N3180

TECHNICAL SPECIFICATION

ISO/IEC TS 18661-4

Second edition
CFP Working Draft
2023-11-13

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

Part 4: Supplementary functions

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

Partie 4: Fonctions supplémentaires

Reference number
ISO/IEC TS 18661-4:20**yy**(E)

COPYRIGHT PROTECTED DOCUMENT

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

Published in Switzerland

Foreword

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

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

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

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

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

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

ISO/IEC TS 18661 originally consisted 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*

— *Part 5: Supplementary attributes*

Parts 1, 2, 3, and some of Part 4 are integrated into ISO/IEC 9899: 2024 (C23).

ISO/IEC TS 18661 Part 4 Version 2, this document, supersedes ISO/IEC TS 18661-4:2015, the previous version of Part 4.

ISO/IEC TS 18661 Part 5 Version 2, a separate document, supersedes ISO/IEC TS 18661-5:2015, the previous version of Part 5.

Introduction

Background

IEC 60559 floating-point standard

The IEC 60559 international standard and the corresponding version of the IEEE 754 standard have equivalent content.

Floating-point standards – matching versions

IEEE 754-1985	IEC 60559:1989
IEEE 754-2008	ISO/IEC/IEEE 60559:2011
IEEE 754-2019	ISO/IEC 60559:2020

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 stated goals of this standard were (and have remained throughout its revisions) the following, quoted from IEEE 754-1985[1], Introduction:

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

— Specified more formats, including decimal as well as binary. It added a 128-bit binary format to its basic formats. It defined extended formats corresponding to all its basic formats. It specified data interchange formats (which may or may not be arithmetic), including a 16-bit binary format and an unbounded sequence 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.

— Specified more operations. It added required operations including (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. It added recommended operations including an extensive set of mathematical operations and seven reduction operations for sums and scaled products.

— Placed more emphasis on reproducible results. This is reflected in its standardization of more operations. For the most part, it completely specified behaviors. It required conversions between floating-point formats and decimal character sequences to be correctly rounded for at least three more decimal digits than is necessary to distinguish all numbers in the widest supported binary format; it completely specified such conversions involving any number of decimal digits. It specified the recommended transcendental functions to be correctly rounded.

— Added 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.

— Added other recommended features including 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 IEEE 754-2019 standard for floating-point arithmetic, which is equivalent to the ISO/IEC 60559:2020 international standard, was a minor revision. As such it was limited to upward-compatible editorial corrections and clarifications and minor enhancements. It added some recommended operations, including ones that might be required features in the next revision.

IEC 60559 (like IEEE 754) defines specific encodings for the exchange of floating-point data between different implementations. However, it does not define the concrete representation (specific layout in storage, or in a processor's register, for example) of data or context.

IEC 60559 (like IEEE 754) does not specify how its features are expressed in programming languages. However, its revisions have added guidance for programming language standards, recognizing that benefits of the floating-point standard, even if well supported in the hardware, are not available to users

unless the programming language provides interfaces for the features and reliable behaviors. 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 a 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 most floating-point operations are also implementation-defined or unspecified, including accuracy and aspects of the way special values and exceptional conditions are handled.

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 the model would have made most of the existing implementations at the time non-conforming.

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

ISO/IEC 9899:2011 (C11) and ISO/IEC 9899:2018 (C17) include refinements to the C99 floating-point specification, though are still based on IEC 60559:1989. C11 upgraded annex G from "informative" to "conditionally normative".

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

ISO/IEC TS 18661 provided a C language binding for ISO/IEC/IEEE 60559:2011, based on the C11 standard. ISO/IEC TS 18661 was organized into five parts:

ISO/IEC TS 18661-1:2014 – Binary floating-point arithmetic

ISO/IEC TS 18661-2:2015 – Decimal floating-point arithmetic, Second edition

ISO/IEC TS 18661-3:2015 – Interchange and extended types

ISO/IEC TS 18661-4:2015 – Supplementary functions

ISO/IEC TS 18661-5:2016 – Supplementary attributes

ISO/IEC 9899:2024 (C23) incorporates ISO/IEC TS 18661 Parts 1 and 2, the mathematical functions in Part 4, and, in an annex, Part 3. C23 also updates its floating-point specification to support ISO/IEC/IEEE 60559:2020.

This document updates ISO/IEC TS 18661-4. It retains the feature that was not incorporated into C23, namely the reduction functions. It also adds support for the augmented arithmetic operations introduced in IEC 60559:2020.

A separate document updates ISO/IEC TS 18661-5, which was not incorporated into C23.

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 and that are not supported in C23.

IEC 60559 specifies and recommends reduction operations which operate on vector operands to perform widely used vector computations. 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.

IEC 60559 also specifies and recommends augmented arithmetic operations, which are versions of operations commonly called twoSum and twoProduct. These operations can be used to implement arithmetic with extra precision, for example for double-double format. They can also be used to implement efficient reproducible dot products.

This document specifies two headers with functions corresponding to the IEC 60559 reduction and augmented arithmetic operations, respectively. This document does not specify type-generic macros for the operations.

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

Part 4: Supplementary functions

1 Scope

This document specifies extensions to programming language C to include functions corresponding to operations specified and recommended in ISO/IEC 60559 and not supported in C23.

2 Conformance

An implementation may conform to one or both feature sets (reduction functions and augmented arithmetic) in this document. It so conforms if

a) it meets the requirements for a conforming implementation of C23;

and to conform to the reduction functions, the following are true:

b) it defines **__STDC_IEC_60559_BFP__** or **__STDC_IEC_60559_DFP__** or both, indicating support for IEC 60559 binary or decimal floating-point arithmetic, as specified in C23 Annex F;

c) it defines **__STDC_IEC_60559_FUNCS_REDUCTION__** to **20yymmL** and provides the **<reduc.h>** header specified in this document (Clause 6);

and to conform to the augmented arithmetic feature, the following are true:

d) it defines **__STDC_IEC_60559_BFP__**, indicating support for IEC 60559 binary floating-point arithmetic, as specified in C23 Annex F;

e) it defines **__STDC_IEC_60559_FUNCS_AUGMENTED_ARITHMETIC__** to **20yymmL** and provides the **<augarith.h>** header specified in this document (Clause 7).

3 Normative references

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

ISO/IEC 9899:2024, *Information technology — Programming languages — C*

ISO/IEC 60559:2020, *Information technology — Microprocessor Systems — Floating-point arithmetic*

4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2024, ISO/IEC 60559:2020, and the following apply.

4.1

C23
standard ISO/IEC 9899:2024, *Information technology — Programming languages C*

5 C standard extensions

5.1 Predefined macros

The implementation defines one or both of the following macros to indicate conformance to the specification in this document for support of the corresponding features specified and recommended in IEC 60559.

__STDC_IEC_60559_FUNCS_REDUCTION__ The integer constant **20yymmL.**

__STDC_IEC_60559_FUNCS_AUGMENTED_ARITHMETIC__ The integer constant **20yymmL.**

5.2 Freestanding implementations

The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines one of the feature macros in 5.1 may also use features in the corresponding header specified in this document. See C23 Clause 4.

5.3 Headers

If the implementation defines one of the feature macros in 5.1 then the implementation provides the corresponding header specified in this document. The header and its use follow the general specification in C23 7.1 for the C Library as though the header were a subclause of C23 Clause 7 for a conditional feature.

5.4 Future directions

For implementations that define **__STDC_IEC_60559_FUNCS_REDUCTION__**, function names that begin with **reduc_** or **scaled_** are potentially reserved identifiers and may be added to the declarations in the **<reduc.h>** header.

For implementations that define **__STDC_IEC_60559_FUNCS_ AUGMENTED_ARITHMETIC__**, tag names that end with **aug_t** and function names that begin with **aug_** are potentially reserved identifiers and may be added to the declarations in the **<augarith.h>** header.

See C23 7.33.

6 Reduction functions <reduc.h>

The header **<reduc.h>** declares the type and functions in this clause.

The type declared is **size_t** (described in C23 7.21).

Each function in this clause is declared in **<reduc.h>** if and only if the corresponding type is supported according to C23 Annex F or Annex H.

The functions in this clause shall be implemented so that intermediate computations do not overflow or underflow. For the **reduc_sum**, **reduc_sumabs**, **reduc_sumsq**, and **reduc_sumprod** functions,

the “overflow” or “underflow” floating-point exception is raised and a range error occurs if and only if 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 and do not cause a range error.

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

With IEC 60559 default exception handling, these functions raise the “inexact” floating-point exception in response to “overflow” and “underflow” exceptions; otherwise, whether they raise the “inexact” floating-point exception is unspecified.

Numerical results and exceptional behavior, including the “invalid” floating-point exception, might differ due to the precision of intermediates and the order of evaluation. However, only one floating-point exception is raised (other than “inexact” in response to “overflow” or “underflow”) per reduction function invocation; exceptions are not raised for each exceptional intermediate operand or result. Reduction functions may raise the “invalid” floating-point exception if an element of an array argument is a signaling NaN (see C23 F.2.1). Once an invalid floating-point exception is raised, due to signaling NaN, ∞−∞, or 0×∞, processing of array elements may stop.

Whether and how rounding direction modes affect functions in this clause are implementation defined and may be indeterminate. This applies to constant as well as dynamic rounding modes, C23 7.6.2 notwithstanding.

The preferred quantum exponent for the reduction functions for decimal floating types is unspecified.

For each of the following synopses, an implementation shall declare the functions suffixed with **f***N* or **f***N***x** only if it supports the corresponding binary floating type and the macro **__STDC_WANT_IEC_60559_TYPES_EXT__** is defined at the point in the code where **<reduc.h>** is first included. An implementation shall declare the functions suffixed with **d***N* for *N* ≠ 32, 64 or 128 or with **d***N***x** only if it supports the corresponding decimal floating type and the macro **__STDC_WANT_IEC_60559_TYPES_EXT__** is defined at the point in the code where **<reduc.h>** is first included.[1] (See C23 Annex H.)

[1] For *N* = 32, 64 and 128, the functions suffixed with **d***N* are declared if the implementation supports decimal floating types (i.e. defines **__STDC_IEC_60559_DFP__**), without the requirement that the macro **__STDC_WANT_IEC_60559_TYPES_EXT__** be defined.

6.1 The `reduc_sum` functions

Synopsis

```
#include <reduc.h>

#ifdef __STDC_IEC_60559_BFP__
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]);
#endif
#ifdef __STDC_IEC_60559_DFP__
_DecimalN reduc_sumdN(size_t n, const _DecimalN p[static n]);
_DecimalNx reduc_sumdNx(size_t n,
   const _DecimalNx p[static n]);
#endif
```

Description

The **reduc_sum** functions compute the sum of the **n** elements of array **p**: $\sum_{i=0}^{\mathbf{n}-1} \mathbf{p}[i]$. If the length **n** = 0, the functions return the value +0. If any element of array **p** is a NaN, the functions return a quiet NaN. If any two elements of array **p** are infinities with different signs, the functions return a quiet NaN and raise the "invalid" floating-point exception and a domain error occurs. Otherwise (if no element of **p** is a NaN and no two elements of **p** are infinities with different signs), if any element of array **p** is an infinity, the functions return that same infinity.

Returns

The **reduc_sum** functions return the computed sum.

6.2 The `reduc_sumabs` functions

Synopsis

```
#include <reduc.h>

#ifdef __STDC_IEC_60559_BFP__
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]);
#endif
#ifdef __STDC_IEC_60559_DFP__
_DecimalN reduc_sumabsdN(size_t n,
   const _DecimalN p[static n]);
_DecimalNx reduc_sumabsdNx(size_t n,
   const _DecimalNx p[static n]);
#endif
```

Description

The **reduc_sumabs** functions compute the sum of the absolute values of the **n** elements of array **p**: $\sum_{i=0}^{\mathbf{n}-1} |\mathbf{p}[i]|$. If the length **n** = 0, the functions return the value +0. If any element of array **p** is an infinity, the functions return +∞; otherwise, if any element of array **p** is a NaN, the functions return a quiet NaN.

Returns

The **reduc_sumabs** functions return the computed sum.

6.3 The reduc_sumsq functions

Synopsis

```
#include <reduc.h>

#ifdef __STDC_IEC_60559_BFP__
double reduc_sumsq(size_t n, const double p[static n]);
float reduc_sumsqf(size_t n, const float p[static n]);
long double reduc_sumsql(size_t n,
   const long double p[static n]);
_FloatN reduc_sumsqfN(size_t n, const _FloatN p[static n]);
_FloatNx reduc_sumsqfNx(size_t n,
   const _FloatNx p[static n]);
#endif
#ifdef __STDC_IEC_60559_DFP__
_DecimalN reduc_sumsqdN(size_t n,
   const _DecimalN p[static n]);
_DecimalNx reduc_sumsqdNx(size_t n,
   const _DecimalNx p[static n]);
#endif
```

Description

The **reduc_sumsq** functions compute the sum of squares of the values of the **n** elements of array **p**: $\sum_{i=0}^{\mathbf{n}-1}(\mathbf{p}[i] \times \mathbf{p}[i])$. If the length **n** = 0, the functions return the value +0. If any element of array **p** is an infinity, the functions return +∞; otherwise, if any element of array **p** is a NaN, the functions return a quiet NaN.

Returns

The **reduc_sumsq** functions return the computed sum.

6.4 The `reduc_sumprod` functions

Synopsis

```
#include <reduc.h>

#ifdef __STDC_IEC_60559_BFP__
double reduc_sumprod(size_t n, const double p[static n],
   const double q[static n]);
float reduc_sumprodf(size_t n, const float p[static n],
   const float q[static n]);
long double reduc_sumprodl(size_t n,
   const long double p[static n],
   const long double q[static n]);
_FloatN reduc_sumprodfN(size_t n, const _FloatN p[static n],
   const _FloatN q[static n]);
_FloatNx reduc_sumprodfNx(size_t n,
   const _FloatNx p[static n],
   const _FloatNx q[static n]);
#endif
#ifdef __STDC_IEC_60559_DFP__
_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]);
#endif
```

Description

The **reduc_sumprod** functions compute the dot product of the sequences of elements of the arrays **p** and **q**: $\sum_{i=0}^{\mathbf{n}-1}(\mathbf{p}[i] \times \mathbf{q}[i])$. If the length **n** = 0, the functions return the value +0. If any element of array **p** or **q** is a NaN, the functions return a quiet NaN. If a product is 0 × ∞, the functions return a quiet NaN and raise the "invalid" floating-point exception and a domain error occurs. If a sum is of infinities of different signs, the functions return a quiet NaN and raise the "invalid" floating-point exception and a domain error occurs. Otherwise (if no array element is a NaN, no product is 0 × ∞, and no sum is of infinities of different signs), if a term in the summation is an infinity, the functions return that same infinity.

Returns

The **reduc_sumprod** functions return the computed dot product.

6.5 The `scaled_prod` functions

Synopsis

```
#include <reduc.h>

#ifdef __STDC_IEC_60559_BFP__
double scaled_prod(size_t n,
   const double p[static restrict n],
   long int * restrict sfptr);
float scaled_prodf(size_t n,
   const float p[static restrict n],
   long int * restrict sfptr);
long double scaled_prodl(size_t n,
   const long double p[static restrict n],
   long int * restrict sfptr);
_FloatN scaled_prodfN(size_t n,
   const _FloatN p[static restrict n],
   long int * restrict sfptr);
_FloatNx scaled_prodfNx(size_t n,
   const _FloatNx p[static restrict n],
   long int * restrict sfptr);
#endif
#ifdef __STDC_IEC_60559_DFP__
_DecimalN scaled_proddN(size_t n,
   const _DecimalN p[static restrict n],
   long int * restrict sfptr);
_DecimalNx scaled_proddNx(size_t n,
   const _DecimalNx p[static restrict n],
   long int * restrict sfptr);
#endif
```

Description

The **scaled_prod** functions compute a scaled product *pr* of the **n** elements of the array **p** and a scale factor *sf*, such that

$$pr \times b^{sf} = \prod_{i=0}^{\mathbf{n}-1} \mathbf{p}[i]$$

where *b* is the radix of the type. These functions store the scale factor *sf* in the object pointed to by **sfptr**. If the length **n** = 0, the functions return the value +1 and store 0 in the object pointed to by **sfptr**. If any element of array **p** is a NaN, the functions return a quiet NaN. If any two elements of array **p** are a zero and an infinity, the functions return a quiet NaN and raise the "invalid" floating-point exception and a domain error occurs. Otherwise, if any element of array **p** is an infinity, the functions return an infinity. Otherwise, if any element of array **p** is a zero, the functions return a zero. Otherwise, if the scale factor is outside the range of the **long int** type, the functions return a quiet NaN and raise the "invalid" floating-point exception. If a zero, infinity, or NaN is returned, the functions store 0 in the object pointed to by **sfptr**.

Returns

The **scaled_prod** functions return the computed scaled product *pr*.

6.6 The `scaled_prodsum` functions

Synopsis

```
#include <reduc.h>

#ifdef __STDC_IEC_60559_BFP__
double scaled_prodsum(size_t n,
   const double p[static restrict n],
   const double q[static restrict n],
   long int * restrict sfptr);
float scaled_prodsumf(size_t n,
   const float p[static restrict n],
   const float q[static restrict n],
   long int * restrict sfptr);
long double scaled_prodsuml(size_t n,
   const long double p[static restrict n],
   const long double q[static restrict n],
   long int * restrict sfptr);
_FloatN scaled_prodsumfN(size_t n,
   const _FloatN p[static restrict n],
   const _FloatN q[static restrict n],
   long int * restrict sfptr);
_FloatNx scaled_prodsumfNx(size_t n,
   const _FloatNx p[static restrict n],
   const _FloatNx q[static restrict n],
   long int * restrict sfptr);
#endif
#ifdef __STDC_IEC_60559_DFP__
_DecimalN scaled_prodsumdN(size_t n,
   const _DecimalN p[static restrict n],
   const _DecimalN q[static restrict n],
   long int * restrict sfptr);
_DecimalNx scaled_prodsumdNx(size_t n,
   const _DecimalNx p[static restrict n],
   const _DecimalNx q[static restrict n],
   long int * restrict sfptr);
#endif
```

Description

The **scaled_prodsum** functions compute a scaled product *pr* of the sums of the corresponding elements of the arrays **p** and **q** and a scale factor *sf*, such that

$$pr \times b^{sf} = \prod_{i=0}^{\mathbf{n}-1}(\mathbf{p}[i] + \mathbf{q}[i])$$

where *b* is the radix of the type. These functions store the scale factor *sf* in the object pointed to by **sfptr**. If the length **n** = 0, the functions return the value +1 and store 0 in the object pointed to by **sfptr**. If any element of array **p** or **q** is a NaN, the functions return a quiet NaN. If any sum is of infinities with different signs or if any two factors in the product are a zero and an infinity, the functions return a quiet NaN and raise the "invalid" floating-point exception and a domain error occurs. Otherwise, if any factor in the product is an infinity, the functions return an infinity. Otherwise, if any factor in the product is a zero, the functions return a zero. Otherwise, if the scale factor is outside the

range of the **long int** type, the functions return a quiet NaN and raise the "invalid" floating-point exception. If a zero, infinity, or NaN is returned, the functions store 0 in the object pointed to by **sfptr**.

Returns

The **scaled_prodsum** functions return the computed scaled product *pr*.

6.7 The scaled_proddiff functions

Synopsis

```
#include <reduc.h>

#ifdef __STDC_IEC_60559_FUNCS_REDUCTION__
#ifdef __STDC_IEC_60559_BFP__
double scaled_proddiff(size_t n,
   const double p[static restrict n],
   const double q[static restrict n],
   long int * restrict sfptr);
float scaled_proddifff(size_t n,
   const float p[static restrict n],
   const float q[static restrict n],
   long int * restrict sfptr);
long double scaled_proddiffl(size_t n,
   const long double p[static restrict n],
   const long double q[static restrict n],
   long int * restrict sfptr);
_FloatN scaled_proddifffN(size_t n,
   const _FloatN p[static restrict n],
   const _FloatN q[static restrict n],
   long int * restrict sfptr);
_FloatNx scaled_proddifffNx(size_t n,
   const _FloatNx p[static restrict n],
   const _FloatNx q[static restrict n],
   long int * restrict sfptr);
#endif
#ifdef __STDC_IEC_60559_DFP__
_DecimalN scaled_proddiffdN(size_t n,
   const _DecimalN p[static restrict n],
   const _DecimalN q[static restrict n],
   long int * restrict sfptr);
_DecimalNx scaled_proddiffdNx(size_t n,
   const _DecimalNx p[static restrict n],
   const _DecimalNx q[static restrict n],
   long int * restrict sfptr);
#endif
```

Description

The **scaled_proddiff** functions compute a scaled product *pr* of the differences of the corresponding elements of the arrays **p** and **q** and a scale factor *sf*, such that

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

where *b* is the radix of the type. These functions store the scale factor *sf* in the object pointed to by **sfptr**. If the length **n** = 0, the functions return the value +1 and store 0 in the object pointed to by **sfptr**. If any element of array **p** or **q** is a NaN, the functions return a quiet NaN. If any difference is of infinities with the same signs or if any two factors in the product are a zero and an infinity, the functions return a quiet NaN and raise the "invalid" floating-point exception and a domain error occurs. Otherwise, if any factor in the product is an infinity, the functions return an infinity. Otherwise, if any factor in the product is a zero, the functions return a zero. Otherwise, if the scale factor is outside the range of the **long int** type, the functions return a quiet NaN and raise the "invalid" floating-point exception. If a zero, infinity, or NaN is returned, the functions store 0 in the object pointed to by **sfptr**.

Returns

The **scaled_proddiff** functions return the computed scaled product *pr*.

EXAMPLE The scaled reduction functions support computing quantities of modest magnitudes whose intermediate results might well overflow and underflow. One example is the computation of Clebsch-Gordan coefficients or Wigner 3-j symbols for quantum physics. Expressions for these quantities involve quotients of products of factorials, and so are prone to intermediate overflow. As a simplified example, consider computing a fragment of the Clebsch-Gordan calculation.

```
#include <reduc.h>
#include <math.h>

// compute quot = n1! * n2! / n3!

int n1 = 140, n2 = 160, n3 = 200; // factorial magnitudes
                                  //  1e241, 1e284, 1e374
                                  // quot magnitude 1e151

// products scaled to avoid overflow
double num1, num2, den;
// scale factors
long int num1e, num2e, dene;

// products scaled again to avoid intermediate overflow
// in final computation
double num1s, num2s, dens;
// scale factors
long int num1es, num2es, denes;

// result
double quot;

// arrays { 2, 3, 4, ... }
double num1p[n1-1], num2p[n2-1], denp[n3-1];

// n1! scaled to avoid overflow
for (int i = 2; i <= n1; i++) {
        num1p[i-2] = i;
}
num1 = scaled_prod(n1-1, num1p, &num1e);

// n2! scaled to avoid overflow
for (int i = 2; i <= n2; i++) {
        num2p[i-2] = i;
```

```
        }
        num2 = scaled_prod(n2-1, num2p, &num2e);

        // n3! scaled to avoid overflow
        for (int i = 2; i <= n3; i++) {
                denp[i-2] = i;
        }
        den = scaled_prod(n3-1, denp, &dene);

        // re-scale to avoid subsequent intermediate overflow
        num1es = llogb(num1);
        num1s = scalbln(num1, -num1es);
        num2es = llogb(num2);
        num2s = scalbln(num2, -num2es);
        denes = llogb(den);
        dens = scalbln(den, -denes);

        // compute result without intermediate overflow
        quot = scalbln(num1s * num2s / dens,
          num1e + num2e − dene + num1es + num2es - denes);
```

7 Augmented arithmetic functions <augarith.h>

This clause supports augmented arithmetic, as recommended by IEC 60559 for its binary formats. Each type and each function in this clause is declared in **<augarith.h>** if and only if the corresponding type is an IEC 60559 floating type supported according to C23 Annex F or an interchange or extended type supported according to C23 Annex H.

The types are structures for returning two floating-point values:

```
struct daug_t { double h; double t; };
struct faug_t { float h; float t; };
struct ldaug_t { long double h; long double t; };
struct _fNaug_t { _FloatN h; _FloatN t; };
struct _fNxaug_t { _FloatNx h; _FloatNx t; };
```

The functions in this clause use these structures to return a “head” value *h* and “tail” value *t* to represent an extra-precise result value given by *h* + *t* . See **EXAMPLE** in this clause.

The functions in this clause round to nearest with ties toward zero, a rounding direction specified by IEC 60559 for use by augmented arithmetic operations.[2] Thus, results are independent of dynamic and constant rounding direction modes. Like other IEC 60559 operations, rounding is done with gradual underflow.

For each of the following synopses, an implementation shall declare the functions suffixed with **f***N* or **f***N***x** only if it supports the corresponding binary floating type and the macro **__STDC_WANT_IEC_60559_TYPES_EXT__** is defined at the point in the code where **<augarith.h>** is first included.

[2] [16] shows how to use currently available IEC 60559 operations and to-nearest, ties-to-even rounding to implement the IEC 60559 augmented arithmetic operations with their special to-nearest, ties-toward-zero rounding.

7.1 The `aug_add` functions

Synopsis

```
#include <augarith.h>

struct daug_t aug_add(double x, double y);
struct faug_t aug_addf(float x, float y);
struct ldaug_t aug_addl(long double x, long double y);
struct _fNaug_t aug_addfN(_FloatN x, _FloatN y);
struct _fNxaug_t aug_addfNx(_FloatNx x, _FloatNx y);
```

Description

The **aug_add** functions compute two result values:

h: the sum **x** + **y** rounded to the type using round-to-nearest with ties toward zero;

t: the error in **h** as a computation of **x** + **y**.

If **h** is a non-zero finite number, **t** has the value **x** + **y** − **h** (which is exactly representable in the type), where if **t** is zero its sign is the sign of **h**. If **h** is zero, **t** has the value of **h** (and both have the same sign). If **h** is infinite, **t** has the value of **h**. If **h** is a NaN, **t** is the same NaN.

These functions raise floating-point exceptions like the computation of **h**, except that under IEC 60559 default exception handling they raise the "inexact" floating-point exception only when the computation of **h** overflows.

A range error occurs when the computation of **h** overflows. A domain error occurs when the arguments are infinities with different signs.

Returns

These functions return the sum and error in a structure.

7.2 The `aug_sub` functions

Synopsis

```
#include <augarith.h>

struct daug_t aug_sub(double x, double y);
struct faug_t aug_subf(float x, float y);
struct ldaug_t aug_subl(long double x, long double y);
struct _fNaug_t aug_subfN(_FloatN x, _FloatN y);
struct _fNxaug_t aug_subfNx(_FloatNx x, _FloatNx y);
```

Description

The **aug_sub** functions compute two result values:

h: the difference **x** − **y** rounded to the type using round-to-nearest with ties toward zero;

t: the error in **h** as a computation of **x** − **y**.

If **h** is a non-zero finite number, **t** has the value **x** − **y** − **h** (which is exactly representable in the type), where if **t** is zero its sign is the sign of **h**. If **h** is zero, **t** has the value of **h** (and both have the same sign). If **h** is infinite, **t** has the value of **h**. If **h** is a NaN, **t** is the same NaN.

These functions raise floating-point exceptions like the computation of **h**, except that under IEC 60559 default exception handling they raise the "inexact" floating-point exception only when the computation of **h** overflows.

A range error occurs when the computation of **h** overflows. A domain error occurs when the arguments are infinities with the same sign.

Returns

These functions return the difference and error in a structure.

7.3 The `aug_mul` functions

Synopsis

```
#include <augarith.h>

struct daug_t aug_mul(double x, double y);
struct faug_t aug_mulf(float x, float y);
struct ldaug_t aug_mull(long double x, long double y);
struct _fNaug_t aug_mulfN(_FloatN x, _FloatN y);
struct _fNxaug_t aug_mulfNx(_FloatNx x, _FloatNx y);
```

Description

The **aug_mul** functions compute two result values:

h: the product **x** × **y** rounded to the type using round-to-nearest with ties toward zero;

t: the error in **h** as a computation of **x** × **y**.

If **h** is a nonzero finite number and **x** × **y** − **h** is exactly representable in the type, **t** has the value **x** × **y** − **h**, where if **t** is zero its sign is the sign of **h**. If **h** is a nonzero finite number and **x** × **y** − **h** is not exactly representable in the type (because the magnitude of its value is too small), **t** has the value **x** × **y** − **h** rounded to the type using round-to-nearest with ties toward zero. If **h** is zero, **t** has the value of **h** (and both have the same sign). If **h** is infinite, **t** has the value of **h**. If **h** is a NaN, **t** is the same NaN.

These functions raise the "overflow" and "invalid" floating-point exceptions like the computation of **h.** They raise the "underflow" floating-point exception like the computation of **t**. They raise the "inexact" floating-point exception when and only when the computation of **h** overflows or the computation of **t** is inexact.

A range error occurs when the computation of **h** overflows and may occur when the computation of **t** underflows. A domain error occurs when one argument is zero and the other an infinity.

Returns

These functions return the product and error in a structure.

EXAMPLE The augmented arithmetic operations are useful for extending precision, particularly for implementing "double-double" arithmetic, which provides a faster, though less precise and less predictable, alternative to IEC 60559 binary128 on systems which lack hardware support for binary128.

Double-double represents numbers as a pair of doubles, the second no larger in magnitude than the first, and usually much smaller. Ideally, in the pair (*h*, *t*), *h* would equal the correctly rounded result of computed (*h*+*t*), and *t* would equal the correctly rounded result of *h* + *t* – correctly-rounded-computed (*h*+*t*). Performance considerations often compromise this ideal. There is no standard specification for double-double.

Let the pair (*ah*, *at*) represent a double-double value *a* = *ah* + *at* in infinite precision, and (*bh*, *bt*) a similar double-double value *b*. Consider the double-double sum *s* represented by (*sh*, *st*) where *s* = *sh* + *st* closely approximates *a* + *b* = *ah* + *at* + *bh* + *bt* . The code below uses augmented addition to compute such a sum.

```
#include <augarith.h>

// components of double-double values a = 1/3, b = 2/3, and s
double ah = 0x0.AAAAAAAAAAAAA8p-1, at = 0x0.AAAAAAAAAAAAA8p-55;
double bh = 0x0.AAAAAAAAAAAAA8p0, bt = 0x0.AAAAAAAAAAAAA8p-54;
double sh, st;

struct daug_t u, v, w, y, z;

// compute components of s = a + b
                        // exact sum is ah + at + bh + bt
u = aug_add(ah, bh);   // exact sum is u.h + u.t + at + bt
v = aug_add(at, bt);   // exact sum is u.h + u.t + v.h + v.t
w = aug_add(u.t, v.t); // exact sum is u.h + v.h + w.h + w.t
y = aug_add(v.h, w.h); // exact sum is u.h + y.h + y.t + w.t
z = aug_add(u.h, y.h); // exact sum is z.h + z.t + y.t + w.t

sh = z.h;
st = z.t;
```

The code gives a good approximation to the ideal result, with absolute error **y.t** + **w.t**, and it is commutative and without conditional branches.

The steps for **w** and **y** could use regular addition (**+**) rather than **aug_add**, because **w.t** and **y.t** are not used in the calculation. The code above gives a name to **w.t** and **y.t** for the didactic purpose of the error formula. It also assures a consistent result regardless of the evaluation method.

The special cases for the code might be no worse than those of any other double-double implementation fast enough to be useful. Its performance might be acceptable if **aug_add** is no slower than two adds. There are faster ways to do double-double addition if hardware support for **aug_add** is not available.

Bibliography

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

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

[3] IEEE 754-2019, *IEEE Standard for Floating-Point Arithmetic*

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

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

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

[7] 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*

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

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

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

[11] ISO/IEC TS 18661-4:2015, *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —* Supplementary functions

[12] ISO/IEC TS 18661-5:2016, *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —* Supplementary attributes

[13] ISO/IEC 9899:1999, *Information technology — Programming languages — C*, Second edition

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

[15] ISO/IEC 9899:2018, *Information technology — Programming languages — C*

[16] Boldo, S., Lauter, C., Muller, J.-M., Emulating round-to-nearest-ties-to-zero “augmented” floating-point operations using round-to-nearest-ties-to-even arithmetic. *IEEE Transactions on Computers*, 2021, 70 (7), pp.1046 - 1058. Available at: https://hal.science/hal-02137968v4.