static n], intmax_t * restrict sfptr);
_FloatNx scaled_proddifffNx(size_t n, const _FloatNx p[static n],
   const _FloatNx q[static n], intmax_t * restrict sfptr);
_DecimalN scaled_proddiffdN(size_t n, const _DecimalN p[static n],
   const _DecimalN q[static n], intmax_t * restrict sfptr);
_DecimalNx scaled_proddiffdNx(size_t n, const _DecimalNx p[static n],
   const _DecimalNx q[static n], intmax_t * restrict sfptr);
```

Description

[2] The **`scaled_proddiff`** functions compute a scaled product *pr* of the differences of the corresponding members of the arrays **`p`** and **`q`** and a scale factor *sf*, such that $pr \times b^{sf} = \Pi_{i=0,n-1}($**`p[i]`** $-$ **`q[i]`**$)$, where *b* is the radix of the type. These functions store the scale factor *sf* in the object pointed to by **`sfptr`**. A domain error occurs if the scale factor is outside the range of the **`intmax_t`** type. These functions should not cause a range error.

Returns

[3] The **`scaled_proddiff`** functions return the computed scaled product *pr*.

After F.10.10a, insert

F.10.10b Reduction functions

The functions in this subclause return a NaN if any member of an array argument is a NaN.

The **`reduc_sum`**, **`reduc_sumabs`**, **`reduc_sumsq`**, and **`reduc_sumprod`** functions avoid overflow and underflow in intermediate computation. They raise the “overflow” or “underflow” floating-point exception if and only if the determination of the final result overflows or underflows.

The **`scaled_prod`**, **`scaled_prodsum`**, and **`scaled_proddiff`** functions do not raise the “overflow” or “underflow” floating-point exceptions.

The functions in this subclause do not raise the “divide-by-zero” floating-point exception.

F.10.10b.1 The `reduc_sum` functions

— **`reduc_sum(n, p)`** returns a NaN if any member of array **`p`** is a NaN.
— **`reduc_sum(n, p)`** returns a NaN and raises the “invalid” floating-point exception if any two members of array **`p`** are infinities with different signs.
— Otherwise, **`reduc_sum(n, p)`** returns ±∞ if the members of p include one or more infinities ±∞ (with the same sign).

F.10.10b.2 The `reduc_sumabs` functions

— **`reduc_sumabs(n, p)`** returns +∞ if any member of array **`p`** is an infinity.
— Otherwise, **`reduc_sumabs(n, p)`** returns a NaN if any member of array **`p`** is a NaN.

F.10.10b.3 The `reduc_sumsq` functions

— **`reduc_sumsq(n, p)`** returns +∞ if any member of array **`p`** is an infinity.
— Otherwise, **`reduc_sumsq(n, p)`** returns a NaN if any member of array **`p`** is a NaN.

F.10.10b.4 The `reduc_sumprod` functions

— **`reduc_sumprod(n, p, q)`** returns a NaN if any member of array **`p`** or **`q`** is a NaN.
— **`reduc_sumprod(n, p, q)`** returns a NaN and raises the "invalid" floating-point exception if any of the products has a zero and an infinite factor.
— **`reduc_sumprod(n, p, q)`** returns a NaN and raises the "invalid" floating-point exception if any two of the products are (exact) infinities with different signs.
— Otherwise, **`reduc_sumprod(n,p)`** returns ±∞ if one or more of the products are (exactly) ±∞ (with the same sign).

F.10.10b.5 The `scaled_prod` functions

— **`scaled_prod(n, p, sfptr)`** returns a NaN if any member of array **`p`** is a NaN.
— **`scaled_prod(n, p, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if any two members of array **`p`** are a zero and an infinity.
— Otherwise, **`scaled_prod(n, p, sfptr)`** returns an infinity if any member of array **`p`** is an infinity.
— Otherwise, **`scaled_prod(n, p, sfptr)`** returns a zero if any member of array **`p`** is a zero.
— Otherwise, **`scaled_prod(n, p, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if the scale factor is outside the range of the **`intmax_t`** type.

F.10.10b.6 The `scaled_prodsum` functions

— **`scaled_prodsum(n, p, q, sfptr)`** returns a NaN if any member of **`p`** or **`q`** is a NaN.
— **`scaled_prodsum(n, p, q, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if any two factors (each of which is a sum) are zero and infinity (exactly).
— **`scaled_prodsum(n, p, q, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if any of the sums is of two infinities with different signs.
— Otherwise, **`scaled_prodsum(n, p, q, sfptr)`** returns an infinity if any factor is an exact infinity.
— Otherwise, **`scaled_prodsum(n, p, q, sfptr)`** returns a zero if any factor is a zero.
— Otherwise, **`scaled_prodsum(n, p, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if the scale factor is outside the range of the **`intmax_t`** type.

F.10.10b.7 The `scaled_proddiff` functions

— **`scaled_proddiff(n, p, q, sfptr)`** returns a NaN if any member of **`p`** or **`q`** is a NaN.
— **`scaled_proddiff(n, p, q, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if any two factors (each of which is a difference) are zero and infinity (exactly).
— **`scaled_proddiff(n, p, q, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if any of the differences is of two infinities with the same signs.
— Otherwise, **`scaled_proddiff(n, p, q, sfptr)`** returns an infinity if any factor is an exact infinity.
— Otherwise, **`scaled_proddiff(n, p, q, sfptr)`** returns a zero if any factor is a zero.
— Otherwise, **`scaled_proddiff(n, p, sfptr)`** returns a NaN and raises the "invalid" floating-point exception if the scale factor is outside the range of the **`intmax_t`** type.

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 C11 + TS18661-1 + TS18661-2 + TS18661-3:

In 7.31.1, add the following after the list of function names:

> and, with the condition that the macro **`__STDC_IEC_60559_FUNCS__`** is defined, the functions
>
> | | | |
> |---|---|---|
> | **`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 C11 + TS18661-1 + TS18661-2 + TS18661-3 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.

Changes to C11 + TS18661-1 + TS18661-2 + TS18661-3:

In 7.25#5, change:

> For each unsuffixed function in **`<math.h>`** without a **`c`**-prefixed counterpart in **`<complex.h>`** (except **`modf`**, **`setpayload`**, **`setpayloadsig`**, and **`canonicalize`**) …

to:

> For each unsuffixed function in **`<math.h>`** without a **`c`**-prefixed counterpart in **`<complex.h>`** (except **`modf`**, **`setpayload`**, **`setpayloadsig`**, **`canonicalize`**, and the reduction functions in 7.12.13b) …

In 7.25#5, 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`**

Bibliography

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

[2] ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*

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

[4] 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*

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

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

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

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

