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
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>iv</td>
</tr>
<tr>
<td>Introduction</td>
<td>vi</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>2</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>5</td>
</tr>
<tr>
<td>7 Mathematical functions in <code>&lt;math.h&gt;</code></td>
<td>6</td>
</tr>
<tr>
<td>8 Reduction functions in <code>&lt;math.h&gt;</code></td>
<td>19</td>
</tr>
<tr>
<td>9 Future directions for <code>&lt;complex.h&gt;</code></td>
<td>26</td>
</tr>
<tr>
<td>10 Type-generic macros <code>&lt;tgmath.h&gt;</code></td>
<td>27</td>
</tr>
<tr>
<td>11 Constant rounding modes <code>&lt;fenv.h&gt;</code></td>
<td>27</td>
</tr>
<tr>
<td>Bibliography</td>
<td>31</td>
</tr>
</tbody>
</table>
ISO/IEC TS 18661-4:2015(E)

Foreword

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

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

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

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

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

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

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

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

The following part is under preparation:

— Part 5: Supplementary attributes


iv © ISO/IEC 2015 – All rights reserved
Introduction

Background

IEC 60559 floating-point standard

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

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

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

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

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

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

6. Enable rather than preclude further refinements and extensions.

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

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

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

— context – status flags for detecting exceptional conditions (invalid operation, division by zero, overflow, underflow, and inexact) and controls for choosing different rounding methods
The ISO/IEC/IEEE 60559:2011 international standard is equivalent to the IEEE 754-2008 standard for floating-point arithmetic, which is a major revision to IEEE 754-1985.

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

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

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

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

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

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

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

**C support for IEC 60559**

The C standard specifies floating-point arithmetic using an abstract model. The representation of a floating-point number is specified in an abstract form where the constituent components (sign, exponent, significand) of the representation are defined but not the internals of these components. In particular, the exponent range, significand size, and the base (or radix) are implementation-defined. This allows flexibility for an implementation to take advantage of its underlying hardware architecture. Furthermore, certain behaviors of operations are also implementation-defined, for example in the area of handling of special numbers and in exceptions.
The reason for this approach is historical. At the time when C was first standardized, before the floating-point standard was established, there were various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would have made most of the existing implementations at the time not conforming.


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


Purpose

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

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


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

Additional background on supplementary functions

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

ISO/IEC/IEEE 60559:2011 specifies and recommends a more extensive set of mathematical operations than C11 provides. The IEC 60559 specification is generally consistent with C11, though it adds requirements for symmetry and antisymmetry. This part of ISO/IEC TS 18661 extends the specification in Library subclause 7.12 Mathematics to include the complete set of IEC 60559 mathematical operations. For implementations conforming to annex F, it also requires full IEC 60559 semantics, including symmetry and antisymmetry properties.
IEC 60559 requires correct rounding for its required operations (squareRoot, fusedMultiplyAdd, etc.), and recommends correct rounding for its recommended mathematical operations. This part of ISO/IEC TS 18661 reserves identifiers, with CR prefixes, for C functions corresponding to correctly rounded versions of the IEC 60559 mathematical operations, which may be provided at the option of the implementation. For example, the identifier crexp is reserved for a correctly rounded version of the exp function.

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

Part 4:
Supplementary functions

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

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

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

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

c) it defines __STDC_IEC_60559_FUNCS__ to 201506L.

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


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

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


4.1


5 C standard conformance

5.1 Freestanding implementations

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

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 201506L, 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:

exp2ml rootnf sinh
exp2mlf rootnl tanh
exp2mll pown tanpi
exp10 pownf tanpif
exp10f pownl tanpil
exp10l powr reduc_sum
exp10ml powrf reduc_sumf
exp10mlf powr1 reduc_sum1
exp10mll acospi reduc_sumabs
logp1 acospf reduc_sumabsf
logpf acospl reduc_sumabs1
```

[1f] 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 <cmath.h> is first included:

exp2ml rootnf sinh
exp2mlf rootnl tanh
exp2mll pown tanpi
exp10 pownf tanpif
exp10f pownl tanpil
exp10l powr reduc_sum
exp10ml powrf reduc_sumf
exp10mlf powr1 reduc_sum1
exp10mll acospi reduc_sumabs
logp1 acospf reduc_sumabsf
logpf acospl reduc_sumabs1
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	powdN
tanpidN
exp10dN	powrdN
reduc_sumdN
exp10mldN	acospidN
reduc_sumqdN
acospidN
logp1dN	asinpidN
reduc_sumsqdN
log2p1dN	atanpidN
reduc_sumproddN
log10p1dN	atan2pidN
scaled_proddN
rsqrtdN
cospidN
scaled_prodsimdN
compoundndN
cospidN
scaled_proddiffdN
rootndN/nsinpif
scaled_proddifff

[1f] 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
acospifN
logp1fN	asinpifN
reduc_sumsqfN
log2p1fN	atanpifN
reduc_sumprodfN
log10p1fN	atan2pifN
scaled_prodfN
rsqrtf
cospifN
scaled_prodsufm
compoundnfN
cospifN
scaled_proddifff
rootnfN
tsinfN
scaled_proddiffff

[1g]
for supported types \_Float\_\text{\text{\textbar}N\text{\textbar}}:

| exp2m1f\text{\textbar}N\text{\textbar} | pownf\text{\textbar}N\text{\textbar} | tanpif\text{\textbar}N\text{\textbar} |
| exp10f\text{\textbar}N\text{\textbar} | powrf\text{\textbar}N\text{\textbar} | reduc\_sumf\text{\textbar}N\text{\textbar} |
| exp10m1f\text{\textbar}N\text{\textbar} | acospif\text{\textbar}N\text{\textbar} | reduc\_sumabsf\text{\textbar}N\text{\textbar} |
| logp1f\text{\textbar}N\text{\textbar} | asinpif\text{\textbar}N\text{\textbar} | reduc\_sums\text{\textbar}N\text{\textbar} |
| log2p1f\text{\textbar}N\text{\textbar} | atanpif\text{\textbar}N\text{\textbar} | reduc\_sumprodf\text{\textbar}N\text{\textbar} |
| log10p1f\text{\textbar}N\text{\textbar} | atan2pif\text{\textbar}N\text{\textbar} | scaled\_prodf\text{\textbar}N\text{\textbar} |
| rsqrtf\text{\textbar}N\text{\textbar} | cospif\text{\textbar}N\text{\textbar} | scaled\_prodsumf\text{\textbar}N\text{\textbar} |
| compoundnf\text{\textbar}N\text{\textbar} | sinpif\text{\textbar}N\text{\textbar} | scaled\_proddifff\text{\textbar}N\text{\textbar} |
| rootnf\text{\textbar}N\text{\textbar} |

for supported types \_Decimal\_\text{\text{\textbar}N\text{\textbar}}, where \text{\text{\textbar}N\text{\textbar}} \neq 32, 64, and 128:

| exp2m1d\text{\textbar}N\text{\textbar} | pownd\text{\textbar}N\text{\textbar} | tanpidd\text{\textbar}N\text{\textbar} |
| exp10d\text{\textbar}N\text{\textbar} | powrd\text{\textbar}N\text{\textbar} | reduc\_sumd\text{\textbar}N\text{\textbar} |
| exp10m1d\text{\textbar}N\text{\textbar} | acospid\text{\textbar}N\text{\textbar} | reduc\_sumabsd\text{\textbar}N\text{\textbar} |
| logp1d\text{\textbar}N\text{\textbar} | asinpid\text{\textbar}N\text{\textbar} | reduc\_sumsqd\text{\textbar}N\text{\textbar} |
| log2p1d\text{\textbar}N\text{\textbar} | atanpidd\text{\textbar}N\text{\textbar} | reduc\_sumprodd\text{\textbar}N\text{\textbar} |
| log10p1d\text{\textbar}N\text{\textbar} | atan2pidd\text{\textbar}N\text{\textbar} | scaled\_prodd\text{\textbar}N\text{\textbar} |
| rsqrd\text{\textbar}N\text{\textbar} | cospidd\text{\textbar}N\text{\textbar} | scaled\_prodsumd\text{\textbar}N\text{\textbar} |
| compoundnd\text{\textbar}N\text{\textbar} | sinpidd\text{\textbar}N\text{\textbar} | scaled\_proddiffd\text{\textbar}N\text{\textbar} |
| rootnd\text{\textbar}N\text{\textbar} |

for supported types \_Decimal\_\text{\text{\textbar}N\text{\textbar}}:

| exp2m1d\text{\textbar}N\text{\textbar} | pownd\text{\textbar}N\text{\textbar} | tanpidd\text{\textbar}N\text{\textbar} |
| exp10d\text{\textbar}N\text{\textbar} | powrd\text{\textbar}N\text{\textbar} | reduc\_sumd\text{\textbar}N\text{\textbar} |
| exp10m1d\text{\textbar}N\text{\textbar} | acospid\text{\textbar}N\text{\textbar} | reduc\_sumabsd\text{\textbar}N\text{\textbar} |
| logp1d\text{\textbar}N\text{\textbar} | asinpidd\text{\textbar}N\text{\textbar} | reduc\_sumsqd\text{\textbar}N\text{\textbar} |
| log2p1d\text{\textbar}N\text{\textbar} | atanpidd\text{\textbar}N\text{\textbar} | reduc\_sumprodd\text{\textbar}N\text{\textbar} |
| log10p1d\text{\textbar}N\text{\textbar} | atan2pidd\text{\textbar}N\text{\textbar} | scaled\_prodd\text{\textbar}N\text{\textbar} |
| rsqrd\text{\textbar}N\text{\textbar} | cospidd\text{\textbar}N\text{\textbar} | scaled\_prodsumd\text{\textbar}N\text{\textbar} |
| compoundnd\text{\textbar}N\text{\textbar} | sinpidd\text{\textbar}N\text{\textbar} | scaled\_proddiffd\text{\textbar}N\text{\textbar} |
| rootnd\text{\textbar}N\text{\textbar} |

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 \textlangle\text{tgmath.h}\textrangle\ is first included:

| exp2m1 \text{\textbar} | rsqrt \text{\textbar} | asinpi |
| exp10 \text{\textbar} | compound \text{\textbar} | atanpi |
| exp10m1 \text{\textbar} | rootn \text{\textbar} | atan2pi |
| logp1 \text{\textbar} | pown \text{\textbar} | cospi |
| log2p1 \text{\textbar} | powr \text{\textbar} | sinpi |
| log10p1 \text{\textbar} | acospit \text{\textbar} | 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>exp2</td>
<td>exp2</td>
<td>7.12.6.2, F.10.3.2</td>
</tr>
<tr>
<td>exp2m1</td>
<td>exp2m1</td>
<td>7.12.6.14, F.10.3.14</td>
</tr>
<tr>
<td>exp10</td>
<td>exp10</td>
<td>7.12.6.15, F.10.3.15</td>
</tr>
<tr>
<td>exp10m1</td>
<td>exp10m1</td>
<td>7.12.6.16, F.10.3.16</td>
</tr>
<tr>
<td>log</td>
<td>log</td>
<td>7.12.6.7, F.10.3.7</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>7.12.6.8, F.10.3.8</td>
</tr>
<tr>
<td>log1p</td>
<td>log1p, log1p</td>
<td>7.12.6.9, F.10.3.9</td>
</tr>
<tr>
<td>log2p1</td>
<td>log2p1</td>
<td>7.12.6.17, F.10.3.17</td>
</tr>
<tr>
<td>log10p1</td>
<td>log10p1</td>
<td>7.12.6.18, F.10.3.18</td>
</tr>
<tr>
<td>hypot</td>
<td>hypot</td>
<td>7.12.7.3, F.10.4.3</td>
</tr>
<tr>
<td>rSqrt</td>
<td>rsqrt</td>
<td>7.12.7.6, F.10.4.6</td>
</tr>
<tr>
<td>compound</td>
<td>compounldn</td>
<td>7.12.7.7, F.10.4.7</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.10.1.14</td>
</tr>
<tr>
<td>acospi</td>
<td>acospi</td>
<td>7.12.4.8, F.10.1.14</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>
</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 support the symmetry and antisymmetry properties that IEC 60559 specifies for mathematical functions.

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);
   _FloatN x acospifN(_FloatN x);
   _DecimalN acospidN(_DecimalN x);
   _DecimalN x acospidNx(_DecimalN x);

Description

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

Returns

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

Synopsis

[1] #include <math.h>
   double asinpi(double x);
   float asinpif(float x);
   long double asinpil(long double x);
   _FloatN asinpifN(_FloatN x);
   _FloatNx asinpifNx(_FloatNx x);
   _DecimalN asinpifD(_DecimalN x);
   _DecimalNx asinpifDx(_DecimalNx x);

Description

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

Returns

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

7.12.4.10 The atanpi functions

Synopsis

[1] #include <math.h>
   double atanpi(double x);
   float atanpif(float x);
   long double atanpil(long double x);
   _FloatN atanpifN(_FloatN x);
   _FloatNx atanpifNx(_FloatNx x);
   _DecimalN atanpifD(_DecimalN x);
   _DecimalNx atanpifDx(_DecimalNx x);

Description

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

Returns

[3] The atanpi functions return arctan(x) / π, in the interval [-1/2, +1/2].
7.12.4.11 The atan2pi functions

Synopsis

[1] #include <math.h>
    double atan2pi(double y, double x);
    float atan2pif(float y, float x);
    long double atan2pil(long double y, long double x);
    _Float N atan2pifN(_Float N y, _Float N x);
    _Float N atan2pifNx(_Float Nx y, _Float Nx x);
    _Decimal N atan2pilN(_Decimal N y, _Decimal N x);
    _Decimal N atan2pilNx(_Decimal Nx y, _Decimal Nx 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, \(\text{atan2pi}(y/x) / \pi\), 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 \(\text{atan2pi}\) 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);
    _Float N cospifN(_Float N x);
    _Float N cospifNx(_Float Nx x);
    _Decimal N cospilN(_Decimal N x);
    _Decimal N cospilNx(_Decimal Nx x);

Description

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

Returns

[3] The \(\text{cospi}\) 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 sinpidN(DecimalN x);
    DecimalNx sinpidNx(DecimalNx x);

Description

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

Returns

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

7.12.4.14 The tanpi functions

Synopsis

[1] #include <math.h>
    double tanpi(double x);
    float tanpif(float x);
    long double tanpil(long double x);
    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
7.12.6.9 The \texttt{log1p} and \texttt{logp1} functions

In 7.12.6.9\#1, append to the Synopsis:

\begin{verbatim}
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
_FloatN log1fN(_FloatN x);
_FloatNx log1fNx(_FloatNx x);
_DecimalN log1dN(_DecimalN x);
_DecimalNx log1dNx(_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 \(1 + x\).

with:

The \texttt{log1p} functions are equivalent to the \texttt{logp1} functions. These functions compute the base-\(e\) (natural) logarithm of \(1 + x\).

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:

\textbf{F.10.3.9 The log1p functions}

with:

\textbf{F.10.3.9 The log1p and logp1 functions}

After 7.12.6.13, insert the following:

7.12.6.14 The \texttt{exp2m1} functions

Synopsis

\begin{verbatim}
#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);
\end{verbatim}
Description

[2] The \texttt{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 \texttt{exp2m1} functions return \(2^x - 1\).

7.12.6.15 The \texttt{exp10} functions

Synopsis

[1]

```c
#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 \texttt{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 \texttt{exp10} functions return \(10^x\).

7.12.6.16 The \texttt{exp10m1} functions

Synopsis

[1]

```c
#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 \texttt{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 \texttt{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 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 \( \log_{10}(1 + x) \).
After 7.12.7.5, insert the following:

### 7.12.7.6 The `rsqrt` functions

#### Synopsis

```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

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

The `rsqrt` functions return $1 / \sqrt{x}$.

### 7.12.7.7 The `compoundn` functions

#### Synopsis

```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 compoundfN(_FloatN x, intmax_t n);
_FloatNx compoundfNx(_FloatNx x, intmax_t n);
_DecimalN compounddN(_DecimalN x, intmax_t n);
_DecimalNx compounddNx(_DecimalNx x, intmax_t n);
```

#### Description

The `compoundn` functions compute $1 + x$, raised to the power $n$. A domain error occurs if $x < -1$. A range error may occur if $n$ is too large, depending on $x$. A pole error may occur if $x$ equals $-1$ and $n < 0$.

#### Returns

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 rootnxD(DecimalNx x, intmax_t n);

Description

[2] The rootn functions compute the principal nth 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 pownxD(DecimalNx x, intmax_t n);

Description

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

Returns

[3] The pown functions return \(x^n\).
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 \( \exp(y \times \log(x)) \). 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

|creexp       | crrsqrt      | cracospi     |
crepml      | crcompoundn  | cratanpi     |
crexp2      | crrrootn     | cratan2pi    |
crexp2ml    | crpown       | crasin       |
crexp10     | crpow        | cracos       |
crexp10ml   | crpowr       | cratan       |
crlog       | crsin        | cratan2      |
crlog2      | crcos        | crsinh       |
crlog10     | crtan        | crcosh       |
crlog1p     | crsinpi      | crtanh       |
crlogpl     | crrcospi     | crasinpi     |
crlog2pl    | crtanpi      | cracosh      |
crlog10pl   | crasinpi     | cratanh      |
crhypt      |               |               |

and the same names suffixed with \_f\, \_l\, \_fN\, \_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 \texttt{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 <\texttt{math.h}> whose mathematical counterpart is symmetric (even), \( f(x) = f(-x) \) for all rounding modes and for all \( x \) in the (valid) domain of the function. For each single-argument function \( f \) in <\texttt{math.h}> whose mathematical counterpart is antisymmetric (odd), \( f(-x) = -f(x) \) for the IEC 60559 rounding modes \texttt{roundTiesToEven}, \texttt{roundTiesToAway}, and \texttt{roundTowardZero}, and for all \( x \) in the (valid) domain of the function. The \texttt{atan2} and \texttt{atan2pi} functions are odd in their first argument.

After F.10.1.7, insert the following:

\textbf{F.10.1.8 The \texttt{acospi} functions}

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

\textbf{F.10.1.9 The \texttt{asinpi} functions}

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

\textbf{F.10.1.10 The \texttt{atanpi} functions}

- \texttt{atanpi}(±0) returns ±0.
- \texttt{atanpi}(±∞) returns ±1/2.

\textbf{F.10.1.11 The \texttt{atan2pi} functions}

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

\textbf{F.10.1.12 The \texttt{cospi} functions}

- \texttt{cospi}(±0) returns 1.
- \texttt{cospi}(n + 1/2) returns +0, for integers \( n \).
- \texttt{cospi}(±∞) returns a NaN and raises the "invalid" floating-point exception.
F.10.1.13 The sinpi functions

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

F.10.1.14 The tanpi functions

- \( \text{tanpi}(\pm 0) \) returns \( \pm 0 \).
- \( \text{tanpi}(n) \) returns \( +0 \), for positive even and negative odd integers \( n \).
- \( \text{tanpi}(n) \) returns \( -0 \), for positive odd and negative even integers \( n \).
- \( \text{tanpi}(n + 1/2) \) returns \( +\infty \) and raises the “divide-by-zero” floating-point exception, for even integers \( n \).
- \( \text{tanpi}(n + 1/2) \) returns \( -\infty \) and raises the “divide-by-zero” floating-point exception, for odd integers \( n \).
- \( \text{tanpi}(\pm \infty) \) 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

- \( \text{exp2m1}(\pm 0) \) returns \( \pm 0 \).
- \( \text{exp2m1}(-\infty) \) returns \( -1 \).
- \( \text{exp2m1}(+\infty) \) returns \( +\infty \).

F.10.3.15 The exp10 functions

- \( \text{exp10}(\pm 0) \) returns \( 1 \).
- \( \text{exp10}(-\infty) \) returns \( +0 \).
- \( \text{exp10}(+\infty) \) returns \( +\infty \).

F.10.3.16 The exp10m1 functions

- \( \text{exp10m1}(\pm 0) \) returns \( \pm 0 \).
- \( \text{exp10m1}(-\infty) \) returns \( -1 \).
- \( \text{exp10m1}(+\infty) \) returns \( +\infty \).

F.10.3.17 The log2p1 functions

- \( \text{log2p1}(\pm 0) \) returns \( \pm 0 \).
- \( \text{log2p1}(-1) \) returns \( -\infty \) and raises the “divide-by-zero” floating-point exception.
- \( \text{log2p1}(x) \) returns a NaN and raises the “invalid” floating-point exception for \( x < -1 \).
- \( \text{log2p1}(+\infty) \) returns \( +\infty \).

F.10.3.18 The log10p1 functions

- \( \text{log10p1}(\pm 0) \) returns \( \pm 0 \).
- \( \text{log10p1}(-1) \) returns \( -\infty \) and raises the “divide-by-zero” floating-point exception.
- \( \text{log10p1}(x) \) returns a NaN and raises the “invalid” floating-point exception for \( x < -1 \).
- \( \text{log10p1}(+\infty) \) returns \( +\infty \).
After F.10.4.5, insert the following:

**F.10.4.6 The \texttt{rsqrt} functions**

- \texttt{rsqrt}(±0) returns ±∞ and raises the “divide-by-zero” floating-point exception.
- \texttt{rsqrt}(x) returns a NaN and raises the “invalid” floating-point exception for \(x < 0\).
- \texttt{rsqrt}(+∞) returns +0.

**F.10.4.7 The \texttt{compoundn} functions**

- \texttt{compoundn}(x, 0) returns 1 for \(x ≥ -1\) or \(x\) a NaN.
- \texttt{compoundn}(x, n) returns a NaN and raises the “invalid” floating-point exception for \(x < -1\).
- \texttt{compoundn}(-1, n) returns +∞ and raises the divide-by-zero floating-point exception for \(n < 0\).
- \texttt{compoundn}(-1, n) returns +0 for \(n > 0\).

**F.10.4.8 The \texttt{rootn} functions**

- \texttt{rootn}(±0, n) returns ±∞ and raises the “divide-by-zero” floating-point exception for odd \(n < 0\).
- \texttt{rootn}(±0, n) returns +∞ and raises the “divide-by-zero” floating-point exception for even \(n < 0\).
- \texttt{rootn}(±0, n) returns +0 for even \(n > 0\).
- \texttt{rootn}(±0, n) returns ±0 for odd \(n > 0\).
- \texttt{rootn}(±∞, n) is equivalent to \texttt{rootn}(±0, -n) for \(n\) not 0, except that the “divide-by-zero” floating-point exception is not raised.
- \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{pown} functions**

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

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

- \texttt{powr}(x, ±0) returns 1 for finite \(x > 0\).
- \texttt{powr}(±0, y) returns +∞ and raises the “divide-by-zero” floating-point exception for finite \(y < 0\).
- \texttt{powr}(±0, −∞) returns +∞.
- \texttt{powr}(±0, y) returns +0 for \(y > 0\).
- \texttt{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>
#include `<stddef.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

7.12.13b.2 The reduc_sumabs functions

Synopsis
[1] #include <math.h>
    #include <stddef.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_sumabs1(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: Σ_{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>
    #include <stddef.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_sumsq1(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: Σ_{i=0,n-1}(p[i] × p[i]). A range error may occur.

Returns
7.12.13b.4 The `reduc_sumprod` functions

Synopsis

[1] #include <math.h>
#include <stddef.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_sumprodll(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`: $\sum_{i=0}^{n-1} (p[i] \times q[i])$. A range error may occur.

Returns

7.12.13b.5 The scaled_prod functions

Synopsis

[1] #include <math.h>
#include <stddef.h>
#include <stdint.h>

double scaled_prod(size_t n,  
                const double p[static restrict n],  
                intmax_t * restrict sfptr);
float scaled_prodf(size_t n,    
                   const float p[static restrict n],  
                   intmax_t * restrict sfptr);
long double scaled_prodl(size_t n,    
                        const long double p[static restrict n],  
                        intmax_t * restrict sfptr);
_FloatN scaled_prodfN(size_t n,    
                      const _FloatN p[static restrict n],  
                      intmax_t * restrict sfptr);
_FloatNx scaled_prodfNx(size_t n,    
                        const _FloatNx p[static restrict n],  
                        intmax_t * restrict sfptr);
_DecimalN scaled_proddN(size_t n,  
                        const _DecimalN p[static restrict n],  
                        intmax_t * restrict sfptr);
_DecimalNx scaled_proddNx(size_t n,    
                        const _DecimalNx p[static restrict 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 \( pr \times b^{sf} = \prod_{i=0}^{n-1} p[i] \), 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 \( \text{intmax}_t \) 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>
#include <stddef.h>
#include <stdint.h>
double scaled_prodsum(size_t n,
   const double p[static restrict n],
   const double q[static restrict n],
   intmax_t * restrict sfptr);
float scaled_prodsumf(size_t n,
   const float p[static restrict n],
   const float q[static restrict n],
   intmax_t * restrict sfptr);
long double scaled_prodsuml(size_t n,
   const long double p[static restrict n],
   const long double q[static restrict n],
   intmax_t * restrict sfptr);
_FloatN scaled_prodsumfN(size_t n,
   const _FloatN p[static restrict n],
   const _FloatN q[static restrict n],
   intmax_t * restrict sfptr);
_FloatNx scaled_prodsumfNx(size_t n,
   const _FloatNx p[static restrict n],
   const _FloatNx q[static restrict n],
   intmax_t * restrict sfptr);
DecimalN scaled_prodsumdN(size_t n,
   const DecimalN p[static restrict n],
   const DecimalN q[static restrict n],
   intmax_t * restrict sfptr);
DecimalNx scaled_prodsumdNx(size_t n,
   const DecimalNx p[static restrict n],
   const DecimalNx q[static restrict 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 \( pr \times b^{sf} = \prod_{i=0,n-1}(p[i] + q[i]) \), 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 \text{intmax_t} \ type. These functions should not cause a range error.

Returns

[3] The scaled_prodsum functions return the computed scaled product \( pr \).
7.12.13.7 The scaled_proddiff functions

Synopsis

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

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, unless explicitly specified otherwise.

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 $\pm\infty$ if the members of `p` include one or more infinities $\pm\infty$ (with the same sign).

F.10.10b.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.10.10b.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.10.10b.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, q)` returns $\pm\infty$ if one or more of the products are (exactly) $\pm\infty$ (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, 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.10.10b.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, 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:

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  cexp10  cexp10m1  clogpl  clog2pl  clog10pl  crsqrtn  ccompoundn  crootn  cwpow  cwpowr  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
logpl   pown  cospi  
log2pl  powr  sinpi
log10pl  acospn  tanpi
```

11 Constant rounding modes `<fenv.h>`

As IEC 60559 operations, the `<math.h>` functions introduced in this part of ISO/IEC TS 18661 are subject to IEC 60559 constant rounding-direction attributes. The following changes to C11 + TS18661-1 + TS18661-2 + TS18661-3 add these new functions to the set of functions affected by constant rounding modes in `<fenv.h>`.

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

In 7.6.1a#4, replace the table:

<table>
<thead>
<tr>
<th>Header</th>
<th>Function groups</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td><code>acos, asin, atan, atan2</code></td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td><code>cos, sin, tan</code></td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td><code>acosh, asinh, atanh</code></td>
</tr>
</tbody>
</table>
<math.h>  
cosh, sinh, tanh
<math.h>  
exp, exp2, expm1
<math.h>  
log, log10, log1p, log2
<math.h>  
scalbn, scalbln, ldexp
<math.h>  
cbrt, hypot, pow, sqrt
<math.h>  
exp, erfc
<math.h>  
lgamma, tgamma
<math.h>  
rnt, nearbyint, lrint, llrint
<math.h>  
fma
<math.h>  
fadd, daddl, fsub, dsbln, fmul, dmull, fd1v, dd1v, 
ffma, dfmal, fsqrt, dsqrt1
<stdlib.h>  
 atof, strfromd, strfromf, strfroml, strtod, 
 strtof, strtol
<wchar.h>  
wctod, wcstof, wcstold
<stdio.h>  
 printf and scanf families
<wchar.h>  
wprintf and wscanf families

with:

<table>
<thead>
<tr>
<th>Header</th>
<th>Function groups</th>
</tr>
</thead>
</table>
| <math.h> | acos, acospi, asin, asinpi, atan, atan2, atan2pi, 
|          | atanpi          |
| <math.h> | cos, cospi, sin, sinpi, tan, tanpi |
| <math.h> | acosh, asinh, atanh |
| <math.h> | cos, sinh, tanh |
| <math.h> | exp, exp10, exp10l, exp2, exp2l, expm1 |
| <math.h> | log, log10, log10p1, log1p, log2, log2p1, logp1 |
| <math.h> | ldexp, scalbln, scalbn |
| <math.h> | cbrt, compoundn, hypot, pow, pown, powr, rootn, 
|          | rsqrt, sqrt |
| <math.h> | erf, erfc |
| <math.h> | lgamma, tgamma |
| <math.h> | llrint, lrint, nearbyint, rint |
| <math.h> | fma |
| <math.h> | fadd, dd1v, dfmal, dmull, dsqrt1, dsbln, fadd, 
|          | fd1v, ffma, fm1, fsqrt, fsub |
| <math.h> | reduc_sum, reduc_sumabs, reduc_sumprod, 
|          | reduc sumsq, scaled prod, scaled_proddiff, 
|          | scaled prodsun |
| <stdlib.h> | atof, strfromd, strfromf, strfroml, strtod, 
|          | strtof, strtol |
| <wchar.h> | wctod, wcstof, wcstold |
| <stdio.h> | printf and scanf families |
| <wchar.h> | wprintf and wscanf families |
In 7.6.1b#2, replace the table:

<table>
<thead>
<tr>
<th>Header</th>
<th>Function groups</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>acosdN, asindN, atandN, atan2dN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>cosdN, sindN, tandN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>acoshdN, asinhN, atanhN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>coshdN, sinhN, tanhdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>expdN, exp2dN, expmldN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>logdN, log10dN, log1pdN, log2dN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>scalbndN, scalblndN, ldexpdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>cbrtN, hypotN, powdN, sqrdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>erfdN, erfcdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>lgammadN, lgammadN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>rintN, nearbyintN, lrintN, llrintN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>quantizedN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>fdimdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>fmadN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>dmaddN, dmsubdN, dmuldN, ddivdN, dMfmadN, dMsqrdN</td>
</tr>
<tr>
<td><code>&lt;stdlib.h&gt;</code></td>
<td>strfromdN, strtodN</td>
</tr>
<tr>
<td><code>&lt;wchar.h&gt;</code></td>
<td>wcstodN</td>
</tr>
<tr>
<td><code>&lt;stdio.h&gt;</code></td>
<td>printf and scanf families</td>
</tr>
<tr>
<td><code>&lt;wchar.h&gt;</code></td>
<td>wprintf and wscanf families</td>
</tr>
</tbody>
</table>

with:

<table>
<thead>
<tr>
<th>Header</th>
<th>Function groups</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>acosdN, acospidN, asindN, asinpidN, atandN, atan2dN, atan2pidN, atanpidN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>cosdN, cospidN, sindN, sinpidN, tandN, tanpidN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>acoshdN, asinhN, atanhN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>coshdN, sinhN, tanhdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>expdN, exp10dN, exp10mldN, exp2dN, exp2mldN, expmldN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>logdN, log10dN, log10pdN, log1pdN, log2dN, log2pdN, log10pdN, log2pdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>ldexpdN, scalblndN, scalbndN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>cbrtN, compounddN, hypotdN, powdN, powndN, powdN, rootdN, rsqrdN, sqrdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>erfdN, erfcdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>lgammadN, tgammadN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>llrintdN, lrintdN, nearbyintdN, rintdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>quantizedN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>fdimdN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>fmadN</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>dmaddN, dmsubdN, dMfmadN, dMmuldN, dMsqrdN, dMsubdN</td>
</tr>
<tr>
<td>Header</td>
<td>Functions</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td><code>reduc_sumdN, reduc_sumabsdN, reduc_sumproddN, reduc_sumsqdN, scaled_proddN, scaled_proddiffdN, scaled_prodsquadN</code></td>
</tr>
<tr>
<td><code>&lt;stdlib.h&gt;</code></td>
<td><code>strfromdN, strtodN</code></td>
</tr>
<tr>
<td><code>&lt;wchar.h&gt;</code></td>
<td><code>wcstodN</code></td>
</tr>
<tr>
<td><code>&lt;stdio.h&gt;</code></td>
<td><code>printf and scanf families</code></td>
</tr>
<tr>
<td><code>&lt;wchar.h&gt;</code></td>
<td><code>wprintf and wscanf families</code></td>
</tr>
</tbody>
</table>
Bibliography


