Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 4: Supplementary functions

Warning

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

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

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

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

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

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

Violators may be prosecuted.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>v</td>
</tr>
<tr>
<td>Background</td>
<td>v</td>
</tr>
<tr>
<td>IEC 60559 floating-point standard</td>
<td>v</td>
</tr>
<tr>
<td>C support for IEC 60559</td>
<td>vi</td>
</tr>
<tr>
<td>Purpose</td>
<td>vii</td>
</tr>
<tr>
<td>Additional background on supplementary functions</td>
<td>vii</td>
</tr>
<tr>
<td>1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>2 Conformance</td>
<td>1</td>
</tr>
<tr>
<td>3 Normative references</td>
<td>1</td>
</tr>
<tr>
<td>4 Terms and definitions</td>
<td>1</td>
</tr>
<tr>
<td>5 C standard conformance</td>
<td>2</td>
</tr>
<tr>
<td>5.1 Freestanding implementations</td>
<td>2</td>
</tr>
<tr>
<td>5.2 Predefined macros</td>
<td>2</td>
</tr>
<tr>
<td>5.3 Standard headers</td>
<td>2</td>
</tr>
<tr>
<td>6 Operation binding</td>
<td>4</td>
</tr>
<tr>
<td>7 Mathematical functions in <code>&lt;math.h&gt;</code></td>
<td>5</td>
</tr>
<tr>
<td>8 Reduction functions in <code>&lt;math.h&gt;</code></td>
<td>17</td>
</tr>
<tr>
<td>9 Future directions for <code>&lt;complex.h&gt;</code></td>
<td>23</td>
</tr>
<tr>
<td>10 Type-generic macros <code>&lt;tgmath.h&gt;</code></td>
<td>23</td>
</tr>
<tr>
<td>Bibliography</td>
<td>24</td>
</tr>
</tbody>
</table>
Foreword

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

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

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

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

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

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

— Part 1: Binary floating-point arithmetic
— Part 2: Decimal floating-point arithmetic
— Part 3: Interchange and extended types
— Part 4: Supplementary functions
— Part 5: Supplementary attributes


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

Introduction

Background

IEC 60559 floating-point standard

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

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

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

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

4. Provide direct support for
   a. Execution-time diagnosis of anomalies
   b. Smoother handling of exceptions
   c. Interval arithmetic at a reasonable cost

5. Provide for development of
   a. Standard elementary functions such as exp and cos
   b. Very high precision (multiword) arithmetic
   c. Coupling of numerical and symbolic algebraic computation

6. Enable rather than preclude further refinements and extensions.

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

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

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

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

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

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

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

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

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

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

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

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

C support for IEC 60559

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

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


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

Purpose

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

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

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

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

Additional background on 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


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 ymmL.

3 Normative references

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

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

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


ISO/IEC 18661-1: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

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 Specification 18661.

5.2 Predefined macros

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

In 6.10.8.3#1, add:

```c
__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:

```c
exp2m1  rootnf  sinpl
exp2m1f rootnl  tanpi
exp2mlf rootnlf tanpif
exp2ml  powln  tanpil
exp10  pownf  tanpil
exp10f pownl  reduc_sum
exp10l powr  reduc_sumf
exp10ml powrl  reduc_sumbs
exp10mlf powrfl  reduc_sumabs
exp10ml  acospi  reduc_sumbsf
logp1  acospif  reduc_sumbsl
logp1f acospil  reduc_sumsq
logp1l acospi  reduc_sumssqf
log2p1  asinpif  reduc_sumssqf
log2p1f asinpil  reduc_sumssql
log2p1l asinpi  reduc_sumprod
log10p1 atanpi  reduc_sumprodfl
log10p1f atanpif  reduc_sumprodsl
log10p1l atanpi  reduc_sumprodql
log10p1f atanpil  scaled_prod
log10p1l atan2pl  scaled_prodf
rsqrt atan2pif scaled_prodl
rsqrtfl atan2pil scaled_prodlf
rsqrtl  cospi  scaled_prodsdf
compoundn  cospif  scaled_prodsdf
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 _DecimalN, where N = 32, 64, and 128:

exp2m1dN  powndN  tanpidN
exp10dN    powrdN  reduc_sumdN
exp10m1dN  acospidN  reduc_sumabsdN
logp1dN    asinpidN  reduc_sumsqdN
log2p1dN   atanpidN  reduc_sumproddN
log10p1dN  atan2pidN  scaled_proddN
rsqrtdN    cospidN  scaled_prodsimdN
compoundndN sinpidN  scaled_proddifldN
rootndN

[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 _FloatN:

exp2m1fN  pownfN  tanpifN
exp10fN    powrfN  reduc_sumfN
exp10m1fN  acospifN  reduc_sumabsfN
logp1fN    asinpifN  reduc_sumsqfN
log2p1fN   atanpifN  reduc_sumprodfN
log10p1fN  atan2pifN  scaled_prodfN
rsqrtfN    cospifN  scaled_prodsimfN
compoundnfN sinpifN  scaled_proddifffN
rootnfN

for supported types _FloatNx:

exp2m1fNx  pownfNx  tanpifNx
exp10fNx    powrfNx  reduc_sumfNx
exp10m1fNx  acospifNx  reduc_sumabsfNx
logp1fNx    asinpifNx  reduc_sumsqfNx
log2p1fNx   atanpifNx  reduc_sumprodfNx
log10p1fNx  atan2pifNx  scaled_prodfNx
rsqrtfNx    cospifNx  scaled_prodsimfNx
compoundnfNx sinpifNx  scaled_proddifffNx
rootnfNx

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

exp2m1dN  powndN  tanpidN
exp10dN    powrdN  reduc_sumdN
exp10m1dN  acospidN  reduc_sumabsdN
logp1dN    asinpidN  reduc_sumsqdN
log2p1dN   atanpidN  reduc_sumproddN
log10p1dN  atan2pidN  scaled_proddN
rsqrtdN    cospidN  scaled_prodsimdN

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, exp10m1, exp2, exp10, logp1, log2p1, log10p1, rsqrt, compoundn, rootn, asinpi, atanpi, cospi, sinpi, 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.

<table>
<thead>
<tr>
<th>IEC 60559 operation</th>
<th>C function</th>
<th>Clauses - C11</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>exp</td>
<td>7.12.6.1, F.10.3.1</td>
</tr>
<tr>
<td>expm1</td>
<td>expm1</td>
<td>7.12.6.3, F.10.3.3</td>
</tr>
<tr>
<td>exp</td>
<td>exp</td>
<td>7.12.6.2, F.10.3.2</td>
</tr>
<tr>
<td>exp2</td>
<td>exp2</td>
<td>7.12.6.14, F.10.3.14</td>
</tr>
<tr>
<td>exp2m1</td>
<td>exp2m1</td>
<td>7.12.6.15, F.10.3.15</td>
</tr>
<tr>
<td>exp10</td>
<td>exp10</td>
<td>7.12.6.16, F.10.3.16</td>
</tr>
<tr>
<td>exp10m1</td>
<td>exp10m1</td>
<td>7.12.6.17, F.10.3.17</td>
</tr>
<tr>
<td>log</td>
<td>log</td>
<td>7.12.6.8, F.10.3.3</td>
</tr>
<tr>
<td>log2</td>
<td>log2</td>
<td>7.12.6.10, F.10.3.10</td>
</tr>
<tr>
<td>log10</td>
<td>log10</td>
<td>7.12.6.9, F.10.3.9</td>
</tr>
<tr>
<td>log1p</td>
<td>log1p, logpl</td>
<td>7.12.6.17, F.10.3.17</td>
</tr>
<tr>
<td>log2p1</td>
<td>log2p1</td>
<td>7.12.6.18, F.10.3.18</td>
</tr>
<tr>
<td>log10p1</td>
<td>log10p1</td>
<td>7.12.7.3, F.10.4.3</td>
</tr>
<tr>
<td>hypot</td>
<td>hypot</td>
<td>7.12.7.6, F.10.4.6</td>
</tr>
<tr>
<td>rsqrt</td>
<td>compoundn</td>
<td>7.12.7.7, F.10.4.7</td>
</tr>
<tr>
<td>compound</td>
<td>compoundn</td>
<td>7.12.7.7, F.10.4.7</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Clause</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>rootn</td>
<td>rootn</td>
<td>7.12.7.8, F.10.4.8</td>
</tr>
<tr>
<td>pown</td>
<td>pown</td>
<td>7.12.7.9, F.10.4.9</td>
</tr>
<tr>
<td>pow</td>
<td>pow</td>
<td>7.12.7.4, F.10.4.4</td>
</tr>
<tr>
<td>powr</td>
<td>powr</td>
<td>7.12.7.10, F.10.4.10</td>
</tr>
<tr>
<td>sin</td>
<td>sin</td>
<td>7.12.4.6, F.10.1.6</td>
</tr>
<tr>
<td>cos</td>
<td>cos</td>
<td>7.12.4.5, F.10.1.5</td>
</tr>
<tr>
<td>tan</td>
<td>tan</td>
<td>7.12.4.7, F.10.1.7</td>
</tr>
<tr>
<td>sinPi</td>
<td>sinpi</td>
<td>7.12.4.13, F.10.1.13</td>
</tr>
<tr>
<td>cosPi</td>
<td>cospi</td>
<td>7.12.4.12, F.10.1.12</td>
</tr>
<tr>
<td>tanpi</td>
<td>tanpi</td>
<td>7.12.4.14, F.10.1.14</td>
</tr>
<tr>
<td>asinpi</td>
<td>asinpi</td>
<td>7.12.4.9, F.10.1.9</td>
</tr>
<tr>
<td>acospi</td>
<td>acospi</td>
<td>7.12.4.8, F.10.1.8</td>
</tr>
<tr>
<td>atanPi</td>
<td>atanpi</td>
<td>7.12.4.10, F.10.1.10</td>
</tr>
<tr>
<td>atan2Pi</td>
<td>atan2pi</td>
<td>7.12.4.11, F.10.1.11</td>
</tr>
<tr>
<td>asin</td>
<td>asin</td>
<td>7.12.4.2, F.10.1.2</td>
</tr>
<tr>
<td>acos</td>
<td>acos</td>
<td>7.12.4.1, F.10.1.1</td>
</tr>
<tr>
<td>atan</td>
<td>atan</td>
<td>7.12.4.3, F.10.1.3</td>
</tr>
<tr>
<td>atan2</td>
<td>atan2</td>
<td>7.12.4.4, F.10.1.4</td>
</tr>
<tr>
<td>sinh</td>
<td>sinh</td>
<td>7.12.5.5, F.10.2.5</td>
</tr>
<tr>
<td>cosh</td>
<td>cosh</td>
<td>7.12.5.4, F.10.2.4</td>
</tr>
<tr>
<td>tanh</td>
<td>tanh</td>
<td>7.12.5.6, F.10.2.6</td>
</tr>
<tr>
<td>asinh</td>
<td>asinh</td>
<td>7.12.5.2, F.10.2.2</td>
</tr>
<tr>
<td>acosh</td>
<td>acosh</td>
<td>7.12.5.1, F.10.2.1</td>
</tr>
<tr>
<td>atanh</td>
<td>atanh</td>
<td>7.12.5.3, F.10.2.3</td>
</tr>
<tr>
<td>sum</td>
<td>reduc_sum</td>
<td>7.12.13b.1, F.10.10b.1</td>
</tr>
<tr>
<td>dot</td>
<td>reduc_sumprod</td>
<td>7.12.13b.4, F.10.10b.4</td>
</tr>
<tr>
<td>sumSquare</td>
<td>reduc_sumsq</td>
<td>7.12.13b.3, F.10.10b.3</td>
</tr>
<tr>
<td>sumAbs</td>
<td>reduc_sumabs</td>
<td>7.12.13b.2, F.10.13b.2</td>
</tr>
<tr>
<td>scaledProd</td>
<td>scaled_prod</td>
<td>7.12.13b.5, F.10.10b.5</td>
</tr>
<tr>
<td>scaledProdSum</td>
<td>scaled_prodsq</td>
<td>7.12.13b.6, F.10.10b.6</td>
</tr>
<tr>
<td>scaledProdDiff</td>
<td>scaled_proddiff</td>
<td>7.12.13b.7, F.10.10b.7</td>
</tr>
</tbody>
</table>

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 acospix functions

Synopsis

```c
#include <math.h>

double acospix(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 acospix functions compute the arc cosine of \( x \), divided by \( \pi \), thus measuring the angle in half-revolutions. A domain error occurs for arguments not in the interval \([-1, +1]\).

Returns

[3] The acospix functions return \( \arccos(x)/\pi \), in the interval \([0, 1]\).

7.12.4.9 The asinpix functions

Synopsis

```c
#include <math.h>

double asinpix(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 asinpix functions compute the arc sine of \( x \), divided by \( \pi \), 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 asinpix functions return \( \arcsin(x)/\pi \), 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 atanpid(_DecimalN x);
    _DecimalNx atanpidNx(_DecimalNx x);

Description

[2] The atanpi functions compute the arc tangent of \( x \), divided by \( \pi \), 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 \( \arctan(x) / \pi \), 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 atan2pid(_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 \( \arctan(y/x) / \pi \), 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 cospilN(_DecimalN x);
   _DecimalNx cospilNx(_DecimalNx 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);
   _FloatN sinpifN(_FloatN x);
   _FloatNx sinpifNx(_FloatNx x);
   _DecimalN sinpilN(_DecimalN x);
   _DecimalNx sinpilNx(_DecimalNx x);

Description

[2] The sinpi functions compute 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);
   _FloatN tanpifN(_FloatN x);
   _FloatNx tanpifNx(_FloatNx x);
   _DecimalN tanpilN(_DecimalN x);
   _DecimalNx tanpilNx(_DecimalNx x);
Description

[2] The \texttt{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 \texttt{tanpi} functions return $\tan(\pi \times x)$.

In 7.12.6.9, replace the subclause title:

7.12.6.9 The \texttt{log1p} functions

with:

7.12.6.9 The \texttt{log1p} and \texttt{logpl} functions

In 7.12.6.9#1, append to the Synopsis:

\begin{verbatim}
    double logpl(double x);
    float logplf(float x);
    long double logpll(long double x);
    _FloatN logplfN(_FloatN x);
    _FloatNx logplfNx(_FloatNx x);
    _DecimalN logpldN(_DecimalN x);
    _DecimalNx logpldNx(_DecimalNx x);
\end{verbatim}

In 7.12.6.9#2, replace the first sentence:

The \texttt{log1p} functions compute the base-e (natural) logarithm of 1 plus the argument.

with:

The \texttt{log1p} functions are equivalent to the \texttt{logpl} functions. These functions compute the base-e (natural) logarithm of 1 plus the argument.

Replace 7.12.6.9#3:

[3] The \texttt{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 \texttt{log1p} functions

with:

F.10.3.9 The \texttt{log1p} and \texttt{logpl} 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 $\log_2(1 + x)$.

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 \texttt{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 \texttt{log10p1} functions return \(\log_{10}(1 + x)\).

After 7.12.7.5, insert the following:

7.12.7.6 The \texttt{rsqrt} functions

Synopsis

[1] 
```c
#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 \texttt{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 \texttt{rsqrt} functions return \(1/\sqrt{x}\).

7.12.7.7 The \texttt{compoundn} functions

Synopsis

[1] 
```c
#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 \texttt{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 \((1 + x)^n\).
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 rootnF(_FloatN x, intmax_t n);
    _FloatNx rootnFx(_FloatNx x, intmax_t n);
    _DecimalN rootnD(_DecimalN x, intmax_t n);
    _DecimalNx rootnFx(_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 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 pownF(_FloatN x, intmax_t n);
    _FloatNx pownFx(_FloatNx x, intmax_t n);
    _DecimalN pownD(_DecimalN x, intmax_t n);
    _DecimalNx pownFx(_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


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 powrF(_FloatN x, _FloatN y);
    _FloatNx powrFx(_FloatNx x, _FloatNx y);
    _DecimalN powrD(_DecimalN x, _DecimalN y);
    _DecimalNx powrFx(_DecimalNx x, _DecimalNx y);
**Description**

[2] The `powr` functions compute \( x \) raised to the power \( y \) as \( \exp(y \times \log(x)) \). 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 \( x^y \).

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

<table>
<thead>
<tr>
<th>crexp</th>
<th>crsqrt</th>
<th>cracospi</th>
</tr>
</thead>
<tbody>
<tr>
<td>crexp1</td>
<td>crcompundn</td>
<td>cratanpi</td>
</tr>
<tr>
<td>crexp2</td>
<td>crrootn</td>
<td>cratan2pi</td>
</tr>
<tr>
<td>crexp2ml</td>
<td>crpow</td>
<td>crasin</td>
</tr>
<tr>
<td>crexp10</td>
<td>crlog2</td>
<td>cratan</td>
</tr>
<tr>
<td>crexp10ml</td>
<td>crlog</td>
<td>cratan2</td>
</tr>
<tr>
<td>crlog10</td>
<td>crlog1p</td>
<td>cranh</td>
</tr>
<tr>
<td>crlog10l</td>
<td>crlog2p</td>
<td>cratanh</td>
</tr>
<tr>
<td>crlog10pl</td>
<td>crlog3p</td>
<td>cratanh</td>
</tr>
<tr>
<td>crhypot</td>
<td>crsinpi</td>
<td>cratanh</td>
</tr>
</tbody>
</table>

and the same names suffixed with \( f, l, fN, fNx, dN, \) or \( dNx \) 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 \texttt{asinpi} functions

- \texttt{asinpi}(\pm 0) returns \pm 0.
- \texttt{asinpi}(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.10 The \texttt{atanpi} functions

- \texttt{atanpi}(\pm 0) returns \pm 0.
- \texttt{atanpi}(\pm \infty) returns \pm 1/2.

F.10.1.11 The \texttt{atan2pi} functions

- \texttt{atan2pi}(\pm 0, -0) returns 1.
- \texttt{atan2pi}(\pm 0, +0) returns 0.
- \texttt{atan2pi}(\pm 0, \pm x) returns $\pm 1$ for $x < 0$.
- \texttt{atan2pi}(\pm 0, \pm x) returns $\pm 0$ for $x > 0$.
- \texttt{atan2pi}(\pm y, \pm 0) returns $-1/2$ for $y < 0$.
- \texttt{atan2pi}(\pm y, \pm 0) returns $+1/2$ for $y > 0$.
- \texttt{atan2pi}(\pm y, -\infty) returns $\pm 1$ for finite $y > 0$.
- \texttt{atan2pi}(\pm y, +\infty) returns $\pm 0$ for finite $y > 0$.
- \texttt{atan2pi}(\pm \infty, x) returns $\pm 1/2$ for finite $x$.
- \texttt{atan2pi}(\pm \infty, -\infty) returns $\pm 3/4$ for finite $x$.
- \texttt{atan2pi}(\pm \infty, +\infty) returns $\pm 1/4$ for finite $x$.

F.10.1.12 The \texttt{cospi} functions

- \texttt{cospi}(\pm 0) returns 1.
- \texttt{cospi}(n + 1/2) returns 0, for integers $n$.
- \texttt{cospi}(\pm \infty) returns a NaN and raises the "invalid" floating-point exception.

F.10.1.13 The \texttt{sinpi} functions

- \texttt{sinpi}(\pm 0) returns 0.
- \texttt{sinpi}(\pm n) returns $\pm 0$, for positive integers $n$.
- \texttt{sinpi}(\pm \infty) returns a NaN and raises the "invalid" floating-point exception.

F.10.1.14 The \texttt{tanpi} functions

- \texttt{tanpi}(\pm 0) returns 0.
- \texttt{tanpi}(n) returns $+0$, for positive even and negative odd integers $n$.
- \texttt{tanpi}(n) returns $-0$, for positive odd and negative even integers $n$.
- \texttt{tanpi}(n + 1/2) returns $+\infty$ and raises the "divide-by-zero" floating-point exception, for even integers $n$.
- \texttt{tanpi}(n + 1/2) returns $-\infty$ and raises the "divide-by-zero" floating-point exception, for odd integers $n$.
- \texttt{tanpi}(\pm \infty) 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 ≥ −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 \texttt{\textbf{rootn}} functions

- \texttt{rootn} \((\pm 0, n)\) returns \(+\infty\) and raises the “divide-by-zero” floating-point exception for odd \(n < 0\).
- \texttt{rootn} \((\pm 0, n)\) returns \(+\infty\) and raises the “divide-by-zero” floating-point exception for even \(n < 0\).
- \texttt{rootn} \((\pm 0, n)\) returns +0 for even \(n > 0\).
- \texttt{rootn} \((\pm 0, n)\) returns 0 for odd \(n > 0\).
- \texttt{rootn} \((\pm 0, n)\) is equivalent to \texttt{rootn} \((\pm 0, -n)\) for \(n\) not 0.
- \texttt{rootn} \((x, 0)\) returns a NaN and raises the “invalid” floating-point exception for all \(x\) (including NaN).
- \texttt{rootn} \((x, n)\) returns a NaN and raises the “invalid” floating-point exception for \(x < 0\) and \(n\) even.

F.10.4.9 The \texttt{\textbf{pown}} functions

- \texttt{pown} \((x, 0)\) returns 1 for all \(x\) not a signaling NaN.
- \texttt{pown} \((\pm 0, n)\) returns \(\pm\infty\) and raises the “divide-by-zero” floating-point exception for odd \(n < 0\).
- \texttt{pown} \((\pm 0, n)\) returns \(+\infty\) and raises the “divide-by-zero” floating-point exception for even \(n < 0\).
- \texttt{pown} \((\pm 0, n)\) returns +0 for even \(n > 0\).
- \texttt{pown} \((\pm 0, n)\) returns 0 for odd \(n > 0\).
- \texttt{pown} \((\pm 0, n)\) is equivalent to \texttt{pown} \((\pm 0, -n)\) for \(n\) not 0.

F.10.4.10 The \texttt{\textbf{powr}} functions

- \texttt{powr} \((x, \pm 0)\) returns 1 for finite \(x > 0\).
- \texttt{powr} \((\pm 0, y)\) returns \(+\infty\) and raises the “divide-by-zero” floating-point exception for finite \(y < 0\).
- \texttt{powr} \((\pm 0, -\infty)\) returns \(+\infty\).
- \texttt{powr} \((\pm 0, y)\) returns +0 for \(y > 0\).
- \texttt{powr} \((+1, y)\) returns 1 for finite \(y\).
- \texttt{powr} \((x, y)\) returns a NaN and raises the “invalid” floating-point exception for \(x < 0\).
- \texttt{powr} \((\pm 0, \pm 0)\) returns a NaN and raises the “invalid” floating-point exception.
- \texttt{powr} \((+\infty, \pm 0)\) returns a NaN and raises the “invalid” floating-point exception.
- \texttt{powr} \((1, \pm 0)\) returns a NaN and raises the “invalid” floating-point exception.

8 Reduction functions in \texttt{\textbf{<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:

30 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 \texttt{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: \( \sum_{i=0}^{n-1} p[i] \). A range error may occur.

Returns


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: \( \sum_{i=0}^{n-1} |p[i]| \). A range error may occur.

Returns


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_sumsqf(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_sumsqfNx(size_t n, const _DecimalN p[static n]);
  _DecimalNx reduc_sumsqfNx(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: \( \sum_{i=0}^{n-1} (p[i] \times p[i]) \). A range error may occur.

Returns


7.12.13b.4 The reduc_sumprod functions

Synopsis

[1] 
```c
#include <math.h>

double reduc_sumprod(size_t n, const double p[static n],
const double q[static n]);
float reduc_sumprod(x[size_t n, const float p[static n],
const float q[static n]);
long double reduc_sumprod(l(size_t n, const long double p[static n],
const long double q[static n]);
_FloatN reduc_sumprodN(size_t n, const _FloatN p[static n],
const _FloatN q[static n]);
_FloatNx reduc_sumprodNx(size_t n, const _FloatNx p[static n],
const _FloatNx q[static n]);
DecimalN reduc_sumprodD(size_t n, const _DecimalN p[static n],
const _DecimalN q[static n]);
DecimalNx reduc_sumprodDx(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: \( \sum_{i=0}^{n-1} (p[i] \times q[i]) \). A range error may occur.

Returns


7.12.13b.5 The scaled_prod functions

Synopsis

[1] 
```c
#include <math.h>

double scaled_prod(size_t n, const double p[static n],
inint_t * restrict sfptr);
float scaled_prodf(size_t n, const float p[static n],
inint_t * restrict sfptr);
long double scaled_prodl(size_t n, const long double p[static n],
inint_t * restrict sfptr);
_FloatN scaled_prodfN(size_t n, const _FloatN p[static n],
inint_t * restrict sfptr);
_FloatNx scaled_prodfNx(size_t n, const _FloatNx p[static n],
inint_t * restrict sfptr);
DecimalN scaled_prodlD(size_t n, const _DecimalN p[static n],
inint_t * restrict sfptr);
DecimalNx scaled_prodlDx(size_t n, const _DecimalNx p[static n],
inint_t * restrict sfptr);
```

Description

[2] The scaled_prod functions compute the dot product of the sequences of members of the arrays p and q: \( \sum_{i=0}^{n-1} (p[i] \times q[i]) \). A range error may occur.

Returns

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 \( pr \times b^sf = \prod_{i=0}^{n-1} p[i] \), 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>

\[
\begin{align*}
\text{double scaled_prodsum(size_t n, const double p[static n],} \\
\text{const double q[static n], intmax_t * restrict sfptr);} \\
\text{float scaled_prodsumf(size_t n, const float p[static n],} \\
\text{const float q[static n], intmax_t * restrict sfptr);} \\
\text{long double scaled_prodsuml(size_t n, const long double p[static n],} \\
\text{const long double q[static n], intmax_t * restrict sfptr);} \\
\text{_FloatN scaled_prodsumfn(size_t n, const _FloatN p[static n],} \\
\text{const _FloatN q[static n], intmax_t * restrict sfptr);} \\
\text{_FloatNx scaled_prodsumfnx(size_t n, const _FloatNx p[static n],} \\
\text{const _FloatNx q[static n], intmax_t * restrict sfptr);} \\
\text{_DecimalN scaled_prodsumdn(size_t n, const _DecimalN p[static n],} \\
\text{const _DecimalN q[static n], intmax_t * restrict sfptr);} \\
\text{_DecimalNx scaled_prodsumdnx(size_t n, const _DecimalNx p[static n],} \\
\text{const _DecimalNx q[static n], intmax_t * restrict sfptr);} \\
\end{align*}
\]

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_proddiffll(size_t n, const long double p[static n],
const long double q[static n], intmax_t * restrict sfptr);
_FloatN scaled_proddiffN(size_t n, const _FloatN p[static n],
const _FloatN q[static n], intmax_t * restrict sfptr);
_FloatNx scaled_proddiffNx(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_proddiffNx(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 \( pr \times b^sf = \prod_{i=0}^{n-1} (p[i] - q[i]) \), 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 \texttt{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 \texttt{reduc_sum}, \texttt{reduc_sumabs}, \texttt{reduc_sumsq}, and \texttt{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 \texttt{scaled_prod}, \texttt{scaled_prodsum}, and \texttt{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

- \texttt{reduc_sum(n, p)} returns a NaN if any member of array \( p \) is a NaN.
- \texttt{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, \texttt{reduc_sum(n, p)} returns \( \pm\infty \) if the members of \( p \) include one or more infinities \( \pm\infty \) (with the same sign).
F.10.10.2 The reduc_sumabs functions

- `reduc_sumabs(n, p)` returns $+\infty$ 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.10.10.3 The reduc_sumsq functions

- `reduc_sumsq(n, p)` returns $+\infty$ 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.10.10.4 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.
- Otherwise, `reduc_sumprod(n, p, q)` returns a NaN and raises the “invalid” floating-point exception if any two factors (each of which is a sum) are of exactly the same sign.
- Otherwise, `reduc_sumprod(n, p, q)` returns a NaN if any member of array `p` or `q` is a NaN.

F.10.10.5 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 a NaN if any member of array `p` is an infinity.
- Otherwise, `scaled_prod(n, p, sfptr)` returns a NaN and raises the “invalid” floating-point exception if any factor is an exact infinity.

F.10.10.6 The scaled_prodsum functions

- `scaled_prodsum(n, p, q, sfptr)` returns a NaN if any member of array `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 has a zero and an infinite factor.
- Otherwise, `scaled_prodsum(n, p, q, sfptr)` returns a NaN if any factor is an exact zero.
- Otherwise, `scaled_prodsum(n, p, q, 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.10.10.7 The scaled_proddiff functions

- `scaled_proddiff(n, p, q, sfptr)` returns a NaN if any member of array `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 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:

5 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

```
  cexp2ml cexp10 cexp10ml clogpl clog2pl clog10pl
  crsqrtr ccompounrd crootn cpownr 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:

```
exp2ml exp10 exp10ml log10 log2pl log10pl
rsqrt compoundn rootn powr acospi
asinpi atanpi atan2pi cospi sinpi tanpi
```
Bibliography


