IEEE STANDARDS ASSOCIATION

# IEEE Standard for Floating-Point Arithmetic

IEEE Computer Society

Developed by the
Microprocessor Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 754™-2019

**IEEE Std 754™-2019**
(Revision of IEEE Std 754-2008)

# IEEE Standard for Floating-Point Arithmetic

Sponsor

**Microprocessor Standards Committee**
of the
**IEEE Computer Society**

Approved 13 June 2019

**IEEE-SA Standards Board**

**Abstract:** This standard specifies interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments. This standard specifies exception conditions and their default handling. An implementation of a floating-point system conforming to this standard may be realized entirely in software, entirely in hardware, or in any combination of software and hardware. For operations specified in the normative part of this standard, numerical results and exceptions are uniquely determined by the values of the input data, sequence of operations, and destination formats, all under user control.

**Keywords:** arithmetic, binary, computer, decimal, exponent, floating-point, format, IEEE 754™, interchange, NaN, number, rounding, significand, subnormal.

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA



PDF: ISBN 978-1-5044-5924-2 STD23738
Print: ISBN 978-1-5044-5925-9 STDPD23738

## Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading "Important Notices and Disclaimers Concerning IEEE Standards Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/ipr/disclaimers.html.

## Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association ("IEEE-SA") Standards Board. IEEE ("the Institute") develops its standards through a consensus development process, approved by the American National Standards Institute ("ANSI"), which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed through scientific, academic, and industry-based technical working groups. Volunteers in IEEE working groups are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied "AS IS" and "WITH ALL FAULTS."

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

## Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

## Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

## Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

### Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

## Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

## Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

## Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. A current IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit IEEE Xplore at http://ieeexplore.ieee.org/ or contact IEEE at the address listed previously. For more information about the IEEE-SA or IEEE's standards development process, visit the IEEE-SA Website at http://standards.ieee.org.

## Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

## Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

## Participants

The following participants in the Floating-Point Working Group contributed to the development of this standard:

**David G. Hough**, *Chair*
**Mike Cowlishaw**, *Editor*

Jonathan Bradbury
Neil Burgess
David H. C. Chen
Marius Cornea
John H. Crawford
Joe Darcy
James Demmel
Florent de Dinechin
Ken Dockser
Hossam A. H. Fahmy
Warren E. Ferguson
David M. Gay
Ivan Godard
Roger A. Golliver
Mrudula Gore
Trenton Grale
Michel Hack
John Hauser
Peter C. B. Henderson
William Kahan
R. Baker Kearfott
Christoph Lauter
Vincent Lefèvre
David Lutz
Terje Mathisen
David Matula
Ian McIntosh
Richard A. Painter
Bogdan Pasca
Nathalie Revol
Jason Riedy
Eric M. Schwarz
James W. Thomas
Leonard Tsai
Fred J. Tydeman
Liang-Kai Wang
Lee Winter

The following individual members of the balloting committee voted on this standard. Balloters might have voted for approval, disapproval, or abstention.

Robert Aiello
Amelia Andersdotter
Israel Barrientos
Demetrio Bucaneg Jr.
David H. C. Chen
James Cloos
Marius Cornea
Mike Cowlishaw
James Demmel
Ken Dockser
Hossam A. H. Fahmy
Andrew Fieldsend
David M. Gay
H. Glickenstein
Roger A. Golliver
Randall Groves
Michel Hack
Peter Harrod
Chris N. Hinds
Werner Hoelzl
David G. Hough
Piotr Karocki
R. Baker Kearfott
Jim Kulchisky
Christoph Lauter
Vincent Lefèvre
Edward Mccall
Jean-Michel Muller
Bruce Muschlitz
Ned Nedialkov
Nick S. A. Nikjoo
Richard A. Painter
John Pryce
Nathalie Revol
Jason Riedy
Randy Saunders
Eric M. Schwarz
James Stine
Walter Struppler
James W. Thomas
Michael Thompson
Leonard Tsai
Forrest Wright
Jian Yu
Oren Yuen

When the IEEE-SA Standards Board approved this standard on 13 June 2019, it had the following membership:

**Gary Hoffman**, *Chair*
**Ted Burse**, *Vice Chair*
**Jean-Philippe Faure**, *Past Chair*
**Konstantinos Karachalios**, *Secretary*

Masayuki Ariyoshi
Stephen D. Dukes
J. Travis Griffith
Guido Hiertz
Christel Hunter
Joseph L. Koepfinger*
Thomas Koshy
John D. Kulick
David J. Law
Joseph Levy
Howard Li
Xiaohui Liu
Kevin Lu
Daleep Mohla
Andrew Myles
Annette Reilly
Dorothy Stanley
Sha Wei
Phil Wennblom
Philip Winston
Howard Wolfman
Feng Wu
Jingyi Zhou

* Member Emeritus

## Introduction

This introduction is not part of IEEE Std 754-2019, IEEE Standard for Floating-Point Arithmetic.

This standard is a product of the Floating-Point Working Group of, and sponsored by, the Microprocessor Standards Committee of the IEEE Computer Society.

This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two. For operations specified in the normative part of this standard, numerical results and exceptions are uniquely determined by the values of the input data, the operation, and the destination, all under user control.

This standard defines a family of commercially feasible ways for systems to perform binary and decimal floating-point arithmetic. Among the desiderata that guided the formulation of this standard were:

a) Facilitate movement of existing programs from diverse computers to those that adhere to this standard as well as among those that adhere to this standard.

b) Enhance the capabilities and safety available to users and programmers who, although not expert in numerical methods, might well be attempting to produce numerically sophisticated programs.

c) 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. Together with language controls it should be possible to write programs that produce identical results on all conforming systems.

d) Provide direct support for
   - execution-time diagnosis of anomalies
   - smoother handling of exceptions
   - interval arithmetic at a reasonable cost.

e) Provide for development of
   - common elementary functions such as *exp* or *cos*
   - high precision (multiword) arithmetic
   - coupled numerical and symbolic algebraic computation.

f) Enable rather than preclude further refinements and extensions.

In programming environments, this standard is also intended to form the basis for a dialog between the numerical community and programming language designers. It is hoped that language-defined methods for the control of expression evaluation and exceptions might be defined in coming years, so that it will be possible to write programs that produce identical results on all conforming systems. However, it is recognized that utility and safety in languages are sometimes antagonists, as are efficiency and portability.

Therefore, it is hoped that language designers will look on the full set of operation, precision, and exception controls described here as a guide to providing the programmer with the ability to portably control expressions and exceptions. It is also hoped that designers will be guided by this standard to provide extensions in a completely portable way.

Informative annexes provide additional information – Annex A lists bibliographical resources, Annex B suggests programming environment features for debugging support, and Annex C lists all references to the operations of the standard.

## Contents

# IEEE Standard for Floating-Point Arithmetic

## 1. Overview

### 1.1 Scope

This standard specifies formats and operations for floating-point arithmetic in computer systems. Exception conditions are defined and handling of these conditions is specified.

### 1.2 Purpose

This standard provides a method for computation with floating-point numbers that will yield the same result whether the processing is done in hardware, software, or a combination of the two. The results of the computation will be identical, independent of implementation, given the same input data. Errors, and error conditions, in the mathematical processing will be reported in a consistent manner regardless of implementation.

### 1.3 Inclusions

This standard specifies:

- Formats for binary and decimal floating-point data, for computation and data interchange.
- Addition, subtraction, multiplication, division, fused multiply add, square root, compare, and other operations.
- Conversions between integer and floating-point formats.
- Conversions between different floating-point formats.
- Conversions between floating-point formats and external representations as character sequences.
- Floating-point exceptions and their handling, including data that are not numbers (NaNs).

### 1.4 Exclusions

This standard does not specify:

- Formats of integers.
- Interpretation of the sign and significand fields of NaNs.

## 1.5 Programming environment considerations

This standard specifies floating-point arithmetic in two radices, 2 and 10. A programming environment may conform to this standard in one radix or in both.

This standard does not define all aspects of a conforming programming environment. Such behavior should be defined by a programming language definition supporting this standard, if available, and otherwise by a particular implementation. Some programming language specifications might permit some behaviors to be defined by the implementation.

**Language-defined** behavior should be defined by a programming language standard supporting this standard. Then all implementations conforming both to this floating-point standard and to that language standard behave identically with respect to such language-defined behaviors. Standards for languages intended to reproduce results exactly on all platforms are expected to specify behavior more tightly than do standards for languages intended to maximize performance on every platform.

Because this standard requires facilities that are not currently available in common programming languages, the standards for such languages might not be able to fully conform to this standard if they are no longer being revised. If the language can be extended by a function library or class or package to provide a conforming environment, then that extension should define all the language-defined behaviors that would normally be defined by a language standard.

**Implementation-defined** behavior is defined by a specific implementation of a specific programming environment conforming to this standard. Implementations define behaviors not specified by this standard nor by any relevant programming language standard or programming language extension.

Conformance to this standard is a property of a specific implementation of a specific programming environment, rather than of a language specification.

However a language standard could also be said to conform to this standard if it were constructed so that every conforming implementation of that language also conformed automatically to this standard.

## 1.6 Word usage

In this standard three words are used to differentiate between different levels of requirements and optionality, as follows:

- **may** indicates a course of action permissible within the limits of the standard with no implied preference (“may” means “is permitted to”)
- **shall** indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (“shall” means “is required to”)
- **should** indicates that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (“should” means “is recommended to”).

Further:

- **might** indicates the possibility of a situation that could occur, with no implication of the likelihood of that situation (“might” means “could possibly”)
- **see** followed by a number is a cross-reference to the clause or subclause of this standard identified by that number
- **NOTE** introduces text that is informative (that is, is not a requirement of this standard).

# 2. Definitions, abbreviations, and acronyms

## 2.1 Definitions

For the purposes of this standard, the following terms and definitions apply.

**applicable attribute:** The value of an attribute governing a particular instance of execution of a computational operation of this standard. Languages specify how the applicable attribute is determined.

**arithmetic format:** A floating-point format that can be used to represent floating-point operands or results for the operations of this standard.

**attribute:** An implicit parameter to operations of this standard, which a user might statically set in a programming language by specifying a constant value. The term attribute might refer to the parameter (as in "rounding-direction attribute") or its value (as in "roundTowardZero attribute").

**basic format:** One of five floating-point representations, three binary and two decimal, whose encodings are specified by this standard, and which can be used for arithmetic. One or more of the basic formats is implemented in any conforming implementation.

**biased exponent:** The sum of the exponent and a constant (bias) chosen to make the biased exponent's range non-negative.

**binary floating-point number:** A floating-point number with radix two.

**block:** A language-defined syntactic unit for which a user can specify attributes. Language standards might provide means for users to specify attributes for blocks of varying scopes, even as large as an entire program and as small as a single operation.

**canonical encoding:** A preferred encoding of a floating-point representation in a format. "Canonical encoding" also applies to declets, significands of finite numbers, infinities, and NaNs, especially in decimal formats.

**cohort:** The set of all floating-point representations that represent a given floating-point number in a given floating-point format. In this context −0 and +0 are considered distinct and are in different cohorts.

**computational operation:** An operation that produces floating-point results or that might signal floating-point exceptions. Computational operations produce results in floating-point or other destination formats by rounding them to fit if necessary.

**correct rounding:** This standard's method of converting an infinitely precise result to a floating-point number, as determined by the applicable rounding direction. A floating-point number so obtained is said to be correctly rounded.

**decimal floating-point number:** A floating-point number with radix ten.

**declet:** An encoding of three decimal digits into ten bits using the densely packed decimal encoding scheme. Computational operations accept all 1024 possible declets in operands. Most computational operations produce only the 1000 canonical declets.

**denormalized number:** *See:* **subnormal number.**

**destination:** The location for the result of an operation upon one or more operands. A destination might be either explicitly designated by the user or implicitly supplied by the system (for example, intermediate results in subexpressions or arguments for procedures). Even though some languages place the results of intermediate calculations in destinations beyond the user's control, this standard defines the result of an operation in terms of that destination's format and the operands' values.

**dynamic mode:** An optional method of dynamically setting attributes by means of operations of this standard to set, test, save, and restore them.

**exception:** An event that occurs when an operation on some particular operands has no outcome suitable for every reasonable application. That operation might signal an exception by invoking default exception handling or alternate exception handling. Exception handling might signal further exceptions. Recognize that *event*, *exception*, and *signal* are defined in diverse ways in different programming environments.

**exponent:** The component of a finite floating-point representation that signifies the integer power to which the radix is raised in determining the value of that floating-point representation. The exponent $e$ is used when the significand is regarded as an integer digit and fraction field, and the exponent $q$ is used when the significand is regarded as an integer; $e=q+p-1$ where $p$ is the precision of the format in digits.

**extendable precision format:** A format with precision and range that are defined under user control.

**extended precision format:** A format that extends a supported basic format by providing wider precision and range.

**external character sequence:** A representation of a floating-point datum as a sequence of characters, including the character sequences in floating-point literals in program text.

**flag:** *See:* **status flag.**

**floating-point datum:** A floating-point number or non-number (NaN) that is representable in a floating-point format. In this standard, a floating-point datum is not always distinguished from its representation or encoding.

**floating-point number:** A finite or infinite number that is representable in a floating-point format. A floating-point datum that is not a NaN. All floating-point numbers, including zeros and infinities, are signed.

**floating-point operation:** An operation where an operand or result is a floating-point datum.

**floating-point representation:** An unencoded member of a floating-point format, representing a finite number, a signed infinity, a quiet NaN, or a signaling NaN. A representation of a finite number has three components: a sign, an exponent, and a significand; its numerical value is the signed product of its significand and its radix raised to the power of its exponent.

**format:** A set of representations of numerical values and symbols, perhaps accompanied by an encoding.

**fusedMultiplyAdd:** The operation fusedMultiplyAdd($x$, $y$, $z$) computes $(x\times y)+z$ as if with unbounded range and precision, rounding only once to the destination format.

**generic operation:** An operation of this standard that can take operands of various formats, for which the formats of the results might depend on the formats of the operands.

**homogeneous operation:** An operation of this standard that takes operands and returns results all in the same format.

**implementation-defined:** Behavior defined by a specific implementation of a specific programming environment conforming to this standard.

**integer format:** A format not defined in this standard that represents a subset of the integers and perhaps additional values representing infinities, NaNs, or negative zeros.

**interchange format:** A format that has a specific fixed-width encoding defined in this standard.

**language-defined:** Behavior defined by a programming language standard supporting this standard.

**NaN:** not a number—a symbolic floating-point datum. There are two kinds of NaN representations: quiet and signaling. Most operations propagate **quiet NaN**s without signaling exceptions, and signal the invalid operation exception when given a **signaling NaN** operand.

**narrower/wider format:** If the set of floating-point numbers of one format is a proper subset of another format, the first is called narrower and the second wider. The wider format might have greater precision, range, or (usually) both.

**non-computational operation**: An operation that is not computational.

**normal number:** For a particular format, a finite non-zero floating-point number with magnitude greater than or equal to a minimum $b^{emin}$ value, where $b$ is the radix. Normal numbers can use the full precision available in a format. In this standard, zero is neither normal nor subnormal.

**not a number:** *See:* **NaN.**

**operation:** this standard defines required and recommended operations which operate on zero or more operands and produce results or side effects, such as changes in dynamic modes or flags or control flow, or both. In this standard, operations are written as named functions; in a specific programming environment

they might be represented by operators, or by families of format-specific functions, or by operations or functions whose names might differ from those in this standard.

**payload:** The information, which might be diagnostic, contained in a NaN.

**precision:** The maximum number $p$ of significant digits that can be represented in a format, or the number of digits to which a result is rounded.

**preferred exponent:** For the result of a decimal operation, the value of the exponent $q$ which best reflects the quanta of the operands when the result is exact.

**preferredWidth method:** A method used by a programming language to determine the destination formats for generic operations and functions. Some **preferredWidth** methods take advantage of the extra range and precision of wide formats without requiring the program to be written with explicit conversions.

**quantum:** The quantum of a finite floating-point representation is the value of a unit in the last position of its significand. This is equal to the radix raised to the exponent $q$, which is used when the significand is regarded as an integer.

**quiet operation:** An operation that never signals any floating-point exception.

**radix:** The base for the representation of binary or decimal floating-point numbers, two or ten.

**result:** The floating-point representation or encoding that is delivered to the destination.

**signal:** When an operation on some particular operands has no outcome suitable for every reasonable application, that operation might signal one or more exceptions by invoking the default handling or, if explicitly requested, a language-defined alternate handling selected by the user.

**significand:** A component of a finite floating-point number containing its significant digits. The significand can be thought of as an integer, a fraction, or some other fixed-point form, by choosing an appropriate exponent offset. A decimal or subnormal binary significand can also contain leading zeros, which are not significant.

**status flag:** A variable that can take two states, raised or lowered. When raised, a status flag might convey additional system-dependent information, possibly inaccessible to some users. The operations of this standard, when exceptional, can as a side effect raise some of the following status flags: inexact, underflow, overflow, divideByZero, and invalid operation.

**subnormal number:** In a particular format, a non-zero floating-point number with magnitude less than the magnitude of that format's smallest normal number. A subnormal number does not use the full precision available to normal numbers of the same format.

**supported format:** A floating-point format provided in the programming environment and implemented in conformance with the requirements of this standard. Thus, a programming environment might provide more formats than it supports, as only those implemented in accordance with the standard are said to be supported. Also, an integer format is said to be supported if conversions between that format and supported floating-point formats are provided in conformance with this standard.

**trailing significand field:** A component of an encoded binary or decimal floating-point format containing all the significand digits except the leading digit. In these formats, the biased exponent or combination field encodes or implies the leading significand digit.

**user:** Any person, hardware, or program not itself specified by this standard, having access to and controlling those operations of the programming environment specified in this standard.

**width of an operation:** The format of the destination of an operation specified by this standard; it will be one of the supported formats provided by an implementation in conformance to this standard.

## 2.2 Abbreviations and acronyms

| | |
|---|---|
| LSB | least significant bit |
| MSB | most significant bit |
| NaN | not a number |
| qNaN | quiet NaN |
| sNaN | signaling NaN |

# 3. Floating-point formats

## 3.1 Overview

### 3.1.1 Formats

This clause defines floating-point formats, which are used to represent a finite subset of real numbers (see 3.2). Formats are characterized by their radix, precision, and exponent range, and each format can represent a unique set of floating-point data (see 3.3).

All formats can be supported as **arithmetic formats**; that is, they may be used to represent floating-point operands or results for the operations described in later clauses of this standard.

Specific fixed-width encodings for binary and decimal formats are defined in this clause for a subset of the formats (see 3.4 and 3.5). These **interchange formats** are identified by their size (see 3.6) and can be used for the exchange of floating-point data between implementations.

Five **basic formats** are defined in this clause:

— Three binary formats, with encodings in lengths of 32, 64, and 128 bits.
— Two decimal formats, with encodings in lengths of 64 and 128 bits.

Additional arithmetic formats are recommended for extending these basic formats (see 3.7).

The choice of which of this standard's formats to support is language-defined or, if the relevant language standard is silent or defers to the implementation, implementation-defined. The names used for formats in this standard are not necessarily those used in programming environments.

### 3.1.2 Conformance

A conforming implementation of any supported format shall provide means to initialize that format and shall provide conversions between that format and all other supported formats.

A conforming implementation of a supported arithmetic format shall provide all the operations of this standard defined in Clause 5, for that format.

A conforming implementation of a supported interchange format shall provide means to read and write that format using a specific encoding defined in this clause, for that format.

A programming environment conforms to this standard, in a particular radix, by implementing one or more of the basic formats of that radix as both a supported arithmetic format and a supported interchange format.

## 3.2 Specification levels

Floating-point arithmetic is a systematic approximation of real arithmetic, as illustrated in Table 3.1. Floating-point arithmetic can only represent a finite subset of the continuum of real numbers. Consequently certain properties of real arithmetic, such as associativity of addition, do not always hold for floating-point arithmetic.

**Table 3.1—Relationships between different specification levels for a particular format**

| | | |
|---|---|---|
| Level 1 | **{−∞ … 0 … +∞}** | Extended real numbers. |
| many-to-one ↓ | *rounding* | ↑ projection (except for NaN) |
| Level 2 | **{−∞ … −0} ∪ {+0 … +∞} ∪ NaN** | Floating-point data—an algebraically closed system. |
| one-to-many ↓ | *representation specification* | ↑ many-to-one |
| Level 3 | **(*sign*, *exponent*, *significand*) ∪ {−∞, +∞} ∪ qNaN ∪ sNaN** | Representations of floating-point data. |
| one-to-many ↓ | *encoding for representations of floating-point data* | ↑ many-to-one |
| Level 4 | **0111000…** | Bit strings. |

The mathematical structure underpinning the arithmetic in this standard is the extended reals, that is, the set of real numbers together with positive and negative infinity. For a given format, the process of *rounding* (see Clause 4) maps an extended real number to a *floating-point number* included in that format. A *floating-point datum*, which can be a signed zero, finite non-zero number, signed infinity, or a NaN (not-a-number), can be mapped to one or more *representations of floating-point data* in a format.

The representations of floating-point data in a format consist of:

— triples (*sign*, *exponent*, *significand*); in radix *b*, the floating-point number represented by a triple is $(-1)^{sign} \times b^{exponent} \times significand$
— +∞, −∞
— qNaN (quiet), sNaN (signaling).

An *encoding* maps a representation of a floating-point datum to a bit string. An encoding might map some representations of floating-point data to more than one bit string. A NaN encoding should be used to store retrospective diagnostic information (see 6.2).

## 3.3 Sets of floating-point data

This subclause specifies the sets of floating-point data representable within all floating-point formats; the encodings for specific representations of floating-point data in interchange formats are defined in 3.4 and 3.5, and the parameters for interchange formats are defined in 3.6.

The set of finite floating-point numbers representable within a particular format is determined by the following integer parameters:

— *b* = the radix, 2 or 10
— *p* = the number of digits in the significand (precision)
— *emax* = the maximum exponent *e*
— *emin* = the minimum exponent *e*
  *emin* shall be 1 − *emax* for all formats.

The values of these parameters for each basic format are given in Table 3.2, in which each format is identified by its radix and the number of bits in its encoding. Constraints on these parameters for extended and extendable precision formats are given in 3.7.

Within each format, the following floating-point data shall be represented:

- Signed zero and non-zero floating-point numbers of the form $(-1)^s \times b^e \times m$, where
  - $s$ is 0 or 1.
  - $e$ is any integer $emin \le e \le emax$.
  - $m$ is a number represented by a digit string of the form $d_0 \bullet d_1 d_2 \ldots d_{p-1}$ where $d_i$ is an integer digit $0 \le d_i < b$ (therefore $0 \le m < b$).
- Two infinities, $+\infty$ and $-\infty$.
- Two NaNs, qNaN (quiet) and sNaN (signaling).

These are the only floating-point data represented.

In the foregoing description, the significand $m$ is viewed in a scientific form, with the radix point immediately following the first digit. It is also convenient for some purposes to view the significand as an integer; in which case the finite floating-point numbers are described thus:

- Signed zero and non-zero floating-point numbers of the form $(-1)^s \times b^q \times c$, where
  - $s$ is 0 or 1.
  - $q$ is any integer $emin \le q + p - 1 \le emax$.
  - $c$ is a number represented by a digit string of the form $d_0 d_1 d_2 \ldots d_{p-1}$ where $d_i$ is an integer digit $0 \le d_i < b$ ($c$ is therefore an integer with $0 \le c < b^p$).

This view of the significand as an integer $c$, with its corresponding exponent $q$, describes exactly the same set of zero and non-zero floating-point numbers as the view in scientific form. (For finite floating-point numbers, $e = q + p - 1$ and $m = c \times b^{1-p}$.)

The smallest positive *normal* floating-point number is $b^{emin}$ and the largest is $b^{emax} \times (b - b^{1-p})$. The non-zero floating-point numbers for a format with magnitude less than $b^{emin}$ are called *subnormal* because their magnitudes lie between zero and the smallest normal magnitude. They always have fewer than $p$ significant digits. Every finite floating-point number is an integral multiple of the smallest subnormal magnitude $b^{emin} \times b^{1-p}$.

For a floating-point number that has the value zero, the sign $s$ provides extra information. Although all formats have distinct representations for +0 and −0, the sign of a zero is significant in some circumstances, such as division by zero, but not in others (see 6.3). Binary interchange formats have just one representation each for +0 and −0, but decimal formats have many. In this standard, 0 and ∞ are written without a sign when the sign is not important.

**Table 3.2—Parameters defining basic format floating-point numbers**

| | **Binary format** ($b$=2) | | | **Decimal format** ($b$=10) | |
|---|---|---|---|---|---|
| **parameter** | **binary32** | **binary64** | **binary128** | **decimal64** | **decimal128** |
| $p$, digits | 24 | 53 | 113 | 16 | 34 |
| *emax* | +127 | +1023 | +16383 | +384 | +6144 |

## 3.4 Binary interchange format encodings

Each floating-point number has just one encoding in a binary interchange format. To make the encoding unique, in terms of the parameters in 3.3, the value of the significand $m$ is maximized by decreasing $e$ until either $e=emin$ or $m \geq 1$. After this process is done, if $e=emin$ and $0<m<1$, the floating-point number is subnormal. Subnormal numbers (and zero) are encoded with a reserved biased exponent value.

Representations of floating-point data in the binary interchange formats are uniquely encoded in $k$ bits in the following three fields ordered as shown in Figure 3.1:

a) 1-bit sign $S$
b) $w$-bit biased exponent $E = e + bias$
c) $(t=p-1)$-bit trailing significand field digit string $T = d_1 d_2 \ldots d_{p-1}$; the leading bit of the significand, $d_0$, is implicitly encoded in the biased exponent $E$.

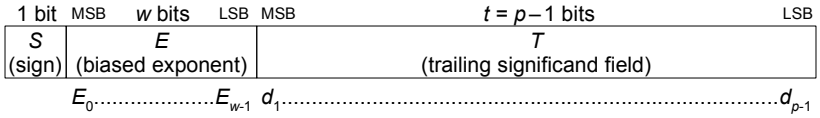


**Figure 3.1—Binary interchange floating-point format**

The values of $k$, $p$, $t$, $w$, and $bias$ for binary interchange formats are listed in Table 3.5 (see 3.6).

The range of the encoding's biased exponent $E$ shall include:

— every integer between 1 and $2^w-2$, inclusive, to encode normal numbers
— the reserved value 0 to encode ±0 and subnormal numbers
— the reserved value $2^w-1$ to encode ±∞ and NaNs.

The representation $r$ of the floating-point datum, and value $v$ of the floating-point datum represented, are inferred from the constituent fields as follows:

a) If $E = 2^w-1$ and $T \neq 0$, then $r$ is qNaN or sNaN and $v$ is NaN regardless of $S$ and then $d_1$ shall exclusively distinguish between qNaN and sNaN (see 6.2.1).
b) If $E = 2^w-1$ and $T=0$ , then $r$ and $v = (-1)^S \times (+\infty)$.
c) If $1 \leq E \leq 2^w-2$, then $r$ is $(S, (E-bias), (1+2^{1-p} \times T))$;
the value of the corresponding floating-point number is $v = (-1)^S \times 2^{E-bias} \times (1+2^{1-p} \times T)$;
thus normal numbers have an implicit leading significand bit of 1.
d) If $E=0$ and $T \neq 0$, then $r$ is $(S, emin, (0+2^{1-p} \times T))$;
the value of the corresponding floating-point number is $v = (-1)^S \times 2^{emin} \times (0+2^{1-p} \times T)$;
thus subnormal numbers have an implicit leading significand bit of 0.
e) If $E=0$ and $T=0$ , then $r$ is $(S, emin, 0)$ and $v = (-1)^S \times (+0)$ (signed zero, see 6.3).

In binary interchange formats, all number and NaN encodings are canonical.

NOTE—Where $k$ is either 64 or a multiple of 32 and ≥ 128, for these encodings all of the following are true (where round( ) rounds to the nearest integer):

$$\begin{aligned} k &= 1 + w + t = w + p = 32 \times \text{ceiling}((p + \text{round}(4 \times \log_2(p + \text{round}(4 \times \log_2(p)) - 13)) - 13)/32) \\ w &= k - t - 1 = k - p = \text{round}(4 \times \log_2(k)) - 13 \\ t &= k - w - 1 = p - 1 = k - \text{round}(4 \times \log_2(k)) + 12 \\ p &= k - w = t + 1 = k - \text{round}(4 \times \log_2(k)) + 13 \\ emax &= bias = 2^{(w-1)} - 1 \\ emin &= 1 - emax = 2 - 2^{(w-1)}. \end{aligned}$$

## 3.5 Decimal interchange format encodings

### 3.5.1 Cohorts

Unlike in a binary floating-point format, in a decimal floating-point format a number might have multiple representations. The set of representations a floating-point number maps to is called the floating-point number's *cohort*; the members of a cohort are distinct *representations* of the same floating-point number. For example, if $c$ is a multiple of 10 and $q$ is less than its maximum allowed value, then ($s$, $q$, $c$) and ($s$, $q+1$, $c/10$) are two representations for the same floating-point number and are members of the same cohort.

While numerically equal, different members of a cohort can be distinguished by the decimal-specific operations (see 5.3.2, 5.5.2, and 5.7.3). The cohorts of different floating-point numbers might have different numbers of members. If a finite non-zero number's representation has $n$ decimal digits from its most significant non-zero digit to its least significant non-zero digit, the representation's cohort will have at most $p-n+1$ members where $p$ is the number of digits of precision in the format.

For example, a one-digit floating-point number might have up to $p$ different representations while a $p$-digit floating-point number with no trailing zeros has only one representation. (An $n$-digit floating-point number might have fewer than $p-n+1$ members in its cohort if it is near the extremes of the format's exponent range.) A zero has a much larger cohort: the cohort of +0 contains a representation for each exponent, as does the cohort of −0.

For decimal arithmetic, besides specifying a numerical result, the arithmetic operations also select a member of the result's cohort according to 5.2. Decimal applications can make use of the additional information cohorts convey.

### 3.5.2 Encodings

Representations of floating-point data in the decimal interchange formats are encoded in $k$ bits in the following three fields, whose detailed layouts and canonical (preferred) encodings are described below.

a) 1-bit sign $S$.

b) A $w+5$ bit combination field $G$ encoding classification and, if the encoded datum is a finite number, the exponent $q$ and four significand bits (1 or 3 of which are implied). The biased exponent $E$ is a $w+2$ bit quantity $q+bias$, where the value of the first two bits of the biased exponent taken together is either 0, 1, or 2.

c) A $t$-bit trailing significand field $T$ that contains $J \times 10$ bits and contains the bulk of the significand. When this field is combined with the leading significand bits from the combination field, the format encodes a total of $p = 3 \times J + 1$ decimal digits.

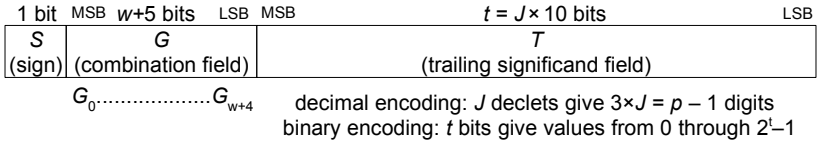


**Figure 3.2—Decimal interchange floating-point formats**

The values of $k$, $p$, $t$, $w$, and $bias$ for decimal interchange formats are listed in Table 3.6 (see 3.6).

The representation $r$ of the floating-point datum, and value $v$ of the floating-point datum represented, are inferred from the constituent fields as follows:

a) If $G_0$ through $G_4$ are 11111, then $v$ is NaN regardless of $S$. Furthermore, if $G_5$ is 1, then $r$ is sNaN; otherwise $r$ is qNaN. The remaining bits of $G$ are ignored, and $T$ is the payload, which can be used to distinguish various NaNs.

   The payload is encoded similarly to finite numbers described below, with $G$ treated as though all bits were zero. The payload corresponds to the significand of finite numbers, interpreted as an

integer with a maximum value of $10^{(3\times J)}-1$, and the exponent field is ignored (it is treated as if it were zero). A NaN is in its preferred (canonical) encoding if the bits $G_6$ through $G_{w+4}$ are zero and the encoding of the payload is canonical.

b) If $G_0$ through $G_4$ are 11110 then $r$ and $v = (-1)^S \times (+\infty)$. The values of the remaining bits in $G$, and $T$, are ignored. The two canonical encodings of infinity have bits $G_5$ through $G_{w+4} = 0$, and $T = 0$.

c) For finite numbers, $r$ is ($S$, $E-bias$, $C$) and $v = (-1)^S \times 10^{(E-bias)} \times C$, where $C$ is the concatenation of the leading significand digit or bits from the combination field $G$ and the trailing significand field $T$, and where the biased exponent $E$ is encoded in the combination field. The encoding within these fields depends on whether the implementation uses the decimal or the binary encoding for the significand.

1) If the implementation uses the *decimal* encoding for the significand, then the least significant $w$ bits of the exponent are $G_5$ through $G_{w+4}$. The most significant two bits of the biased exponent and the decimal digit string $d_0\ d_1 \ldots d_{p-1}$ of the significand are formed from bits $G_0$ through $G_4$ and $T$ as follows:

   i) When the most significant five bits of $G$ are 110xx or 1110x, the leading significand digit $d_0$ is $8+G_4$, a value 8 or 9, and the leading biased exponent bits are $2G_2+G_3$ , a value 0, 1, or 2.

   ii) When the most significant five bits of $G$ are 0xxxx or 10xxx, the leading significand digit $d_0$ is $4G_2+2G_3+G_4$, a value in the range 0 through 7, and the leading biased exponent bits are $2G_0+G_1$, a value 0, 1, or 2. Consequently if $T$ is 0 and the most significant five bits of $G$ are 00000, 01000, or 10000, then $v = (-1)^S \times (+0)$.

   The $p-1 = 3 \times J$ decimal digits $d_1 \ldots d_{p-1}$ are encoded by $T$, which contains $J$ declets encoded in densely packed decimal.

   A canonical significand has only canonical declets, as shown in Tables 3.3 and 3.4. Computational operations accept all 1024 possible declets in operands. Except for quiet-computational operations (see 5.5), computational operations produce only the 1000 canonical declets.

2) Alternatively, if the implementation uses the *binary* encoding for the significand, then:

   i) If $G_0$ and $G_1$ together are one of 00, 01, or 10, then the biased exponent $E$ is formed from $G_0$ through $G_{w+1}$ and the significand is formed from bits $G_{w+2}$ through the end of the encoding (including $T$).

   ii) If $G_0$ and $G_1$ together are 11 and $G_2$ and $G_3$ together are one of 00, 01, or 10, then the biased exponent $E$ is formed from $G_2$ through $G_{w+3}$ and the significand is formed by prefixing the 4 bits $(8+G_{w+4})$ to $T$.

   The maximum value of the binary-encoded significand is the same as that of the corresponding decimal-encoded significand; that is, $10^{(3\times J+1)}-1$ (or $10^{(3\times J)}-1$ when $T$ is used as the payload of a NaN). If the value exceeds the maximum, the significand $c$ is non-canonical and the value used for $c$ is zero.

Computational operations generally produce only canonical significands, and always accept non-canonical significands in operands.

NOTE—Where $k$ is a positive multiple of 32, for these encodings all of the following are true:

$$\begin{aligned} k &= 1 + 5 + w + t &&= 32 \times \text{ceiling}((p+2)/9) \\ w &= k - t - 6 &&= k/16 + 4 \\ t &= k - w - 6 &&= 15 \times k/16 - 10 \\ p &= 3 \times t/10 + 1 &&= 9 \times k/32 - 2 \\ emax &= 3 \times 2^{(w-1)} \\ emin &= 1 - emax \\ bias &= emax + p - 2. \end{aligned}$$

**Decoding densely packed decimal:** Table 3.3 decodes a declet, with 10 bits $\mathbf{b_{(0)}}$ to $\mathbf{b_{(9)}}$, into 3 decimal digits $\mathbf{d_{(1)}}$, $\mathbf{d_{(2)}}$, $\mathbf{d_{(3)}}$. The first column is in binary and an "x" denotes a "don't care" bit. Thus all 1024 possible 10-bit patterns shall be accepted and mapped into 1000 possible 3-digit combinations with some redundancy.

**Table 3.3—Decoding 10-bit densely packed decimal to 3 decimal digits**

| $\mathbf{b_{(6)}, b_{(7)}, b_{(8)}, b_{(3)}, b_{(4)}}$ | $\mathbf{d_{(1)}}$ | $\mathbf{d_{(2)}}$ | $\mathbf{d_{(3)}}$ |
|---|---|---|---|
| 0 x x x x | $4b_{(0)} + 2b_{(1)} + b_{(2)}$ | $4b_{(3)} + 2b_{(4)} + b_{(5)}$ | $4b_{(7)} + 2b_{(8)} + b_{(9)}$ |
| 1 0 0 x x | $4b_{(0)} + 2b_{(1)} + b_{(2)}$ | $4b_{(3)} + 2b_{(4)} + b_{(5)}$ | $8 + b_{(9)}$ |
| 1 0 1 x x | $4b_{(0)} + 2b_{(1)} + b_{(2)}$ | $8 + b_{(5)}$ | $4b_{(3)} + 2b_{(4)} + b_{(9)}$ |
| 1 1 0 x x | $8 + b_{(2)}$ | $4b_{(3)} + 2b_{(4)} + b_{(5)}$ | $4b_{(0)} + 2b_{(1)} + b_{(9)}$ |
| 1 1 1 0 0 | $8 + b_{(2)}$ | $8 + b_{(5)}$ | $4b_{(0)} + 2b_{(1)} + b_{(9)}$ |
| 1 1 1 0 1 | $8 + b_{(2)}$ | $4b_{(0)} + 2b_{(1)} + b_{(5)}$ | $8 + b_{(9)}$ |
| 1 1 1 1 0 | $4b_{(0)} + 2b_{(1)} + b_{(2)}$ | $8 + b_{(5)}$ | $8 + b_{(9)}$ |
| 1 1 1 1 1 | $8 + b_{(2)}$ | $8 + b_{(5)}$ | $8 + b_{(9)}$ |

**Encoding densely packed decimal:** Table 3.4 encodes 3 decimal digits $\mathbf{d_{(1)}}$, $\mathbf{d_{(2)}}$, and $\mathbf{d_{(3)}}$, each having 4 bits which can be expressed by a second subscript $\mathbf{d_{(1,0:3)}}$, $\mathbf{d_{(2,0:3)}}$, and $\mathbf{d_{(3,0:3)}}$, where bit 0 is the most significant and bit 3 the least significant, into a declet, with 10 bits $\mathbf{b_{(0)}}$ to $\mathbf{b_{(9)}}$. Most computational operations generate only the 1000 canonical 10-bit patterns defined by Table 3.4.

**Table 3.4—Encoding 3 decimal digits to 10-bit densely packed decimal**

| $\mathbf{d_{(1,0)}, d_{(2,0)}, d_{(3,0)}}$ | $\mathbf{b_{(0)}, b_{(1)}, b_{(2)}}$ | $\mathbf{b_{(3)}, b_{(4)}, b_{(5)}}$ | $\mathbf{b_{(6)}}$ | $\mathbf{b_{(7)}, b_{(8)}, b_{(9)}}$ |
|---|---|---|---|---|
| 0 0 0 | $d_{(1,1:3)}$ | $d_{(2,1:3)}$ | 0 | $d_{(3,1:3)}$ |
| 0 0 1 | $d_{(1,1:3)}$ | $d_{(2,1:3)}$ | 1 | 0, 0, $d_{(3,3)}$ |
| 0 1 0 | $d_{(1,1:3)}$ | $d_{(3,1:2)}$, $d_{(2,3)}$ | 1 | 0, 1, $d_{(3,3)}$ |
| 0 1 1 | $d_{(1,1:3)}$ | 1, 0, $d_{(2,3)}$ | 1 | 1, 1, $d_{(3,3)}$ |
| 1 0 0 | $d_{(3,1:2)}$, $d_{(1,3)}$ | $d_{(2,1:3)}$ | 1 | 1, 0, $d_{(3,3)}$ |
| 1 0 1 | $d_{(2,1:2)}$, $d_{(1,3)}$ | 0, 1, $d_{(2,3)}$ | 1 | 1, 1, $d_{(3,3)}$ |
| 1 1 0 | $d_{(3,1:2)}$, $d_{(1,3)}$ | 0, 0, $d_{(2,3)}$ | 1 | 1, 1, $d_{(3,3)}$ |
| 1 1 1 | 0, 0, $d_{(1,3)}$ | 1, 1, $d_{(2,3)}$ | 1 | 1, 1, $d_{(3,3)}$ |

The 24 non-canonical patterns of the form 01x11x111x, 10x11x111x, or 11x11x111x (where an "x" denotes a "don't care" bit) are not generated in the result of a computational operation. However, as listed in Table 3.3, these 24 bit patterns do map to values in the range 0 through 999. The bit pattern in a NaN trailing significand field can affect how the NaN is propagated (see 6.2).

## 3.6 Interchange format parameters

Interchange formats support the exchange of floating-point data between implementations. In each radix, the precision and range of an interchange format is defined by its size; interchange of a floating-point datum of a given size is therefore always exact with no possibility of overflow or underflow.

This standard defines binary interchange formats of widths 16, 32, 64, and 128 bits, and in general for any multiple of 32 bits of at least 128 bits. Decimal interchange formats are defined for any multiple of 32 bits of at least 32 bits.

The parameters *p* and *emax* for every interchange format width are shown in Table 3.5 for binary interchange formats and in Table 3.6 for decimal interchange formats. The encodings for the interchange formats are as described in 3.4 and 3.5.2; the encoding parameters for each interchange format width are also shown in Tables 3.5 and 3.6.

**Table 3.5—Binary interchange format parameters**

| Parameter | binary16 | binary32 | binary64 | binary128 | binary{*k*} ($k \geq 128$) |
|---|---|---|---|---|---|
| *k*, storage width in bits | 16 | 32 | 64 | 128 | multiple of 32 |
| *p*, precision in bits | 11 | 24 | 53 | 113 | $k - \text{round}(4 \times \log_2(k)) + 13$ |
| *emax*, maximum exponent *e* | 15 | 127 | 1023 | 16383 | $2^{(k-p-1)} - 1$ |
| *Encoding parameters* | | | | | |
| *bias*, $E - e$ | 15 | 127 | 1023 | 16383 | *emax* |
| sign bit | 1 | 1 | 1 | 1 | 1 |
| *w*, exponent field width in bits | 5 | 8 | 11 | 15 | $\text{round}(4 \times \log_2(k)) - 13$ |
| *t*, trailing significand field width in bits | 10 | 23 | 52 | 112 | $k - w - 1$ |
| *k*, storage width in bits | 16 | 32 | 64 | 128 | $1 + w + t$ |

In Table 3.5, round( ) rounds to the nearest integer.

For example, binary256 would have $p = 237$ and $emax = 262143$.

**Table 3.6—Decimal interchange format parameters**

| Parameter | decimal32 | decimal64 | decimal128 | decimal{*k*} ($k \geq 32$) |
|---|---|---|---|---|
| *k*, storage width in bits | 32 | 64 | 128 | multiple of 32 |
| *p*, precision in digits | 7 | 16 | 34 | $9 \times k/32 - 2$ |
| *emax* | 96 | 384 | 6144 | $3 \times 2^{(k/16+3)}$ |
| *Encoding parameters* | | | | |
| *bias*, $E - q$ | 101 | 398 | 6176 | $emax + p - 2$ |
| sign bit | 1 | 1 | 1 | 1 |
| *w*+5, combination field width in bits | 11 | 13 | 17 | $k/16 + 9$ |
| *t*, trailing significand field width in bits | 20 | 50 | 110 | $15 \times k/16 - 10$ |
| *k*, storage width in bits | 32 | 64 | 128 | $1 + 5 + w + t$ |

For example, decimal256 would have $p = 70$ and $emax = 1572864$.

NOTE—This standard defines the representation of individual data as conceptual Level 4 entities. Applications exchanging data between different implementations must communicate certain parameters that describe the formats and layout of the data. Besides issues such as byte order which affect all data interchange, certain implementation options allowed by this standard must also be considered:

— for binary formats, how signaling NaNs are distinguished from quiet NaNs

— for decimal formats, whether binary or decimal encoding is used.

This standard does not define how these parameters are to be communicated.

## 3.7 Extended and extendable precisions

Extended and extendable precision formats are recommended for extending the precisions used for arithmetic beyond the basic formats. Specifically:

— An **extended precision format** is an arithmetic format that extends a supported basic format with both wider precision and wider range.

— An **extendable precision format** is an arithmetic format with a precision and range that are defined under user control.

These formats are characterized by the parameters *b, p,* and *emax,* which may match those of an interchange format and shall:

— provide all the representations of floating-point data defined in terms of those parameters in 3.2 and 3.3

— provide all the operations of this standard, as defined in Clause 5, for that format.

Encodings in these formats should be fixed width and may match those of an interchange format. Each representation of a floating-point number has a unique canonical encoding and may have non-canonical encodings. Each NaN in these formats has a payload, which might encode diagnostic information. Each NaN payload has a canonical encoding and may have non-canonical encodings. All other aspects of encodings for these formats are implementation defined.

Language standards should define mechanisms supporting extendable precision for each supported radix. Language standards supporting extendable precision shall permit users to specify $p$ and *emax*. Language standards shall also allow the specification of an extendable precision by specifying $p$ alone; in this case *emax* shall be defined by the language standard to be at least $1000 \times p$ when $p$ is $\geq 237$ bits in a binary format or $p$ is $\geq 51$ digits in a decimal format.

Language standards or implementations should support an extended precision format that extends the widest basic format that is supported in that radix. Table 3.7 specifies the minimum precision and exponent range of the extended precision format for each basic format.

**Table 3.7—Extended format parameters for floating-point numbers**

| | Extended formats associated with: | | | | |
|---|---|---|---|---|---|
| **Parameter** | **binary32** | **binary64** | **binary128** | **decimal64** | **decimal128** |
| $p$ digits $\geq$ | 32 | 64 | 128 | 22 | 40 |
| *emax* $\geq$ | 1023 | 16383 | 65535 | 6144 | 24576 |

NOTE 1—For extended formats, the minimum exponent range is that of the next wider basic format, if there is one, while the minimum precision is intermediate between a given basic format and the next wider basic format.

NOTE 2—For interchange of binary floating-point data, the width $k$ in bits of the smallest format that will allow the encoding of a significand of at least $p$ bits is given by:

$k = 32 \times \text{ceiling}((p + \text{round}(4 \times \log_2(p + \text{round}(4 \times \log_2(p)) - 13)) - 13)/32)$, where round( ) rounds to the nearest integer and $p \geq 113$; for smaller values of $p$, see Table 3.5.

For interchange of decimal floating-point data, the width $k$ in bits of the smallest format that will allow the encoding of a significand of at least $p$ digits is given by:

$k = 32 \times \text{ceiling}((p + 2)/9)$, where $p \geq 1$.

In both cases the chosen format might have a larger precision (see 3.4 and 3.5.2).

NOTE 3—For binary formats, the precision $p$ should be at least 3, as some numerical properties do not hold for lower precisions.

Similarly, *emax* should be at least 2 to support the operations listed in 9.2.

# 4. Attributes and rounding

## 4.1 Attribute specification

An attribute is logically associated with a program block to modify its numerical and exception semantics. A user can specify a constant value for an attribute parameter.

Some attributes have the effect of an implicit parameter to most individual operations of this standard; language standards shall specify

— rounding-direction attributes (see 4.3)

and should specify

— alternate exception handling attributes (see Clause 8).

Other attributes change the mapping of language expressions into operations of this standard; language standards that permit more than one such mapping should provide support for:

— preferredWidth attributes (see 10.3)
— value-changing optimization attributes (see 10.4)
— reproducibility attributes (see Clause 11).

For attribute specification, the implementation shall provide language-defined means, such as compiler directives, to specify a constant value for the attribute parameter for the operations in a block; the scope of the attribute value is the block with which it is associated. Language standards shall provide for constant specification of the default and each specific value of the attribute.

## 4.2 Dynamic modes for attributes

Attributes in this standard shall be supported with the constant specification of 4.1. Particularly to support debugging, language standards should also support dynamic-mode specification of attributes.

With dynamic-mode specification, a user can specify that the attribute parameter assumes the value of a dynamic-mode variable whose value might not be known until program execution. This standard does not specify the underlying implementation mechanisms for constant attributes or dynamic modes.

For dynamic-mode specification, the implementation shall provide language-defined means to specify that the attribute parameter assumes the value of a dynamic-mode variable for the operations within the scope of the dynamic-mode specification in a block. The implementation initializes a dynamic-mode variable to the default value for the dynamic mode. Within its language-defined (dynamic) scope, changes to the value of a dynamic-mode variable are under the control of the user via the operations in 9.3.1 and 9.3.2.

The following aspects of dynamic-mode variables are language-defined; language standards may explicitly defer the definitions to implementations:

— The precedence of static attribute specifications and dynamic-mode assignments.
— The effect of changing the value of the dynamic-mode variable in an asynchronous event, such as in another thread or signal handler.
— Whether the value of the dynamic-mode variable can be determined by non-programmatic means, such as a debugger.

NOTE—A constant value for an attribute can be specified and meet the requirements of 4.1 by a dynamic mode specification with appropriate scope of that constant value.

## 4.3 Rounding-direction attributes

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception, underflow, or overflow when appropriate (see Clause 7). Except where stated otherwise, every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the attributes in this clause.

Except where stated otherwise, the rounding-direction attribute affects all computational operations that might be inexact. Inexact numeric floating-point results always have the same sign as the unrounded result.

The rounding-direction attribute affects the signs of exact zero sums (see 6.3), and also affects the thresholds beyond which overflow (see 7.4) and underflow (see 7.5) are signaled.

Implementations supporting both decimal and binary formats shall provide separate rounding-direction attributes for binary and decimal, the binary rounding direction and the decimal rounding direction. Operations returning results in a floating-point format shall use the rounding-direction attribute associated with the radix of the results. Operations converting from an operand in a floating-point format to a result in integer format or to an external character sequence (see 5.8 and 5.12) shall use the rounding-direction attribute associated with the radix of the operand.

NaNs are not rounded (but see 6.2.3).

### 4.3.1 Rounding-direction attributes to nearest

In the following two rounding-direction attributes, an infinitely precise result with magnitude at least $b^{emax} \times (b - \frac{1}{2} b^{1-p})$ shall round to $\infty$ with no change in sign; here *emax* and *p* are determined by the destination format (see 3.3). With:

- roundTiesToEven, the floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with an even least significant digit shall be delivered; if that is not possible, the one larger in magnitude shall be delivered (this can happen for one-digit precision, possible with convertToDecimalCharacter for example, as when rounding 9.5 to one digit in which case both 9 and $1 \times 10^1$ have odd significands)
- roundTiesToAway, the floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with larger magnitude shall be delivered.

### 4.3.2 Directed rounding attributes

Three other user-selectable rounding-direction attributes are defined, the directed rounding attributes roundTowardPositive, roundTowardNegative, and roundTowardZero. With:

- roundTowardPositive, the result shall be the format's floating-point number (possibly $+\infty$) closest to and no less than the infinitely precise result
- roundTowardNegative, the result shall be the format's floating-point number (possibly $-\infty$) closest to and no greater than the infinitely precise result
- roundTowardZero, the result shall be the format's floating-point number closest to and no greater in magnitude than the infinitely precise result.

### 4.3.3 Rounding attribute requirements

An implementation of this standard shall provide roundTiesToEven and the three directed rounding attributes. A decimal format implementation of this standard shall provide roundTiesToAway as a user-selectable rounding-direction attribute. The rounding attribute roundTiesToAway is not required for a binary format implementation.

The roundTiesToEven rounding-direction attribute shall be the default rounding-direction attribute for results in binary formats. The default rounding-direction attribute for results in decimal formats is language-defined, but should be roundTiesToEven.

# 5. Operations

## 5.1 Overview

All conforming implementations of this standard shall provide the operations listed in this clause for all supported arithmetic formats, except as stated below. Unless otherwise specified, each of the computational operations specified by this standard that returns a numeric result shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format (see Clause 4 and Clause 7). Clause 6 augments the following specifications to cover ±0, ±∞, and NaN. Clause 7 describes default exception handling.

In this standard, operations are written as named functions; in a specific programming environment they might be denoted by operators, or by families of format-specific functions, or by operations or functions whose names might differ from those in this standard.

Operations are broadly classified into four groups according to the kinds of results and exceptions they produce:

- General-computational operations produce floating-point or integer results, round all results according to Clause 4, and might signal the floating-point exceptions of Clause 7.
- Quiet-computational operations produce floating-point results and do not signal floating-point exceptions.
- Signaling-computational operations produce no floating-point results and might signal floating-point exceptions; comparisons are signaling-computational operations.
- Non-computational operations do not produce floating-point results and do not signal floating-point exceptions.

Operations in the first three groups are referred to collectively as "computational operations".

Operations are also classified in two ways according to the relationship between the result format and the operand formats:

- homogeneous operations, in which the floating-point operands and floating-point result are all of the same format
- *formatOf* operations, which indicate the format of the result, independent of the formats of the operands.

Language standards might permit other kinds of operations and combinations of operations in expressions. By their expression evaluation rules, language standards specify when and how such operations and expressions are mapped into the operations of this standard. Operations (except re-encoding operations) do not have to accept operands or produce results of differing encodings.

Except as specified otherwise in 5.5, operation results shall be canonical.

In the operation descriptions that follow, operand and result formats are indicated by:

- *source* to represent homogeneous floating-point operand formats
- *source1, source2, source3* to represent non-homogeneous floating-point operand formats
- *int* to represent integer operand formats
- *boolean* to represent a value of *false* or *true* (for example, 0 or 1)
- *enum* to represent one of a small set of enumerated values
- *sourceFormat* to represent a destination format that is the same as the *source* format
- *integralFormat* to represent a format that contains integral values
- *logBFormat* to represent an *integralFormat* for the destination of the *logB* operation and the scale exponent operand of the *scaleB* operation
- *decimalCharacterSequence* to represent a decimal character sequence
- *hexCharacterSequence* to represent a hexadecimal-significand character sequence

— *conversionSpecification* to represent a language dependent conversion specification
— *decimal* to represent a supported decimal floating-point format
— *decimalEncoding* to represent a decimal floating-point format encoded in decimal
— *binaryEncoding* to represent a decimal floating-point format encoded in binary
— *exceptionGroup* to represent a set of exceptions as a set of *booleans*
— *flags* to represent a set of status flags
— *binaryRoundingDirection* to represent the rounding direction for binary
— *decimalRoundingDirection* to represent the rounding direction for decimal
— *modeGroup* to represent dynamically-specifiable modes
— *void* to indicate that an operation has no explicit operand or has no explicit result; the operand or result might be implicit.

*formatOf* indicates that the name of the operation specifies the floating-point destination *format*, which might be different from the floating-point operands' formats. There are *formatOf* versions of these operations for every supported arithmetic format.

*intFormatOf* indicates that the name of the operation specifies the integer destination format.

In the operation descriptions that follow, languages define which of their types correspond to operands and results. Languages with both signed and unsigned integer types should support both signed and unsigned *int* and *intFormatOf* operands and results.

## 5.2 Decimal exponent calculation

As discussed in 3.5, a floating-point number might have multiple representations in a decimal format. Therefore, decimal arithmetic involves not only computing the proper numerical result but also selecting the proper member of that floating-point number's cohort.

Except for the **quantize** operation, the value of a floating-point result (and hence its cohort) is determined by the operation and the operands' values; it is never dependent on the representation or encoding of an operand.

The selection of a particular representation for a floating-point result is dependent on the operands' representations, as described below, but is not affected by their encoding.

For all computational operations except where stated otherwise, if the result is inexact the cohort member of least possible exponent is used to get the maximum number of significant digits. If the result is exact, the cohort member is selected based on the preferred exponent for a result of that operation, a function of the exponents of the inputs. Thus for finite $x$, depending on the representation of zero, $0+x$ might result in a different member of $x$'s cohort. If the result's cohort does not include a member with the preferred exponent, the member with the exponent closest to the preferred exponent is used.

For **quantize** and **roundToIntegralExact**, a finite result has the preferred exponent, whether or not the result is exact.

In the operation descriptions that follow, $Q(x)$ is the exponent $q$ of the representation of a finite floating-point number $x$. If $x$ is infinite, $Q(x)$ is $+\infty$.

## 5.3 Homogeneous general-computational operations

### 5.3.1 General operations

Implementations shall provide the following homogeneous general-computational operations for all supported arithmetic formats:

— *sourceFormat* **roundToIntegralTiesToEven**(*source*)
*sourceFormat* **roundToIntegralTiesToAway**(*source*)
*sourceFormat* **roundToIntegralTowardZero**(*source*)
*sourceFormat* **roundToIntegralTowardPositive**(*source*)
*sourceFormat* **roundToIntegralTowardNegative**(*source*)

See 5.9 for details.

The preferred exponent is max(Q(*source*), 0).

— *sourceFormat* **roundToIntegralExact**(*source*)

See 5.9 for details.

The preferred exponent is max(Q(*source*), 0), even when the inexact exception is signaled.

— *sourceFormat* **nextUp**(*source*)
*sourceFormat* **nextDown**(*source*)

**nextUp**(*x*) is the least floating-point number in the format of *x* that compares greater than *x*. If *x* is the negative number of least magnitude in *x*'s format, **nextUp**(*x*) is −0. **nextUp**(±0) is the positive number of least magnitude in *x*'s format. **nextUp**(+∞) is +∞, and **nextUp**(−∞) is the finite negative number largest in magnitude. When *x* is a NaN, then the result is according to 6.2. **nextUp**(x) is quiet except for signaling NaNs.

The preferred exponent is the least possible.

**nextDown**(*x*) is −**nextUp**(−*x*).

— *sourceFormat* **remainder**(*source, source*)

When $y \neq 0$, the remainder $r = \mathbf{remainder}(x, y)$ is defined for finite $x$ and $y$ regardless of the rounding-direction attribute by the mathematical relation $r = x - y \times n$ , where $n$ is the integer nearest the exact number $x/y$; whenever $|n - x/y| = ½$, then $n$ is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of $x$. **remainder**($x$, ∞) is $x$ for finite $x$.

The preferred exponent is min(Q($x$), Q($y$)).

NOTE—The minNum and maxNum operations of the 2008 version of the standard have been replaced by the recommended operations of 9.6.

### 5.3.2 Decimal operations

Implementations supporting decimal formats shall provide the following homogeneous general-computational operation for all supported decimal arithmetic formats:

— *sourceFormat* **quantize**(*source, source*)

For finite decimal operands $x$ and $y$ of the same format, **quantize**($x$, $y$) is a floating-point number in the same format that has, if possible, the same numerical value as $x$ and the same quantum as $y$. If the exponent is being increased, rounding according to the applicable rounding-direction attribute might occur: the result is a different floating-point representation and the inexact exception is signaled if the result does not have the same numerical value as $x$. If the exponent is being decreased and the significand of the result would have more than $p$ digits, the invalid operation exception is signaled and the result is a NaN. If one or both operands are NaN, the rules in 6.2 are followed. Otherwise if only one operand is infinite then the invalid operation exception is signaled and the result is a NaN. If both operands are infinite then the result is canonical ∞ with the sign of $x$. **quantize** does not signal underflow or overflow.

The preferred exponent is Q($y$).

Implementations supporting decimal formats should provide the following homogeneous general-computational operation for all supported decimal arithmetic formats:

— *sourceFormat* **quantum**(*source*)

If $x$ is a finite number, the operation **quantum**($x$) is the number represented by (0, $q$, 1) where $q$ is the exponent of $x$. If $x$ is infinite, **quantum**($x$) is +∞ with no exception.

The preferred exponent is Q($x$).

### 5.3.3 *logBFormat* operations

Implementations shall provide the following general-computational operations for all supported floating-point formats available for arithmetic.

For each supported arithmetic format, languages define an associated *logBFormat* to contain the integral values of **logB**($x$). The *logBFormat* shall have enough range to include all integers between $\pm 2 \times (emax + p)$ inclusive, which includes the scale factors for scaling between the finite numbers of largest and smallest magnitude.

— *sourceFormat* **scaleB**(*source, logBFormat*)

**scaleB**($x$, $N$) is $x \times b^N$ for integral values $N$. The result is computed as if the exact product were formed and then rounded to the destination format, subject to the applicable rounding-direction attribute. When *logBFormat* is a floating-point format, the behavior of **scaleB** is language-defined when the second operand is non-integral. For non-zero values of $N$, **scaleB**(±0, $N$) returns ±0 and **scaleB**(±∞, N) returns ±∞. For zero values of $N$, **scaleB**($x$, $N$) returns $x$.

The preferred exponent is Q($x$) + $N$.

— *logBFormat* **logB**(*source*)

**logB**($x$) is the exponent $e$ of $x$, a signed integral value, determined as though $x$ were represented with infinite range and minimum exponent. Thus $1 \le$ **scaleB**($x$, −**logB**($x$)) $< b$ when $x$ is positive and finite. **logB**(1) is +0.

When *logBFormat* is a floating-point format, **logB**(NaN) is a NaN, **logB**(∞) is +∞, and **logB**(0) is −∞ and signals the divideByZero exception. When *logBFormat* is an integer format, then **logB**(NaN), **logB**(∞), and **logB**(0) return language-defined values outside the range $\pm 2 \times (emax + p - 1)$ and signal the invalid operation exception.

The preferred exponent is 0.

NOTE—For positive finite $x$, the value of **logB**($x$) is floor($\log_2(x)$) in a binary format, and is floor($\log_{10}(x)$) in a decimal format.

## 5.4 formatOf general-computational operations

### 5.4.1 Arithmetic operations

Implementations shall provide the following *formatOf* general-computational operations, for destinations of all supported arithmetic formats, and, for each destination format, for operands of all supported arithmetic formats with the same radix as the destination format.

— *formatOf*-**addition**(*source1, source2*)

The operation **addition**($x$, $y$) computes $x+y$.

The preferred exponent is min(Q($x$), Q($y$)).

— *formatOf*-**subtraction**(*source1, source2*)

The operation **subtraction**($x$, $y$) computes $x-y$.

The preferred exponent is min(Q($x$), Q($y$)).

— *formatOf*-**multiplication**(*source1, source2*)

The operation **multiplication**($x$, $y$) computes $x \times y$.

The preferred exponent is Q($x$)+Q($y$).

— *formatOf*-**division**(*source1, source2*)

The operation **division**($x$, $y$) computes $x/y$.

The preferred exponent is Q($x$) − Q($y$).

— *formatOf*-**squareRoot**(*source1*)

The operation **squareRoot**($x$) computes $\sqrt{x}$. It has a positive sign for all operands $\geq 0$, except that **squareRoot**(−0) shall be −0.

The preferred exponent is floor(Q($x$)/2).

— *formatOf*-**fusedMultiplyAdd**(*source1, source2, source3*))

The operation **fusedMultiplyAdd**($x$, $y$, $z$) computes $(x \times y)+z$ as if with unbounded range and precision, rounding only once to the destination format. An underflow, overflow, or inexact exception (see Clause 7) can only arise due to the rounding of the exact value $(x \times y)+z$. Thus at most one exception is signaled per fused operation invocation. **fusedMultiplyAdd** differs from a **multiplication** operation followed by an **addition** operation.

The preferred exponent is min(Q($x$)+Q($y$), Q($z$)).

— *formatOf*-**convertFromInt(***int*)

It shall be possible to convert from all supported signed and unsigned integer formats to all supported arithmetic formats. Integral values are converted exactly from integer formats to floating-point formats whenever the value is representable in both formats. If the converted value is not exactly representable in the destination format, the result is determined according to the applicable rounding-direction attribute, and an inexact or floating-point overflow exception arises as specified in Clause 7, just as with arithmetic operations. The signs of integer zeros are preserved. Integer zeros without signs are converted to +0.

The preferred exponent is 0.

Implementations shall provide the following *intFormatOf* general-computational operations for destinations of all supported integer formats and for operands of all supported arithmetic formats.

— *intFormatOf*-**convertToIntegerTiesToEven**(*source*)
*intFormatOf*-**convertToIntegerTowardZero**(*source*)
*intFormatOf*-**convertToIntegerTowardPositive**(*source*)
*intFormatOf*-**convertToIntegerTowardNegative**(*source*)
*intFormatOf*-**convertToIntegerTiesToAway**(*source*)

See 5.8 for details.

— *intFormatOf*-**convertToIntegerExactTiesToEven**(*source*)
*intFormatOf*-**convertToIntegerExactTowardZero**(*source*)
*intFormatOf*-**convertToIntegerExactTowardPositive**(*source*)
*intFormatOf*-**convertToIntegerExactTowardNegative**(*source*)
*intFormatOf*-**convertToIntegerExactTiesToAway**(*source*)

See 5.8 for details.

NOTE—Implementations might provide some of the operations in this subclause, and the **convertFormat** operations in 5.4.2, as sequences of one or more of a subset of the operations in subclause 5.4 when those sequences produce the correct numerical value, quantum, and exception results.

### 5.4.2 Conversion operations for floating-point formats and decimal character sequences

Implementations shall provide the following *formatOf* conversion operations from all supported floating-point formats to all supported floating-point formats, as well as conversions to and from decimal character sequences. Some format conversion operations produce results in a different radix than the operands.

— *formatOf*-**convertFormat**(*source*)

If the conversion is to a format in a different radix or to a narrower precision in the same radix, the result shall be rounded as specified in Clause 4. Conversion to a format with the same radix but wider precision and range is always exact.

For inexact conversions from binary to decimal formats, the preferred exponent is the least possible. For exact conversions from binary to decimal formats, the preferred exponent is 0.

For conversions between decimal formats, the preferred exponent is Q(*source*).

NOTE—A *formatOf*-**convertFormat** operation with identical source and destination formats is a canonicalizing operation that signals the invalid operation exception for signaling NaN operands, unlike the **copy** operation (5.5.1).

— *formatOf*-**convertFromDecimalCharacter**(*decimalCharacterSequence*)

See 5.12 for details. The preferred exponent is Q(*decimalCharacterSequence*), which is the exponent value *q* of the last digit in the significand of the decimalCharacterSequence.

— *decimalCharacterSequence* **convertToDecimalCharacter**(*source, conversionSpecification*)

See 5.12 for details. The *conversionSpecification* specifies the precision and formatting of the *decimalCharacterSequence* result.

### 5.4.3 Conversion operations for binary formats

Implementations shall provide the following *formatOf* conversion operations to and from all supported binary floating-point formats.

— *formatOf*-**convertFromHexCharacter**(*hexCharacterSequence*)

See 5.12 for details.

— *hexCharacterSequence* **convertToHexCharacter**(*source, conversionSpecification*)

See 5.12 for details. The *conversionSpecification* specifies the precision and formatting of the *hexCharacterSequence* result.

## 5.5 Quiet-computational operations

### 5.5.1 Sign bit operations

Implementations shall provide the following homogeneous quiet-computational sign bit operations for all supported arithmetic interchange formats; they only affect the sign bit. The operations treat floating-point numbers and NaNs alike, and signal no exception. These operations may propagate non-canonical encodings.

— *sourceFormat* **copy**(*source*)
*sourceFormat* **negate**(*source*)
*sourceFormat* **abs**(*source*)

**copy**(*x*) copies a floating-point operand *x* to a destination in the same format, with no change to the sign bit.

**negate**(*x*) copies a floating-point operand *x* to a destination in the same format, reversing the sign bit. **negate**(*x*) is not the same as **subtraction**(0,*x*) (see 6.3).

**abs**(*x*) copies a floating-point operand *x* to a destination in the same format, setting the sign bit to 0 (positive).

— *sourceFormat* **copySign**(*source*, *source*)

**copySign**(*x, y*) copies a floating-point operand *x* to a destination in the same format as *x*, but with the sign bit of *y*.

This standard does not specify encodings for non-interchange formats.

For all supported non-interchange formats, the implementation shall provide **copy**, **negate**, **abs**, and **copySign** operations that match the sign bit operations above at the representation level (see 3.2 and 3.3):

For numeric *x*,

**copy**(*x*) has the same representation as *x*, with no change of sign

**negate**(*x*) has the same representation as *x*, but with the opposite sign

**abs**(*x*) has the same representation as *x*, but with the sign 0 (positive)

**copySign**(*x, y*) has the same representation as *x*, but with the sign of *y* if *y* is numeric and with unspecified sign if *y* is a NaN.

For *x* a NaN, the results of the operations have the same representation as *x* (qNaN or sNaN).

The operations for non-interchange formats should follow the specification for sign bit operations for interchange formats if the encoding permits.

### 5.5.2 Decimal re-encoding operations

For each supported decimal interchange format, the implementation shall provide the following operations to convert between the decimal format and the two encodings for that format (see 3.5.2). These operations enable portable programs that are independent of the implementation's encoding for decimal formats to access data represented with either encoding. If the significand encoding of the result format is the same as the significand encoding of the operand format, then these operations should be **copy** operations. These operations may propagate non-canonical encodings; if the significand encoding of the result format is different from the significand encoding of the operand format, then the result should be canonical.

— *decimalEncoding* **encodeDecimal**(*decimal*)
Encodes the value of the operand using decimal encoding.

— *decimal* **decodeDecimal**(*decimalEncoding*)
Decodes the decimal-encoded operand.

— *binaryEncoding* **encodeBinary**(*decimal*)
Encodes the value of the operand using the binary encoding.

— *decimal* **decodeBinary(***binaryEncoding*)
Decodes the binary-encoded operand.

## 5.6 Signaling-computational operations

### 5.6.1 Comparisons

Implementations shall provide the following comparison operations, for all supported floating-point operands of the same radix in arithmetic formats:

— *boolean* **compareQuietEqual**(*source1, source2*)
*boolean* **compareQuietNotEqual**(*source1, source2*)
*boolean* **compareSignalingEqual**(*source1, source2*)
*boolean* **compareSignalingGreater**(*source1, source2*)
*boolean* **compareSignalingGreaterEqual**(*source1, source2*)
*boolean* **compareSignalingLess**(*source1, source2*)
*boolean* **compareSignalingLessEqual**(*source1, source2*)
*boolean* **compareSignalingNotEqual**(*source1, source2*)
*boolean* **compareSignalingNotGreater**(*source1, source2*)
*boolean* **compareSignalingLessUnordered**(*source1, source2*)
*boolean* **compareSignalingNotLess**(*source1, source2*)
*boolean* **compareSignalingGreaterUnordered**(*source1, source2*)
*boolean* **compareQuietGreater**(*source1, source2*)
*boolean* **compareQuietGreaterEqual**(*source1, source2*)
*boolean* **compareQuietLess**(*source1, source2*)
*boolean* **compareQuietLessEqual**(*source1, source2*)
*boolean* **compareQuietUnordered**(*source1, source2*)
*boolean* **compareQuietNotGreater(***source1, source2*)
*boolean* **compareQuietLessUnordered**(*source1, source2*)
*boolean* **compareQuietNotLess(***source1, source2*)
*boolean* **compareQuietGreaterUnordered**(*source1, source2*)
*boolean* **compareQuietOrdered(***source1, source2*).

See 5.11 for details.

## 5.7 Non-computational operations

### 5.7.1 Conformance predicates

Implementations shall provide the following non-computational operations, true if and only if the indicated conditions are true:

— *boolean* **is754version1985**(*void*)

**is754version1985**( ) is true if and only if this programming environment conforms to the 1985 version of the standard.

— *boolean* **is754version2008**(*void*)

**is754version2008**( ) is true if and only if this programming environment conforms to the 2008 version of the standard.

— *boolean* **is754version2019**(*void*)

**is754version2019**( ) is true if and only if this programming environment conforms to this standard.

Implementations should make these predicates available at translation time (if applicable) in cases where their values can be determined at that point.

### 5.7.2 General operations

Implementations shall provide the following non-computational operations for all supported arithmetic formats and should provide them for all supported interchange formats. They are never exceptional, even for signaling NaNs.

— *enum* **class**(*source*)

**class**(*x*) tells which of the following ten classes *x* falls into:

signalingNaN
quietNaN
negativeInfinity
negativeNormal
negativeSubnormal
negativeZero
positiveZero
positiveSubnormal
positiveNormal
positiveInfinity.

— *boolean* **isSignMinus**(*source*)

**isSignMinus**(*x*) is true if and only if *x* has negative sign. **isSignMinus** applies to zeros and NaNs as well.

— *boolean* **isNormal**(*source*)

**isNormal**(*x*) is true if and only if *x* is normal (not zero, subnormal, infinite, or NaN).

— *boolean* **isFinite**(*source*)

**isFinite**(*x*) is true if and only if *x* is zero, subnormal or normal (not infinite or NaN).

— *boolean* **isZero**(*source*)

**isZero**(*x*) is true if and only if *x* is ±0.

— *boolean* **isSubnormal**(*source*)

**isSubnormal**(*x*) is true if and only if *x* is subnormal.

— *boolean* **isInfinite**(*source*)

**isInfinite**(*x*) is true if and only if *x* is infinite.

— *boolean* **isNaN**(*source*)

**isNaN**(*x*) is true if and only if *x* is a NaN.

— *boolean* **isSignaling**(*source*)

**isSignaling**(*x*) is true if and only if *x* is a signaling NaN.

— *boolean* **isCanonical**(*source*)

**isCanonical**(*x*) is true if and only if *x* is a finite number, infinity, or NaN that is canonical. Implementations should extend **isCanonical**(*x*) to formats that are not interchange formats in ways appropriate to those formats, which might, or might not, have finite numbers, infinities, or NaNs that are non-canonical.

— *enum* **radix**(*source*)

**radix**(*x*) is the radix *b* of the format of *x*, that is, two or ten.

— *boolean* **totalOrder**(*source, source*)

**totalOrder**(*x, y*) is defined in 5.10.

— *boolean* **totalOrderMag**(*source, source*)

**totalOrderMag**(*x, y*) is **totalOrder**(**abs**(*x*), **abs**(*y*)).

Implementations should make these predicates available at translation time (if applicable) in cases where their values can be determined at that point.

### 5.7.3 Decimal operation

Implementations supporting decimal formats shall provide the following non-computational operation for all supported decimal arithmetic formats:

— *boolean* **sameQuantum**(*source, source*)

For numerical decimal operands $x$ and $y$ of the same format, **sameQuantum**($x$, $y$) is true if the exponents of $x$ and $y$ are the same, that is, Q($x$) = Q($y$), and false otherwise. **sameQuantum**(NaN, NaN) and **sameQuantum**($\infty$, $\infty$) are true; if exactly one operand is infinite or exactly one operand is a NaN, **sameQuantum** is false. **sameQuantum** signals no exception, even if an argument is a signaling NaN.

### 5.7.4 Operations on subsets of flags

Implementations shall provide the following non-computational operations that act upon multiple status flags collectively; these operations do not signal exceptions:

— *void* **lowerFlags**(*exceptionGroup*)

Lowers (clears) the flags corresponding to the exceptions specified in the *exceptionGroup* operand, which can represent any subset of the exceptions.

— *void* **raiseFlags**(*exceptionGroup*)

Raises (sets) the flags corresponding to the exceptions specified in the *exceptionGroup* operand, which can represent any subset of the exceptions.

— *boolean* **testFlags**(*exceptionGroup*)

Queries whether any of the flags corresponding to the exceptions specified in the *exceptionGroup* operand, which can represent any subset of the exceptions, are raised.

— *boolean* **testSavedFlags**(*flags, exceptionGroup*)

Queries whether any of the flags in the *flags* operand corresponding to the exceptions specified in the *exceptionGroup* operand, which can represent any subset of the exceptions, are raised.

— *void* **restoreFlags**(*flags*, *exceptionGroup*)

Restores the flags corresponding to the exceptions specified in the *exceptionGroup* operand, which can represent any subset of the exceptions, to their state represented in the *flags* operand.

— *flags* **saveAllFlags**(*void*)

Returns a representation of the state of all status flags.

The return value of the **saveAllFlags** operation is for use as the first operand to a **restoreFlags** or **testSavedFlags** operation in the same program; this standard does not require support for any other use.

## 5.8 Details of conversions from floating-point to integer formats

Implementations shall provide conversion operations from all supported arithmetic formats to all supported signed and unsigned integer formats. Integral values are converted exactly from floating-point formats to integer formats whenever the value is representable in both formats.

Conversion to integer shall round as specified in Clause 4; the rounding direction is indicated by the operation name.

When a NaN or infinite operand cannot be represented in the destination format and this cannot otherwise be indicated, the invalid operation exception shall be signaled. When a numeric operand would convert to an integer outside the range of the destination format, the invalid operation exception shall be signaled if this situation cannot otherwise be indicated.

When the value of the conversion operation's result differs from its operand value, yet is representable in the destination format, some conversion operations are specified below to signal the inexact exception and others to not signal the inexact exception.

A language standard that permits implicit conversions or expressions involving mixed types should require that these be implemented with the inexact-signaling conversion operations below.

The operations for conversion from floating-point to a specific signed or unsigned integer format without signaling the inexact exception are:

— *intFormatOf*-**convertToIntegerTiesToEven**(*source*)

**convertToIntegerTiesToEven**(*x*) rounds *x* to the nearest integral value, with halfway cases rounded to even.

— *intFormatOf*-**convertToIntegerTowardZero**(*source*)

**convertToIntegerTowardZero**(*x*) rounds *x* to an integral value toward zero.

— *intFormatOf*-**convertToIntegerTowardPositive**(*source*)

**convertToIntegerTowardPositive**(*x*) rounds *x* to an integral value toward positive infinity.

— *intFormatOf*-**convertToIntegerTowardNegative**(*source*)

**convertToIntegerTowardNegative**(*x*) rounds *x* to an integral value toward negative infinity.

— *intFormatOf*-**convertToIntegerTiesToAway**(*source*)

**convertToIntegerTiesToAway**(*x*) rounds *x* to the nearest integral value, with halfway cases rounded away from zero.

The operations for conversion from floating-point to a specific signed or unsigned integer format, signaling if inexact, are:

— *intFormatOf*-**convertToIntegerExactTiesToEven**(*source*)

**convertToIntegerExactTiesToEven**(*x*) rounds *x* to the nearest integral value, with halfway cases rounded to even.

— *intFormatOf*-**convertToIntegerExactTowardZero**(*source*)

**convertToIntegerExactTowardZero**(*x*) rounds *x* to an integral value toward zero.

— *intFormatOf*-**convertToIntegerExactTowardPositive**(*source*)

**convertToIntegerExactTowardPositive**(*x*) rounds *x* to an integral value toward positive infinity.

— *intFormatOf*-**convertToIntegerExactTowardNegative**(*source*)

**convertToIntegerExactTowardNegative**(*x*) rounds *x* to an integral value toward negative infinity.

— *intFormatOf*-**convertToIntegerExactTiesToAway**(*source*)

**convertToIntegerExactTiesToAway**(*x*) rounds *x* to the nearest integral value, with halfway cases rounded away from zero.

## 5.9 Details of operations to round a floating-point datum to integral value

Several operations round a floating-point number to an integral-valued floating-point number in the same format.

The rounding is analogous to that specified in Clause 4, but the rounding chooses only from among those floating-point numbers of integral values in the format. These operations convert zero operands to zero results of the same sign, and infinite operands to infinite results of the same sign.

For the following operations, the rounding direction is specified by the operation name and does not depend on a rounding-direction attribute. These operations shall not signal any exception except for signaling NaN input.

— *sourceFormat* **roundToIntegralTiesToEven**(*source*)

**roundToIntegralTiesToEven**(*x*) rounds *x* to the nearest integral value, with halfway cases rounding to even.

— *sourceFormat* **roundToIntegralTowardZero**(*source*)

**roundToIntegralTowardZero**(*x*) rounds *x* to an integral value toward zero.

— *sourceFormat* **roundToIntegralTowardPositive**(*source*)

**roundToIntegralTowardPositive**(*x*) rounds *x* to an integral value toward positive infinity.

— *sourceFormat* **roundToIntegralTowardNegative**(*source*)

**roundToIntegralTowardNegative**(*x*) rounds *x* to an integral value toward negative infinity.

— *sourceFormat* **roundToIntegralTiesToAway**(*source*)

**roundToIntegralTiesToAway**(*x*) rounds *x* to the nearest integral value, with halfway cases rounding away from zero.

For the following operation, the rounding direction is the applicable rounding-direction attribute. This operation signals the invalid operation exception for a signaling NaN operand, and for a numerical operand, signals the inexact exception if the result does not have the same numerical value as *x*.

— *sourceFormat* **roundToIntegralExact**(*source*)

**roundToIntegralExact**(*x*) rounds *x* to an integral value according to the applicable rounding-direction attribute.

## 5.10 Details of totalOrder predicate

For each supported arithmetic format, an implementation shall provide the following predicate that defines an ordering among all operands in a particular format:

— *boolean* **totalOrder**(*source, source*)

**totalOrder**(*x, y*) imposes a total ordering on canonical members of the format of *x* and *y*:

a) If $x<y$, **totalOrder**(*x, y*) is true.
b) If $x>y$, **totalOrder**(*x, y*) is false.
c) If $x=y$:
   1) **totalOrder**(−0, +0) is true.
   2) **totalOrder**(+0, −0) is false.
   3) If *x* and *y* represent the same floating-point datum:
      i) If *x* and *y* have negative sign,
      **totalOrder**(*x*, *y*) is true if and only if the exponent of $x \geq$ the exponent of *y*
      ii) otherwise,
      **totalOrder**(*x*, *y*) is true if and only if the exponent of $x \leq$ the exponent of *y*.
d) If *x* and *y* are unordered numerically because *x* or *y* is a NaN:
   1) **totalOrder**(−NaN, *y*) is true where −NaN represents a NaN with negative sign bit and *y* is a floating-point number.
   2) **totalOrder**(*x,* −NaN) is false where −NaN represents a NaN with negative sign bit and *x* is a floating-point number.
   3) **totalOrder**(*x*, +NaN) is true where +NaN represents a NaN with positive sign bit and *x* is a floating-point number.
   4) **totalOrder**(+NaN, *y*) is false where +NaN represents a NaN with positive sign bit and *y* is a floating-point number.
   5) If *x* and *y* are both NaNs, then **totalOrder** reflects a total ordering based on:
      i) negative sign orders below positive sign
      ii) signaling orders below quiet for +NaN, reverse for −NaN
      iii) otherwise, the order of NaNs is implementation-defined.

Neither signaling NaNs nor quiet NaNs signal an exception. For canonical *x* and *y*, **totalOrder**(*x, y*) and **totalOrder**(*y, x*) are both true if *x* and *y* are bitwise identical.

Unsigned NaNs, as may occur in non-interchange formats, should order like NaNs with positive sign bit.

NOTE— **totalOrder** does not impose a total ordering on all encodings in a format. In particular, it does not distinguish among different encodings of the same floating-point representation, as when one or both encodings are non-canonical.

## 5.11 Details of comparison predicates

For every supported arithmetic format, it shall be possible to compare one floating-point datum to another in that format (see 5.6.1). Additionally, it shall be possible to compare one floating-point datum to another in a different format as long as the operands' formats have the same radix.

Four mutually exclusive relations are possible: *less than*, *equal*, *greater than*, and *unordered*; *unordered* arises when at least one operand is a NaN. Every NaN shall compare *unordered* with everything, including itself. Comparisons shall ignore the sign of zero (so +0 = −0). Infinite operands of the same sign shall compare *equal*.

Language standards shall define the comparison predicates in Tables 5.1 and 5.2. These predicates deliver a true-false response and are defined in pairs. Each predicate is true if and only if any of its indicated relations is true. Each member of a pair is the logical negation of the other. Applying a prefix such as NOT to negate a predicate reverses the true-false sense of its associated entries, but does not change whether *unordered* relations signal an invalid operation exception.

Invalid operation is the only exception that a comparison predicate can signal. All predicates signal the invalid operation exception on signaling NaN operands. The predicates named Quiet shall not signal any exception, unless an operand is a signaling NaN. The predicates named Signaling shall signal the invalid operation exception on quiet NaN operands.

For the common mathematical symbols = ≠ > ≥ < ≤, many language standards define notations that do not explicitly indicate signaling or quiet. In Table 5.1, those notations are represented by their common mathematical symbols. Language standards should map their notations for the symbols = and ≠ to the Quiet predicates in Table 5.1, and their notations for the symbols > ≥ < ≤ to the Signaling predicates in Table 5.1.

**Table 5.1—Required predicates and negations, recommended for common math symbols**

| math symbol | **predicate** *true relations* | math symbol | **negation** *true relations* |
|---|---|---|---|
| = | **compareQuietEqual** *equal* | ≠, NOT = | **compareQuietNotEqual** *less than, greater than, unordered* |
| > | **compareSignalingGreater** *greater than* | NOT > | **compareSignalingNotGreater** *less than, equal, unordered* |
| ≥ | **compareSignalingGreaterEqual** *equal, greater than* | NOT ≥ | **compareSignalingLessUnordered** *less than, unordered* |
| < | **compareSignalingLess** *less than* | NOT < | **compareSignalingNotLess** *equal, greater than, unordered* |
| ≤ | **compareSignalingLessEqual** *less than, equal* | NOT ≤ | **compareSignalingGreaterUnordered** *greater than, unordered* |

NOTE—When NaNs are present, operands have the *unordered* relation, so trichotomy does not apply. For example, NOT (X < Y) is not logically equivalent to (X ≥ Y). The Signaling predicates in Table 5.1 signal the invalid operation exception on quiet NaN operands to warn of potential incorrect behavior of programs written assuming trichotomy.

**Table 5.2—Other required predicates and negations**

| **predicate**<br>*true relations* | **negation**<br>*true relations* |
|---|---|
| **compareSignalingEqual**<br>*equal* | **compareSignalingNotEqual**<br>*less than, greater than, unordered* |
| **compareQuietGreater**<br>*greater than* | **compareQuietNotGreater**<br>*less than, equal, unordered* |
| **compareQuietGreaterEqual**<br>*equal, greater than* | **compareQuietLessUnordered**<br>*less than, unordered* |
| **compareQuietLess**<br>*less than* | **compareQuietNotLess**<br>*equal, greater than, unordered* |
| **compareQuietLessEqual**<br>*less than, equal* | **compareQuietGreaterUnordered**<br>*greater than, unordered* |
| **compareQuietUnordered**<br>*unordered* | **compareQuietOrdered**<br>*less than, equal, greater than* |

## 5.12 Details of conversion between floating-point data and external character sequences

This clause specifies conversions between supported formats and external character sequences. Observe that conversions between supported formats of different radices are correctly rounded and set exceptions correctly as described in 5.4.2, subject to limits stated in 5.12.2 below.

Implementations shall provide conversions between each supported binary format and external decimal character sequences such that, under roundTiesToEven, conversion from the supported format to external decimal character sequence and back recovers the original floating-point representation, except that a signaling NaN might be converted to a quiet NaN. See 5.12.1 and 5.12.2 for details.

Implementations shall provide exact conversions from each supported decimal format to external decimal character sequences, and shall provide conversions back that recover the original floating-point representation, except that a signaling NaN might be converted to a quiet NaN. See 5.12.1 and 5.12.2 for details.

Implementations shall provide exact conversions from each supported binary format to external character sequences representing numbers with hexadecimal digits for the significand, and shall provide conversions back that recover the original floating-point representation, except that a signaling NaN might be converted to a quiet NaN. See 5.12.1 and 5.12.3 for details.

This clause primarily discusses conversions during program execution; there is one special consideration applicable to program translation separate from program execution: translation-time conversion of constants in program text from external character sequences to supported formats, in the absence of other specification in the program text, shall use this standard's default rounding direction and language-defined exception handling. An implementation might also provide means to permit constants to be translated at execution time with the attributes in effect at execution time and exceptions generated at execution time.

Issues of character codes (ASCII, Unicode, etc.) are not defined by this standard.

### 5.12.1 External character sequences representing zeros, infinities, and NaNs

The conversions (described in 5.4.2) from supported formats to external character sequences and back that recover the original floating-point representation, shall recover zeros, infinities, and quiet NaNs, as well as non-zero finite numbers. In particular, signs of zeros and infinities are preserved.

Conversion of an infinity in a supported format to an external character sequence shall produce a language-defined one of "inf" or "infinity" or a sequence that is equivalent except for case (*e.g.*, "Infinity" or "INF"), with a preceding minus sign if the input is negative. Whether the conversion produces a preceding plus sign if the input is positive is language-defined.

Conversion of external character sequences "inf" and "infinity" (regardless of case) with an optional preceding sign, to a supported floating-point format shall produce an infinity (with the same sign as the input).

Conversion of a quiet NaN in a supported format to an external character sequence shall produce a language-defined one of "nan" or a sequence that is equivalent except for case (*e.g.*, "NaN"), with an optional preceding sign. (This standard does not interpret the sign of a NaN.)

Conversion of a signaling NaN in a supported format to an external character sequence should produce a language-defined one of "snan" or "nan" or a sequence that is equivalent except for case, with an optional preceding sign. If the conversion of a signaling NaN produces "nan" or a sequence that is equivalent except for case, with an optional preceding sign, then the invalid operation exception should be signaled.

Conversion of external character sequences "nan" (regardless of case) with an optional preceding sign, to a supported floating-point format shall produce a quiet NaN.

Conversion of an external character sequence "snan" (regardless of case) with an optional preceding sign, to a supported format should either produce a signaling NaN or else produce a quiet NaN and signal the invalid operation exception.

Language standards should provide an optional conversion of NaNs in a supported format to external character sequences which appends to the basic NaN character sequences a suffix that can represent the payload (see 6.2). The form and interpretation of the payload suffix is language-defined. The language standard shall require that any such optional output sequences be accepted as input in conversion of external character sequences to supported formats.

### 5.12.2 External decimal character sequences representing finite numbers

An implementation shall provide operations that convert from all supported floating-point formats to external decimal character sequences (see 5.4.2). For finite numbers, these operations can be thought of as parameterized by the source format, the number of significant digits in the result (if specified), and whether the quantum is preserved (for decimal formats). Observe that specifying the number of significant digits and specifying quantum preservation are mutually incompatible. The means of specifying the number of significant digits and of specifying quantum preservation are language-defined and are typically embodied in the *conversionSpecification* of 5.4.2.

An implementation shall also provide operations that convert external decimal character sequences to all supported formats. These operations can be thought of as parameterized by the result format.

Within the limits stated in this clause, conversions in both directions shall preserve the value of a number unless rounding is necessary and shall preserve its sign. If rounding is necessary, they shall use correct rounding and shall correctly signal the inexact and other exceptions.

All conversions from external character sequences to supported decimal formats shall preserve the quantum (see 5.4.2) unless rounding is necessary. At least one conversion from each supported decimal format shall preserve the quantum as well as the value and sign.

If a conversion to an external character sequence requires an exponent but the exponent is not of sufficient width to avoid overflow or underflow (see 7.4 and 7.5), the overflow or underflow should be indicated to the user by appropriate language-defined character sequences.

For the purposes of discussing the limits on correctly rounded conversion, define the following quantities:

- for binary16, $Pmin(\text{binary16}) = 5$
- for binary32, $Pmin(\text{binary32}) = 9$
- for binary64, $Pmin(\text{binary64}) = 17$
- for binary128, $Pmin(\text{binary128}) = 36$
- for all other binary formats *bf*, $Pmin(bf) = 1 + \text{ceiling}(p \times \log_{10}(2))$, where $p$ is the number of significant bits in *bf*
- $M = \max(Pmin(bf))$ for all supported binary formats *bf*
- for decimal32, $Pmin(\text{decimal32}) = 7$
- for decimal64, $Pmin(\text{decimal64}) = 16$
- for decimal 128, $Pmin(\text{decimal 128}) = 34$
- for all other decimal formats *df*, $Pmin(df)$ is the number of significant digits in *df*.

Conversions to and from supported decimal formats shall be correctly rounded regardless of how many digits are requested or given.

There might be an implementation-defined limit on the number of significant digits that can be converted with correct rounding to and from supported binary formats. That limit, H, shall be such that $H \geq M + 3$ and it should be that H is unbounded.

For all supported binary formats the conversion operations shall support correctly rounded conversions to or from external character sequences for all significant digit counts from 1 through H (that is, for all expressible counts if H is unbounded).

Conversions from supported binary formats to external character sequences for which more than H significant digits are specified shall pad with trailing zeros.

Conversion from a character sequence of more than H significant digits or larger in exponent range than the destination binary format first shall be correctly rounded to H digits according to the applicable rounding direction and shall signal exceptions as though narrowing from a wider format and then the resulting character sequence of H digits shall be converted with correct rounding according to the applicable rounding direction.

NOTE 1—As a consequence of the foregoing, the following are true:

- Conversions to or from decimal formats are correctly rounded.
- For binary formats, all conversions of H significant digits or fewer round correctly according to the applicable rounding direction; conversions of greater than H significant digits might incur additional rounding of the order of $10^{(M-H)} < 10^{-3}$ units in the last place.
- Intervals are respected, in the sense that directed-rounding constraints are honored even when more than H significant digits are given: the directed rounding error has the correct sign in all cases, and never exceeds 1 + 1/1000 units in the last place in magnitude.
- Conversions are monotonic; increasing the value of a supported floating-point number does not decrease its value after conversion to an external character sequence, and increasing the value of an external character sequence does not decrease its value after conversion to a supported floating-point number.
- Conversions from a supported binary format *bf* to an external character sequence and back again results in a copy of the original number so long as there are at least $Pmin(bf)$ significant digits specified and the rounding-direction attributes in effect during the two conversions are round to nearest rounding-direction attributes.
- Conversions from a supported decimal format *df* to an external character sequence and back again results in a canonical copy of the original number so long as the conversion to the external character sequence is one that preserves the quantum.
- Conversions from a supported decimal format *df* to an external character sequence and back again recovers the value (but not necessarily the quantum) of the original number so long as there are at least $Pmin(df)$ significant digits specified.

— All implementations exchange equivalent decimal sequences: two decimal character sequences are equivalent if they represent the same value (and quantum, for decimal formats); if two implementations support a given format they convert any floating-point representation in that format to equivalent decimal character sequences when the same number of digits is specified and (for binary formats) the specified number of digits is no greater than H (for both implementations), or (for decimal formats) when the quantum-preserving conversion is specified.

— Similarly, any two implementations convert equivalent decimal sequences to the same floating-point number (with the same quantum, for decimal formats) when the number of significant digits and the result format are supported on both implementations.

NOTE 2—H should be as large as practical, noting that "practical" might well include "unbounded" on many systems because any H at least as large as the number of digits required for the longest exact decimal representation is effectively as good as unbounded. The length of the longest exact decimal representation is less than twelve thousand digits for binary128.

### 5.12.3 External hexadecimal-significand character sequences representing finite numbers

Language standards should provide conversions between all supported binary formats and external hexadecimal-significand character sequences. External hexadecimal-significand character sequences for finite numbers shall be described by the following grammar, which defines a *hexSequence*:

```
sign            [+ −]
digit           [0123456789]
hexDigit        [0123456789abcdefABCDEF]
hexExpIndicator [Pp]
hexIndicator    "0" [Xx]
hexSignificand  ( {hexDigit}* "." {hexDigit}+ | {hexDigit}+ "." | {hexDigit}+ )
decExponent     {hexExpIndicator} {sign}? {digit}+
hexSequence     {sign}? {hexIndicator} {hexSignificand} {decExponent}
```

where each line is a name followed by a rule in which '[...]' selects one of the terminal characters listed between the brackets, '{...}' refers to an earlier named rule, '(...|...|...)' indicates a choice of one of three alternatives, straight double quotes enclose a terminal character, '?' indicates that there shall be either no instance or one instance of the preceding item, '*' indicates that there shall be zero or more instances of the preceding item, and '+' indicates that there shall be one or more instances of the preceding item.

The *hexSignificand* is interpreted as a hexadecimal constant in which each *hexDigit* represents a value in the range 0 through 15 with the letters 'a' through 'f' representing 10 through 15, regardless of case. Within the *hexSignificand*, the first (leftmost) character is the most significant. If present, the period defines the start of a hexadecimal fractional part; if the period is to the right of all hexadecimal digits the *hexSignificand* is an integer. The *decExponent* is interpreted as an optionally-signed integer expressed in decimal following the *hexExpIndicator*, again with the most significant digit first.

The value of a *hexSequence* is the value of the *hexSignificand* multiplied by two raised to the power of the value of the *decExponent*, negated if there is a leading '−' sign. The *hexIndicator* and the *hexExpIndicator* have no effect on the value.

When converting to hexadecimal-significand character sequences in the absence of an explicit precision specification, enough hexadecimal characters shall be used to represent the binary floating-point number exactly. Conversions to hexadecimal-significand character sequences with an explicit precision specification, and conversions from hexadecimal-significand character sequences to supported binary formats, are correctly rounded according to the applicable binary rounding-direction attribute, and signal all exceptions appropriately.

NOTE—The external hexadecimal-significand character sequences described here follow those specified for finite numbers in ISO/IEC 9899:2018(E) Programming languages—C (C17), in:

6.4.4.2 floating constants
7.21.6.1 fprintf (conversion specifiers 'a' and 'A')
7.21.6.2 fscanf (conversion specifier 'a')
7.22.1.3 strtod.

# 6. Infinity, NaNs, and sign bit

## 6.1 Infinity arithmetic

The behavior of infinity in floating-point arithmetic is derived from the limiting cases of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists. Infinities shall be interpreted in the affine sense, that is: $-\infty <$ {every finite number} $< +\infty$.

Operations on infinite operands are usually exact and therefore signal no exceptions, including, among others,

- **addition**($\infty$, $x$), **addition**($x$, $\infty$), **subtraction**($\infty$, $x$), or **subtraction**($x$, $\infty$), for finite $x$
- **multiplication**($\infty$, $x$) or **multiplication**($x$, $\infty$) for finite or infinite $x \neq 0$
- **division**($\infty$, $x$) or **division**($x$, $\infty$) for finite $x$
- **squareRoot**($+\infty$)
- **remainder**($x$, $\infty$) for finite normal $x$
- conversion of an infinity into the same infinity in another format.

The exceptions that do pertain to infinities are signaled only when

- $\infty$ is an invalid operand (see 7.2)
- $\infty$ is created from finite operands by overflow (see 7.4) or division by zero (see 7.3)
- **remainder**(subnormal, $\infty$) signals underflow.

## 6.2 Operations with NaNs

Two different kinds of NaN, signaling and quiet, shall be supported in all floating-point operations. Signaling NaNs afford representations for uninitialized variables and arithmetic-like enhancements (such as complex-affine infinities or extremely wide range) that are not in the scope of this standard. Quiet NaNs should, by means left to the implementer's discretion, afford retrospective diagnostic information inherited from invalid or unavailable data and results. To facilitate propagation of diagnostic information contained in NaNs, as much of that information as possible should be preserved in NaN results of operations.

Under default exception handling, any operation signaling an invalid operation exception and for which a floating-point result is to be delivered, except as stated otherwise, shall deliver a quiet NaN. For non-default treatment, see 8.

Signaling NaNs shall be reserved operands that signal the invalid operation exception (see 7.2) for every general-computational and signaling-computational operation except for the conversions described in 5.12.

Every general-computational and quiet-computational operation involving one or more input NaNs, none of them signaling, shall signal no exception, except **fusedMultiplyAdd** might signal the invalid operation exception (see 7.2). For an operation with quiet NaN inputs, except as stated otherwise, if a floating-point result is to be delivered the result shall be a canonical quiet NaN. Recognize that format conversions, including conversions between supported formats and external representations as character sequences, might be unable to deliver the same NaN. Quiet NaNs signal exceptions on some operations that do not deliver a floating-point result; these operations, namely comparison and conversion to a format that has no NaNs, are discussed in 5.6, 5.8, and 7.2.

### 6.2.1 NaN encodings in binary interchange formats

All binary NaN bit strings have the sign bit $S$ set to 0 or 1 and all the bits of the biased exponent field $E$ set to 1 (see 3.4). A quiet NaN bit string should be encoded with the first bit ($d_1$) of the trailing significand field $T$ being 1. A signaling NaN bit string should be encoded with the first bit of the trailing significand field being 0. If the first bit of the trailing significand field is 0, some other bit of the trailing significand field must be non-zero to distinguish the NaN from infinity. In the preferred encoding just described, a signaling NaN shall be quieted by setting $d_1$ to 1, leaving the remaining bits of $T$ unchanged. Bits $d_2$ $d_3$ …

$d_{p-1}$ of the trailing significand field contain the encoding of the payload, which might be diagnostic information (see 6.2).

### 6.2.2 NaN encodings in decimal interchange formats

A decimal signaling NaN shall be quieted by clearing $G_5$ and leaving the values of the digits $d_1$ through $d_{p-1}$ of the trailing significand field unchanged (see 3.5).

For decimal formats, the payload is the trailing significand field, as defined in 3.5.

### 6.2.3 NaN propagation

An operation that propagates a NaN operand to its result and has a single NaN as an input should produce a NaN with the payload of the input NaN if representable in the destination format.

If two or more inputs are NaN, then the payload of the resulting NaN should be identical to the payload of one of the input NaNs if representable in the destination format. This standard does not specify which of the input NaNs will provide the payload.

Conversion of a quiet NaN from a narrower format to a wider format in the same radix, and then back to the same narrower format, should not change the quiet NaN payload in any way except to make it canonical.

Conversion of a quiet NaN to a floating-point format of the same or a different radix that does not allow the payload to be preserved shall return a quiet NaN that should provide some language-defined diagnostic information.

There should be means to read and write payloads from and to external character sequences (see 5.12.1).

Except for the operations specified otherwise in 5.5, a NaN result shall be canonical, even if that NaN result were derived from a non-canonical NaN operand.

## 6.3 The sign bit

When either an input or result is a NaN, this standard does not interpret the sign of a NaN. However, operations on bit strings—**copy**, **negate**, **abs**, **copySign**—specify the sign bit of a NaN result, sometimes based upon the sign bit of a NaN operand. The logical predicates **totalOrder** and **isSignMinus** are also affected by the sign bit of a NaN operand. For all other operations, this standard does not specify the sign bit of a NaN result, even when there is only one input NaN, or when the NaN is produced from an invalid operation.

When neither the inputs nor result are NaN, the sign of a product or quotient is the exclusive OR of the operands' signs; the sign of a sum, or of a difference $x - y$ regarded as a sum $x + (-y)$, differs from at most one of the addends' signs; and the sign of the result of conversions, the **quantize** operation, the roundToIntegral operations, and the **roundToIntegralExact** (see 5.3.1) is the sign of the first or only operand. These rules shall apply even when operands or results are zero or infinite.

When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be +0 under all rounding-direction attributes except roundTowardNegative; under that attribute, the sign of an exact zero sum (or difference) shall be −0. However, under all rounding-direction attributes, when $x$ is zero, $x + x$ and $x - (-x)$ have the sign of $x$.

When $(a \times b) + c$ is exactly zero, the sign of **fusedMultiplyAdd**($a$, $b$, $c$) shall be determined by the rules above for a sum of operands. When the exact result of $(a \times b) + c$ is non-zero yet the result of **fusedMultiplyAdd** is zero because of rounding, the zero result takes the sign of the exact result.

Except that **squareRoot**(−0) shall be −0, every numeric **squareRoot** result shall have a positive sign.

# 7. Exceptions and default exception handling

## 7.1 Overview: exceptions and flags

This clause specifies five kinds of exceptions that shall be signaled when they arise; the signal invokes default or alternate handling for the signaled exception. For each kind of exception the implementation shall provide a corresponding status flag.

This clause also specifies default non-stop exception handling for exception signals, which is to deliver a default result, continue execution, and raise the corresponding status flag (except in the case of exact underflow, see 7.5). Clause 8 specifies alternate exception handling attributes for those signals; a language standard might specify that some of those attributes be implemented and then define means for users to enable them. Default or alternate exception handling for one exception might also signal other exceptions (see overflow and underflow, 7.4 and 7.5). Therefore, a status flag might be raised by default, by alternate exception handling, or by explicit user action (see 5.7.4).

With default exception handling, a raised status flag usually indicates that the corresponding exception was signaled and handled by default. Exceptions are handled without raising status flags only in the case of exact underflow and status flags are raised without an exception being signaled only at the user's request. Status flags shall be lowered only at the user's request. The user shall be able to test and to alter the status flags individually or collectively, and shall further be able to save and restore all at one time (see 5.7.4).

A program that does not inherit status flags from another source begins execution with all status flags lowered. Language standards should specify defaults in the absence of any explicit user specification, governing:

- Whether any particular flag exists (in the sense of being testable by non-programmatic means such as debuggers) outside of scopes in which a program explicitly sets or tests that flag.
- When a flag has scope greater than within an invoked function, whether and when an asynchronous event, such as raising or lowering the flag in another thread or signal handler, affects the flag tested within that invoked function; this includes events arising from explicit asynchronicity in the program and also events arising from asynchronicity introduced by language or implementation.
- When a flag has scope greater than within an invoked function, whether that flag's state can be determined by non-programmatic means (such as a debugger) within that invoked function.
- Whether flags raised in invoked functions raise flags in invoking functions.
- Whether flags raised in invoking functions raise flags in invoked functions.
- Whether to allow, and if so the means, to specify that flags shall be persistent in the absence of any explicit program statement otherwise:
  - The flags standing at the beginning of execution of a particular function are inherited from an outer environment, typically an invoking function.
  - On return from or termination of an invoked function, the flags standing in an invoking function are the flags that were standing in the invoked function at the time of return or termination.

An invocation of any operation required by this standard signals at most one exception; additional exceptions might be signaled by default exception handling or by alternate exception handling for the first exception. Default exception handling for overflow (see 7.4) signals the inexact exception. Default exception handling for underflow (see 7.5) signals the inexact exception if the default result is inexact. Default exception handling for invalid (see 7.2) due to a signaling NaN operand may signal another invalid operation exception for comparisons by way of unordered-signaling predicates.

An invocation of the **restoreFlags** or **raiseFlags** operation (see 5.7.4) might raise any combination of status flags. An invocation of any other operation required by this standard, when all exceptions are handled by default, might raise at most two status flags, overflow with inexact (see 7.4) or underflow with inexact (see 7.5).

For the computational operations defined in this standard, exceptions are defined below to be signaled if and only if certain conditions arise. Thus, these operations shall be computed in a way that avoids user-

observable signals of the exceptions for other conditions, even if the operation is implemented in software using other exception-signaling operations. Operations not specified by this standard, such as complex arithmetic or certain transcendental functions, should signal exceptions according to the definitions below for operations defined in this standard, but that might not always be economical. The signaling of exceptions for operations not specified in this standard is language-defined.

NOTE—Redundant signals of an exception by the implementation of an operation are not detectable by the user under default exception handling. Such redundant signals might be detectable by the user under the recordException attribute for (recommended) alternate exception handling (see 8.2), and, in the case of unordered-signaling predicates with a signaling NaN operand, under other alternate exception handling attributes if sub-exceptions (see 8.1) are supported.

## 7.2 Invalid operation

The invalid operation exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed.

For operations producing results in floating-point format, the default result of an operation that signals the invalid operation exception shall be a quiet NaN that should provide some diagnostic information (see 6.2). These operations are:

a) any general-computational operation on a signaling NaN (see 6.2), except for some conversions (see 5.12)
b) **multiplication**: **multiplication**(0, ∞) or **multiplication**(∞, 0)
c) **fusedMultiplyAdd**: **fusedMultiplyAdd**(0, ∞, *c*) or **fusedMultiplyAdd**(∞, 0, *c*) unless *c* is a quiet NaN; if *c* is a quiet NaN then it is implementation defined whether the invalid operation exception is signaled
d) **addition** or **subtraction** or **fusedMultiplyAdd**: magnitude subtraction of infinities, such as: **addition**(+∞, −∞)
e) **division**: **division**(0, 0) or **division**(∞, ∞)
f) **remainder**: **remainder**(*x, y*), when *y* is zero or *x* is infinite and neither is a NaN
g) **squareRoot** if the operand is less than zero
h) **quantize** when the result does not fit in the destination format or when one operand is finite and the other is infinite

For operations producing no result in floating-point format, the operations that signal the invalid operation exception are:

i) any signaling-computational operation on a signaling NaN (see 6.2); then, under default exception handling, the operation is evaluated with quiet NaNs in place of the signaling NaN operands to determine the result, which for unordered-signaling comparisons might signal another invalid operation exception
j) conversion of a floating-point number to an integer format, when the source is a NaN, infinity, or a value that would convert to an integer outside the range of the result format under the applicable rounding attribute
k) comparison by way of unordered-signaling predicates listed in Table 5.2, when the operands are *unordered*
l) **logB**(NaN), **logB**(∞), or **logB**(0), when *logBFormat* is an integer format (see 5.3.3).

## 7.3 Division by zero

The divideByZero exception shall be signaled if and only if an exact infinite result is defined for an operation on finite operands. The default result of divideByZero shall be an ∞ correctly signed according to the operation:

— For **division**, when the divisor is zero and the dividend is a finite non-zero number, the sign of the infinity is the exclusive OR of the operands' signs (see 6.3).
— For **logB**(0) when *logBFormat* is a floating-point format, the sign of the infinity is minus (−∞).

## 7.4 Overflow

The overflow exception shall be signaled if and only if the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (see Clause 4) were the exponent range unbounded. The default result shall be determined by the rounding-direction attribute and the sign of the intermediate result as follows:

a) roundTiesToEven and roundTiesToAway carry all overflows to ∞ with the sign of the intermediate result.
b) roundTowardZero carries all overflows to the format's largest finite number with the sign of the intermediate result.
c) roundTowardNegative carries positive overflows to the format's largest finite number, and carries negative overflows to −∞.
d) roundTowardPositive carries negative overflows to the format's most negative finite number, and carries positive overflows to +∞.

In addition, under default exception handling for overflow, the overflow flag shall be raised and the inexact exception shall be signaled.

## 7.5 Underflow

The underflow exception shall be signaled when a tiny non-zero result is detected. For binary formats, this shall be either:

a) *after rounding*—when a non-zero result computed as though the exponent range were unbounded would lie strictly between $\pm b^{emin}$, or
b) *before rounding*—when a non-zero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm b^{emin}$.

The implementer shall choose how tininess is detected, but shall detect tininess in the same way for all operations in radix two, including conversion operations under a binary rounding attribute.

For decimal formats, tininess is detected before rounding—when a non-zero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm b^{emin}$.

The default exception handling for underflow shall always deliver a rounded result. The method for detecting tininess does not affect the rounded result delivered, which might be zero, subnormal, or $\pm b^{emin}$.

In addition, under default exception handling for underflow, if the rounded result is inexact—that is, it differs from what would have been computed were both exponent range and precision unbounded—the underflow flag shall be raised and the inexact (see 7.6) exception shall be signaled. If the rounded result is exact, no flag is raised and no inexact exception is signaled. This is the only case in this standard of an exception signal receiving default handling that does not raise the corresponding flag. Such an underflow signal has no observable effect under default handling.

## 7.6 Inexact

Except as specified otherwise (e.g., 5.8 and 5.9), an operation delivering a numerical result that signals no other exception shall signal inexact if its rounded result differs from what would have been computed were both exponent range and precision unbounded. The rounded result shall be delivered to the destination.

NOTE—Default exception handling for overflow raises the overflow flag and signals inexact. When the rounded result is not an exact subnormal, default exception handling for underflow raises the underflow flag and signals inexact. When all of these exceptions are handled by default, the inexact flag is always raised when either the overflow or underflow flag is raised.

# 8. Alternate exception handling attributes

## 8.1 Overview

Language standards should define, and require implementations to provide, means for the user to associate alternate exception handling attributes with blocks (see 4.1). Alternate exception handlers specify lists of exceptions and actions to be taken for each listed exception if it is signaled. Language standards should define exception lists containing any subset of the exceptions listed in Clause 7: invalid operation, divideByZero, overflow, underflow, or inexact. Language standards should also define exception lists containing:

— **allExceptions**: all five exceptions listed in Clause 7, or

— any subset of sub-exceptions — sub-cases of the exceptions in Clause 7 (*e.g.*, the sub-cases of the invalid operation exception in 7.2); the action for a listed sub-exception is taken if and only if the operation is a specified case for the sub-exception; the sub-exception names are language-defined.

Language standards should define all the alternate exception handling attributes of this clause. In particular, language standards should define at least one delayed alternate exception handling attribute (see 8.3) for each of the five exceptions listed in Clause 7. The syntax and scope for such specifications of attribute values are language-defined.

Changing an exception handling attribute does not signal any exception.

## 8.2 Resuming alternate exception handling attributes

Associating a resuming alternate exception handling attribute with a block means: handling the implied exceptions according to the resuming attribute specified, and resuming execution of the associated block. Implementations should support these resuming attributes:

— **default** (raise flag)

Provide the default exception handling (see Clause 7) in the associated block despite alternate exception handling that might be in effect in wider scope.

— **raiseNoFlag**

Provide the default exception handling (see Clause 7) without raising the corresponding status flag.

— **mayRaiseFlag**

Provide the default exception handling (see Clause 7), except languages define whether a flag is raised. Languages may defer to implementations for performance.

— **recordException**

Provide the default exception handling (see Clause 7) and record the corresponding exception whenever Clause 7 specifies raising a flag. Recording an exception means storing a description of the exception, including language-defined details which might include the current operation and operands, and the location of the exception. Language standards define operations to convert exception descriptions to and from character sequences, and to inspect, save, and restore exception descriptions.

— **substitute(*x*)**

Specifiable for any exception: replace the default result of such an exceptional operation with a variable or expression *x*. The timing and scope in which *x* is evaluated is language-defined.

— **substituteXor(*x*)**

Specifiable for any exception arising from multiplication or division operations: like substitute(*x*), but replace the default result of such an exceptional operation with |x| and, if |x| is not a NaN, obtaining the sign bit from the XOR of the signs of the operands.

— **abruptUnderflow**

When underflow is signaled because a tiny non-zero result is detected, replace the default result with a zero of the same sign or a minimum normal rounded result of the same sign, raise the underflow flag, and signal the inexact exception. When roundTiesToEven, roundTiesToAway, or

the roundTowardZero attribute is applicable, the rounded result magnitude shall be zero. When the roundTowardPositive attribute is applicable, the rounded result magnitude shall be the minimum normal magnitude for positive tiny results, and zero for negative tiny results. When the roundTowardNegative attribute is applicable, the rounded result magnitude shall be the minimum normal magnitude for negative tiny results, and zero for positive tiny results. This attribute has no effect on the interpretation of subnormal operands.

For the augmented arithmetic operations (9.5), which return two results ($x$ operation $y$ and its error, where operation is +, −, or ×), if underflow is signaled because $x$ operation $y$ rounded using roundTiesTowardZero would signal underflow, both results are zero with the sign of $x$ operation $y$. If underflow is signaled because the error ($x$ operation $y$ − roundTiesTowardZero($x$ operation $y$)) is non-zero and lies strictly between $\pm b^{emin}$, the default error result is replaced with a zero with the sign of ($x$ operation $y$ − roundTiesTowardZero($x$ operation $y$)). These cases raise the underflow flag and signal the inexact exception.

## 8.3 Immediate and delayed alternate exception handling attributes

Associating alternate exception handling with a block means: handling the indicated exception(s) according to the attribute specified. If the indicated exception is signaled then, depending on the exception and the exception handling attribute, the execution of the associated block might be abandoned immediately or might continue with default handling. In the latter case the exception handling is delayed and takes place when the associated block terminates normally. Delayed exception handling is fully deterministic, while immediate exception handling licenses but does not require an implementation to trade determinism for performance, because intermediate results being computed within the associated block might not be deterministic.

Language standards should define, and require implementations to provide, these attributes:

- Immediate alternate exception handler block associated with a block: if the indicated exception is signaled, abandon execution of the associated block as soon as possible and execute the handler block, then continue execution where execution would have continued after normal termination of the associated block, according to the semantics of the language.
- Delayed alternate exception handler block associated with a block: if the indicated exception is signaled, handle it by default until the associated block terminates normally, then execute the handler block, then continue execution where execution would have continued after normal termination of the associated block, according to the semantics of the language.
- Immediate transfer associated with a block: if the indicated exception is signaled, transfer control as soon as possible; no return is possible.
- Delayed transfer associated with a block: if the indicated exception is signaled, handle it by default until the associated block terminates normally, then transfer control; no return is possible.

Immediate alternate exception handling for underflow shall be invoked when underflow is signaled, whether the default result would be exact or inexact. Delayed alternate exception handling for underflow shall be invoked only for underflow signals corresponding to inexact default results for which the underflow flag would be raised.

NOTE 1—Delayed alternate exception handling for an exception listed in Clause 7 (but not sub-exceptions) can be implemented by testing status flags. However implemented, the status flag corresponding to the indicated exception should be saved prior to the beginning of the associated block and then lowered. At the end of the associated block, the current status flag should be saved, and the previously saved status flag should be restored. The recently saved status flag should then be tested to determine whether to execute the handler block or transfer control.

NOTE 2—Immediate alternate exception handling for an exception can be implemented by traps or, for exceptions listed in Clause 7 other than underflow, by testing status flags after each operation or at the end of the associated block. Thus for exceptions listed in Clause 7 other than underflow, immediate exception handling can be implemented with the same mechanism as delayed exception handling, if no better implementation mechanism is available. No matter how implemented, if the indicated exception is not signaled in the associated block, then the corresponding status flag should not be changed. If the indicated exception is signaled in the associated block, causing execution of the handler block or transfer of control, then the state of the corresponding status flag might not be deterministic.

NOTE 3—A transfer is a language-specific idiom for non-resumable control transfer. Language standards might offer several transfer idioms such as:

- **break:** Abandon the associated block and continue execution where execution would continue after normal termination of the associated block, according to the semantics of the language.
- **throw exceptionName:** Causes an exceptionName not to be handled locally, but rather signaled to the next handling in scope, perhaps the function that invoked the current subprogram, according to the semantics of that language. The invoker might handle exceptionName by default or by alternate handling such as signaling exceptionName to the next higher invoking subprograms.
- **goto label:** Jump; the label might be local or global according to the semantics of the language.

# 9. Recommended operations

Clause 5 completely specifies the operations required for all supported arithmetic formats.

This clause specifies additional operations that are recommended. In a specific programming environment, these operations might be represented in operator notation or in function notation. The function names used in a specific programming environment might differ from the names of the corresponding mathematical functions or from the names of this standard's corresponding operations.

## 9.1 Conforming language- and implementation-defined operations

For one or more formats, language standards and implementations might define one or more floating-point operations, not otherwise defined in this document, that conform to this standard by meeting all the requirements of this clause. In particular, language standards should define, to be implemented according to this clause, as many of the operations of 9.2 through 9.7 as are appropriate to the language.

## 9.2 Additional mathematical operations

Language standards should define, to be implemented according to this subclause, as many of the operations in Table 9.1 as is appropriate to the language. As with other operations of this standard, the names of the operations in Table 9.1 do not necessarily correspond to the names that any particular programming language would use.

In this subclause the domain of an operation is that subset of the affinely extended reals for which the corresponding mathematical function is well defined.

A conforming operation shall return results correctly rounded for the applicable rounding direction for all operands in its domain.

Operation results shall be canonical.

Except as specified here, operations signal all appropriate exceptions according to Clause 7. An operation that returns a floating-point result shall return a quiet NaN as a result if there is a signaling NaN among the operation's operands. An operation that returns a floating-point result shall return a quiet NaN as a result if there is a quiet NaN among the operation's operands, except in the cases stated otherwise in this subclause.

— invalid operation: For all operations, signaling NaN operands shall signal the invalid operation exception.

  Outside its domain an operation shall return a quiet NaN and signal the invalid operation exception.

— divideByZero: An operation that has a simple pole for some finite floating-point operand shall signal the divideByZero exception and return an infinity by default.

— inexact: Operations should signal the inexact exception if the result is inexact. Operations should not signal the inexact exception if the result is exact.

Other exceptions are shown in Table 9.1.

**Table 9.1—Additional mathematical operations**

| Operation | Function | Domain | Other exceptions; see also 9.2.1 |
|---|---|---|---|
| **exp**<br>**expm1**<br>**exp2**<br>**exp2m1**<br>**exp10**<br>**exp10m1** | $e^x$<br>$e^x - 1$<br>$2^x$<br>$2^x - 1$<br>$10^x$<br>$10^x - 1$ | $[-\infty, +\infty]$ | overflow; underflow |
| **log**<br>**log2**<br>**log10** | $\log_e(x)$<br>$\log_2(x)$<br>$\log_{10}(x)$ | $[0, +\infty]$ | $x = 0$: divideByZero;<br>$x < 0$: invalid operation |
| **logp1**<br>**log2p1**<br>**log10p1** | $\log_e(1+x)$<br>$\log_2(1+x)$<br>$\log_{10}(1+x)$ | $[-1, +\infty]$ | $x = -1$: divideByZero;<br>$x < -1$: invalid operation; underflow |
| **hypot**$(x, y)$ | $\sqrt{(x^2+y^2)}$ | $[-\infty, +\infty]$ **×** $[-\infty, +\infty]$ | overflow; underflow |
| **rSqrt** | $1/\sqrt{x}$ | $[0, +\infty]$ | $x < 0$: invalid operation;<br>$x$ is $\pm 0$: divideByZero |
| **compound**$(x, n)$ | $(1+x)^n$ | $[-1, +\infty]$ **× Z** | $x < -1$: invalid operation; overflow; underflow |
| **rootn**$(x, n)$ | $x^{1/n}$ | $[-\infty, +\infty]$ **× Z** | $n = 0$: invalid operation;<br>$x < 0$ and n even: invalid operation;<br>$n = -1$: overflow, underflow<br>$x = 0$ and $n < 0$: divideByZero |
| **pown**$(x, n)$ | $x^n$ | $[-\infty, +\infty]$ **× Z** | overflow; underflow |
| **pow**$(x, y)$ | $x^y$ | $[-\infty, +\infty]$ **×** $[-\infty, +\infty]$ | overflow; underflow |
| **powr**$(x, y)$ | $x^y$ | $[0, +\infty]$ **×** $[-\infty, +\infty]$ | overflow; underflow |

See continuation overleaf for circular and hyperbolic trigonometric operations.

**Table 9.1—Additional mathematical operations (continued)**

| Operation | Function | Domain | Other exceptions; see also 9.2.1 |
|---|---|---|---|
| **sin** | $\sin(x)$ | $(-\infty, +\infty)$ | $\|x\| = \infty$: invalid operation; underflow |
| **cos** | $\cos(x)$ | $(-\infty, +\infty)$ | $\|x\| = \infty$: invalid operation |
| **tan** | $\tan(x)$ | $(-\infty, +\infty)$ | $\|x\| = \infty$: invalid operation; underflow |
| **sinPi** | $\sin(\pi \times x)$ | $(-\infty, +\infty)$ | $\|x\| = \infty$: invalid operation; underflow |
| **cosPi** | $\cos(\pi \times x)$ | $(-\infty, +\infty)$ | $\|x\| = \infty$: invalid operation |
| **tanPi** | $\tan(\pi \times x)$ | $(-\infty, +\infty)$ | $\|x\| = \infty$: invalid operation; underflow;<br>$\|x\| = (n + 0.5)$ for integer $n$: divideByZero |
| **asin** | $\text{asin}(x)$ | $[-1, +1]$ | $\|x\| > 1$: invalid operation; underflow |
| **acos** | $\text{acos}(x)$ | $[-1, +1]$ | $\|x\| > 1$: invalid operation |
| **atan** | $\text{atan}(x)$ | $[-\infty, +\infty]$ | underflow |
| **atan2**$(y, x)$ | see 9.2.1 | $[-\infty, +\infty] \times [-\infty, +\infty]$ | underflow |
| **asinPi** | $\text{asin}(x)/\pi$ | $[-1, +1]$ | $\|x\| > 1$: invalid operation; underflow |
| **acosPi** | $\text{acos}(x)/\pi$ | $[-1, +1]$ | $\|x\| > 1$: invalid operation |
| **atanPi** | $\text{atan}(x)/\pi$ | $[-\infty, +\infty]$ | underflow |
| **atan2Pi**$(y, x)$ | see 9.2.1 | $[-\infty, +\infty] \times [-\infty, +\infty]$ | underflow |
| **sinh** | $\sinh(x)$ | $[-\infty, +\infty]$ | overflow; underflow |
| **cosh** | $\cosh(x)$ | $[-\infty, +\infty]$ | overflow |
| **tanh** | $\tanh(x)$ | $[-\infty, +\infty]$ | underflow |
| **asinh** | $\text{asinh}(x)$ | $[-\infty, +\infty]$ | underflow |
| **acosh** | $\text{acosh}(x)$ | $[+1, +\infty]$ | $x < 1$: invalid operation |
| **atanh** | $\text{atanh}(x)$ | $[-1, +1]$ | underflow;<br>$\|x\| = 1$: divideByZero;<br>$\|x\| > 1$: invalid operation |

Interval notation is used for the domain: a value adjacent to a bracket is included in the domain and a value adjacent to a parenthesis is not. **Z** is the set of integers.

The notation A × B in the domain denotes the set of ordered pairs of elements $(a, b)$ where $a$ is an element of A and $b$ is an element of B.

The operations **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, and **atan2** measure angles in radians. The operations **sinPi**, **cosPi**, **tanPi**, **asinPi**, **acosPi**, **atanPi**, and **atan2Pi** measure angles in half-revolutions.

For operations $f$ defined by even mathematical functions, $f(-x)$ is $f(x)$ for all rounding attributes for their entire domain and range. For operations $f$ defined by odd mathematical functions, $f(-x)$ is $-f(x)$ for roundTiesToEven, roundTiesToAway, and roundTowardZero for their entire domain and range. **atan2**$(y, x)$ and **atan2Pi**$(y, x)$ are odd in their first operand. **hypot**$(x, y)$ is even in both operands.

NOTE—Non-interchange formats with very large precision relative to exponent range might signal additional exceptions not listed in Table 9.1. For instance, **log** might signal underflow and **tan** might signal overflow.

### 9.2.1 Special values

For the operations **sin**, **tan**, **sinPi**, **tanPi**, **asin**, **atan**, **asinPi**, **atanPi**, **sinh**, **tanh**, **asinh**, **atanh**, **expm1**, **exp2m1**, **exp10m1**, **logp1**, **log2p1**, and **log10p1**, $f(+0)$ is +0 and $f(-0)$ is −0 with no exception.

For the operations **cos**, **cosPi**, **cosh**, **exp**, **exp2**, and **exp10**, $f(\pm 0)$ is +1 with no exception.

**sinPi**(+$n$) is +0 and **sinPi**(−$n$) is −0 for positive integers $n$. This implies, under roundTiesToEven, roundTiesToAway, and roundTowardZero, that **sinPi**(−$x$) and −**sinPi**($x$) are the same number (or both NaN) for all $x$. **cosPi**($n$ + ½) is +0 for any integer $n$ when $n$ + ½ is representable. This implies that **cosPi**(−$x$) and **cosPi**($x$) are the same number (or both NaN) for all $x$, in all rounding directions.

For integer $n \geq 0$:
**tanPi**($2 \times n + 0.5$) is +∞ and signals the divideByZero exception
**tanPi**($2 \times n + 1$) is −0
**tanPi**($2 \times n + 1.5$) is −∞ and signals the divideByZero exception
**tanPi**($2 \times n + 2$) is +0
For integer $n \leq 0$:
**tanPi**($2 \times n - 0.5$) is −∞ and signals the divideByZero exception
**tanPi**($2 \times n - 1$) is +0
**tanPi**($2 \times n - 1.5$) is +∞ and signals the divideByZero exception
**tanPi**($2 \times n - 2$) is −0
This implies, under roundTiesToEven, roundTiesToAway, and roundTowardZero, that **tanPi**(−$x$) and −**tanPi**($x$) are the same number (or both NaN) for all $x$.

**acos**(1), **acosPi**(1), and **acosh**(1) are +0.

**atan**(±∞) is ±π/2 rounded and thus should signal the inexact exception.

**atan2**($y$, $x$) is the angle subtended at the origin by the point ($x$, $y$) and the positive $x$-axis; that angle is also the argument or phase or imaginary part of the logarithm of the complex number $x + \mathrm{i}y$. The unrounded range of **atan2** is [−π, +π].

For $y$ with positive sign bit, the general cases of **atan2**($y$, $x$) for finite non-zero numeric $x$ are correctly rounded from the following expressions:
**atan2**($y$, $x$) for finite $x > 0$ is **atan**($|y/x|$), which could signal the underflow exception
**atan2**($y$, $x$) for finite $x < 0$ is π − **atan**($|y/x|$).

The special cases of **atan2**($y$, $x$) involving 0 and ∞ are constants which should signal the inexact exception when the result is non-zero, but they signal no other exception:
**atan2**(±0, −0) is ±π
**atan2**(±0, +0) is ±0
**atan2**(±0, $x$) is ±π for $x < 0$
**atan2**(±0, $x$) is ±0 for $x > 0$
**atan2**($y$, ±0) is −π/2 for $y < 0$
**atan2**($y$, ±0) is +π/2 for $y > 0$
**atan2**(±$y$, −∞) is ±π for finite $y > 0$
**atan2**(±$y$, +∞) is ±0 for finite $y > 0$
**atan2**(±∞, $x$) is ±π/2 for finite $x$
**atan2**(±∞, −∞) is ±3π/4
**atan2**(±∞, +∞) is ±π/4.

For some formats under some rounding attributes the rounded magnitude range of **atan** (**atan2**) might exceed the unrounded magnitude of π/2 (π). A programmer must then take care to properly handle any anomalous manifold jump that might occur under the inverse operation.

**atanPi**(±∞) is ±½ with no exception.

**atan2Pi**($y$, $x$) is the angle subtended at the origin by the point ($x$, $y$) and the positive $x$-axis. The range of **atan2Pi** is [−1, +1].

For $y$ with positive sign bit, the general cases of **atan2Pi**($y$, $x$) for finite non-zero numeric $x$ are correctly rounded from the following expressions:

**atan2Pi**($y$, $x$) for finite $x>0$ is **atan**($|y/x|$)/π, which could signal the underflow exception
**atan2Pi**($y$, $x$) for finite $x<0$ is 1− **atan**($|y/x|$)/π.

The special cases of **atan2Pi**($y$, $x$) involving 0 and ∞ are exact constants that signal no exception:

**atan2Pi**(±0, −0) is ±1
**atan2Pi**(±0, +0) is ±0
**atan2Pi**(±0, $x$) is ±1 for $x<0$
**atan2Pi**(±0, $x$) is ±0 for $x>0$
**atan2Pi**($y$, ±0) is −½ for $y<0$
**atan2Pi**($y$, ±0) is +½ for $y>0$
**atan2Pi**(±$y$, −∞) is ±1 for finite $y>0$
**atan2Pi**(±$y$, +∞) is ±0 for finite $y>0$
**atan2Pi**(±∞, $x$) is ±½ for finite $x$
**atan2Pi**(±∞, −∞) is ±¾
**atan2Pi**(±∞, +∞) is ±¼.

**sinh**(±∞) and **asinh**(±∞) are ±∞ with no exception. **cosh**(±∞) and **acosh**(+∞) are +∞ with no exception. **tanh**(±∞) is ±1 with no exception. **atanh**(±1) is ±∞ and signals the divideByZero exception.

For the operations **exp**, **exp2**, and **exp10**, $f(+\infty)$ is +∞ and $f(-\infty)$ is +0 with no exception. For the operations **expm1**, **exp2m1**, and **exp10m1**, $f(+\infty)$ is +∞ and $f(-\infty)$ is −1 with no exception.

For the operations **log**, **log2**, **log10**, **logp1**, **log2p1**, and **log10p1**, $f(+\infty)$ is +∞ with no exception. For the operations **log**, **log2**, and **log10**, $f(\pm 0)$ is −∞ and signals the divideByZero exception, and $f(1)$ is +0. For the operations **logp1**, **log2p1**, and **log10p1**, $f(-1)$ is −∞ and signals the divideByZero exception.

For the **hypot** operation, **hypot**(±0, ±0) is +0, **hypot**(±∞, qNaN) is +∞, and **hypot**(qNaN, ±∞) is +∞.

**rSqrt**(+∞) is +0 with no exception. **rSqrt**(±0) is ±∞ and signals the divideByZero exception.

For the **compound**, **rootn**, and **pown** operations, $n$ is a finite integral value in *integralFormat*. When *integralFormat* is a floating-point format, the behavior of these operations is language-defined when the second operand is non-integral or infinite.

For the **compound** operation:

**compound**($x$, 0) is 1 for $x\geq -1$ or quiet NaN
**compound**(−1, $n$) is +∞ and signals the divideByZero exception for $n<0$
**compound**(−1, $n$) is +0 for $n>0$
**compound**(±0, $n$) is 1
**compound**(+∞, $n$) is +∞ for $n>0$
**compound**(+∞, $n$) is +0 for $n<0$
**compound**($x$, $n$) is qNaN and signals the invalid operation exception for $x<-1$
**compound**(qNaN, $n$) is qNaN for $n\neq 0$.

For the **rootn** operation:

**rootn**(±0, $n$) is ±∞ and signals the divideByZero exception for odd $n<0$
**rootn**(±0, $n$) is +∞ and signals the divideByZero exception for even $n<0$
**rootn**(±0, $n$) is +0 for even $n>0$
**rootn**(±0, $n$) is ±0 for odd $n>0$
**rootn**(+∞, $n$) is +∞ for $n>0$
**rootn**(−∞, $n$) is −∞ for odd $n>0$
**rootn**(−∞, $n$) is qNaN and signals the invalid operation exception for even $n>0$
**rootn**(+∞, $n$) is +0 for $n<0$
**rootn**(−∞, $n$) is −0 for odd $n<0$
**rootn**(−∞, $n$) is qNaN and signals the invalid operation exception for even $n<0$.

NOTE—**rootn** (−0, 2) differs from **squareRoot**(−0) because they have different consistency considerations.

For the **pown** operation:

**pown**($x$, 0) is 1 if $x$ is not a signaling NaN
**pown**($\pm 0$, $n$) is $\pm\infty$ and signals the divideByZero exception for odd $n<0$
**pown**($\pm 0$, $n$) is $+\infty$ and signals the divideByZero exception for even $n<0$
**pown**($\pm 0$, $n$) is +0 for even $n>0$
**pown**($\pm 0$, $n$) is $\pm 0$ for odd $n>0$
**pown**($+\infty$, $n$) is $+\infty$ for $n>0$
**pown**($-\infty$, $n$) is $-\infty$ for odd $n>0$
**pown**($-\infty$, $n$) is $+\infty$ for even $n>0$
**pown**($+\infty$, $n$) is +0 for $n<0$
**pown**($-\infty$, $n$) is $-0$ for odd $n<0$
**pown**($-\infty$, $n$) is +0 for even $n<0$.

For the **pow** operation:

**pow**($x$, $\pm 0$) is 1 if $x$ is not a signaling NaN
**pow**($\pm 0$, $y$) is $\pm\infty$ and signals the divideByZero exception for $y$ an odd integer $<0$
**pow**($\pm 0$, $-\infty$) is $+\infty$ with no exception
**pow**($\pm 0$, $+\infty$) is +0 with no exception
**pow**($\pm 0$, $y$) is $\pm 0$ for finite $y>0$ an odd integer
**pow**($-1$, $\pm\infty$) is 1 with no exception
**pow**($+1$, $y$) is 1 for any $y$ (even a quiet NaN)
**pow**($x$, $+\infty$) is +0 for $-1<x<1$
**pow**($x$, $+\infty$) is $+\infty$ for $x<-1$ or for $1<x$ (including $\pm\infty$)
**pow**($x$, $-\infty$) is $+\infty$ for $-1<x<1$
**pow**($x$, $-\infty$) is +0 for $x<-1$ or for $1<x$ (including $\pm\infty$)
**pow**($+\infty$, $y$) is +0 for a number $y<0$
**pow**($+\infty$, $y$) is $+\infty$ for a number $y>0$
**pow**($-\infty$, $y$) is $-0$ for finite $y<0$ an odd integer
**pow**($-\infty$, $y$) is $-\infty$ for finite $y>0$ an odd integer
**pow**($-\infty$, $y$) is +0 for finite $y<0$ and not an odd integer
**pow**($-\infty$, $y$) is $+\infty$ for finite $y>0$ and not an odd integer
**pow**($\pm 0$, $y$) is $+\infty$ and signals the divideByZero exception for finite $y<0$ and not an odd integer
**pow**($\pm 0$, $y$) is +0 for finite $y>0$ and not an odd integer
**pow**($x$, $y$) signals the invalid operation exception for finite $x<0$ and finite non-integer $y$.

NOTE—To support special cases that could occur in decimal floating-point numbers but not in binary floating-point numbers when x is negative and y is not an odd integer, language standards might define another power operation, **powd**, whose specification expands the last five rules above as:

**powd**($-\infty$, $y$) is +0 for finite $y<0$ an even integer
**powd**($-\infty$, $y$) is $+\infty$ for finite $y>0$ an even integer
**powd**($+0$, $y$) is $+\infty$ and signals the divideByZero exception for finite $y<0$ and not an odd integer
**powd**($-0$, $y$) is $+\infty$ and signals the divideByZero exception for finite $y<0$ an even integer
**powd**($+0$, $y$) is +0 for finite $y>0$ and not an odd integer
**powd**($-0$, $y$) is +0 for finite $y>0$ an even integer
**powd**($-1$, $y$) is +1 for finite non-integer $y$ whose simplest form is $m/n$ with $m$ even and $n$ odd
**powd**($-1$, $y$) is $-1$ for finite non-integer $y$ whose simplest form is $m/n$ with both $m$ and $n$ odd
**powd**($-1$, $y$) is qNaN and signals the invalid operation exception for finite non-integer $y$ whose simplest form is $m/n$ with $n$ even
**powd**($x$, $y$) is **powd**($-1$, $y$) × **powd**(abs($x$), $y$) for finite non-integer $y$ and negative $x$ (including $-0$, finite negative $x$, and $-\infty$).

For the **powr** operation:

**powr**($x$, $\pm 0$) is 1 for finite $x>0$
**powr**($\pm 0$, $y$) is $+\infty$ and signals the divideByZero exception for finite $y<0$
**powr**($\pm 0$, $-\infty$) is $+\infty$
**powr**($\pm 0$, $y$) is +0 for $y>0$
**powr**($+1$, $y$) is 1 for finite $y$
**powr**($x$, $y$) signals the invalid operation exception for $x<0$
**powr**($\pm 0$, $\pm 0$) signals the invalid operation exception

**powr**(+∞, ±0) signals the invalid operation exception
**powr**(+1, ±∞) signals the invalid operation exception
**powr**(*x*, qNaN) is qNaN for $x \geq 0$
**powr**(qNaN, *y*) is qNaN.

NOTE—This standard defines several operations to raise *x* to a given power:
**pown**(*x*, *n*) accepts integral powers *n* only, for any *x*
**pow**(*x*, *y*) behaves the same as **pown**(*x*, *n*) when y contains a value which is equal to an integral *n*
**powr**(*x*, *y*) is defined by considering **exp**(*y* × **log**(*x*)), and thus its domain excludes negative *x*.

### 9.2.2 Preferred exponents

The preferred exponent for operations in sub-clause 9.2 is 0, except for the following:

Q(**hypot**(*x*, *y*)) is min(Q(*x*), Q(*y*))
Q(**rSqrt**(*x*)) is −floor(Q(*x*)/2)
Q(**compound**(*x*, *n*)) is floor(*n* × min(0, Q(*x*)))
Q(**rootn**(*x*, *n*)) is floor(Q(*x*)/*n*)
Q(**pow**(*x*, *y*)) is floor(*y* × Q(*x*))
Q(**pown**(*x*, *n*)) is floor(*n* × Q(*x*))
Q(**powr**(*x*, *y*)) is floor(*y* × Q(*x*)).

## 9.3 Dynamic mode operations

### 9.3.1 Operations on individual dynamic modes

Language standards that define dynamic mode specification (see 4.2) for binary or decimal rounding directions shall define corresponding non-computational operations to get and set the applicable value of each specified dynamic mode rounding direction. The applicable value of the rounding direction might have been set by a constant attribute specification or a dynamic-mode assignment, according to the scoping rules of the language. The effect of these operations, if used outside the static scope of a dynamic specification for a rounding direction, is language-defined (and may be unspecified).

Language standards that define dynamic mode specification for binary rounding direction shall define:

— *binaryRoundingDirection* **getBinaryRoundingDirection**(*void*)
— *void* **setBinaryRoundingDirection**(*binaryRoundingDirection*).

Language standards that define dynamic mode specification for decimal rounding direction shall define:

— *decimalRoundingDirection* **getDecimalRoundingDirection**(*void*)
— *void* **setDecimalRoundingDirection**(*decimalRoundingDirection*).

Language standards that define dynamic mode specification for other attributes shall define corresponding operations to get and set those dynamic modes.

### 9.3.2 Operations on all dynamic modes

Language standards that define dynamic mode specification shall define the following non-computational operations for all dynamic-specifiable modes collectively:

— *modeGroup* **saveModes(***void*)
  Saves the values of all dynamic-specifiable modes as a group.
— *void* **restoreModes**(*modeGroup*)
  Restores the values of all dynamic-specifiable modes as a group.
— *void* **defaultModes**(*void*)
  Sets all dynamic-specifiable modes to default values.

*modeGroup* represents the set of dynamically-specifiable modes. The return values of the **saveModes** operation are for use as operands of the **restoreModes** operation in the same program; this standard does not require support for any other use.

## 9.4 Reduction operations

Language standards should define the following reduction operations for all supported arithmetic formats. Unlike the other operations in this standard, these operate on vectors of operands in one format and return a result in the same format. Implementations may associate in any order or evaluate in any wider format.

The vector length operand shall have integral values in a language-defined format, *integralFormat*. If *integralFormat* is a floating-point format, it shall have a precision at least as large as *sourceFormat* and have the same radix. The behavior of these operations is language-defined when the vector length operand is non-integral or negative.

Numerical results and exceptional behavior, including the invalid operation exception and its sub-exceptions, might differ due to the precision of intermediates and the order of evaluation. However, only one exception is signaled per reduction operation invocation; exceptions are not signaled for each exceptional intermediate operand or result. All reduction operations signal the invalid operation exception if any operand is a signaling NaN. Once an invalid operation condition is signaled, due to signaling NaN, $\infty - \infty$, or $0 \times \infty$, processing of vector elements may stop.

Under default exception handling, inexact is also signaled when these operations signal overflow or inexact underflow. Otherwise whether the inexact exception is signaled is not specified.

Preferred exponents and cohort members of results are not specified for these operations.

Language standards should define the following sum reductions:

- *sourceFormat* **sum**(*source vector, integralFormat*)
  **sum**(*p, n*) is an implementation-defined approximation to $\sum_{(i=1,\,n)} p_i$, where *p* is a vector of length *n*.
- *sourceFormat* **dot**(*source vector, source vector, integralFormat*)
  **dot**(*p, q, n*) is an implementation-defined approximation to $\sum_{(i=1,\,n)} (p_i \times q_i)$, where *p* and *q* are vectors of length *n*.
- *sourceFormat* **sumSquare**(*source vector, integralFormat*)
  **sumSquare**(*p, n*) is an implementation-defined approximation to $\sum_{(i=1,\,n)} p_i^2$, where *p* is a vector of length *n*.
- *sourceFormat* **sumAbs**(*source vector, integralFormat*)
  **sumAbs**(*p, n*) is an implementation-defined approximation to $\sum_{(i=1,\,n)} |p_i|$, where *p* is a vector of length *n*.

For **sum** and **dot**, if any operand element is a NaN a quiet NaN is returned. A product of $\infty \times 0$ signals the invalid operation exception. A sum of infinities of different signs signals the invalid operation exception. Otherwise, a sum of infinities of the same sign returns that infinity and does not signal any exception. Otherwise, sums are computed with no avoidable intermediate exception conditions in the calculation and the final result is determined from that intermediate result. If the final result overflows, signal overflow. If the final result underflows, signal underflow.

For **sumSquare** and **sumAbs**, if any operand element is an infinity, $+\infty$ is returned. Otherwise, if any operand element is a NaN a quiet NaN is returned. Otherwise, sums are computed with no avoidable intermediate exception conditions in the calculation and the final result is determined from that. If the final result overflows, signal overflow. If the final result underflows, signal underflow.

When the vector length operand is zero, the return value is +0 without exception.

Language standards should define the following scaled product reduction operations:

- (*sourceFormat, integralFormat*) **scaledProd**(*source vector, integralFormat*)
  **scaledProd**(*p, n*) returns {*pr, sf*} so that **scaleB**(*pr, sf*) is an implementation-defined approximation to $\prod_{(i\,=\,1,\,n)} p_i$, where *p* is a vector of length *n.*
- (*sourceFormat, integralFormat*) **scaledProdSum**(*source vector, source vector, integralFormat*)
  **scaledProdSum**(*p, q, n*) returns {*pr, sf*} so that **scaleB**(*pr, sf*) is an implementation-defined approximation to $\prod_{(i\,=\,1,\,n)} (p_i + q_i)$, where *p* and *q* are vectors of length *n.*

— (*sourceFormat, integralFormat*) **scaledProdDiff**(*source vector, source vector, integralFormat*) **scaledProdDiff**(*p, q, n*) returns {*pr, sf*} so that **scaleB**(*pr, sf*) is an implementation-defined approximation to $\prod_{(i = 1, n)} (p_i - q_i)$, where *p* and *q* are vectors of length *n*.

The vector operands and the scaled product member of the result shall be of the same format. The vector length operand and the scale factor member of the result shall have integral values and should be of the same language-defined format, *integralFormat*.

For **scaledProd**, **scaledProdSum**, and **scaledProdDiff**, if any operand element is a NaN a quiet NaN is returned. A product of $\infty \times 0$ signals the invalid operation exception. A sum of infinities of different signs (or a difference of infinities of like signs) signals the invalid operation exception. Otherwise, if there are infinities in the product, an infinity is returned and the invalid operation exception is not signaled. Otherwise, if there are zeros in the product, a zero is returned and the invalid operation exception is not signaled.

In the absence of any of the above, the scaled result, *pr*, shall not be affected by overflow or underflow. These operations should not signal the divideByZero exception, even if implemented with **logB**. If the scale factor is too large in magnitude to be represented exactly in the format of *sf,* then these operations shall signal the invalid operation exception and by default return quiet NaN for *pr*, and also for *sf* if *integralFormat* is a floating-point format. When the vector length operand is zero, *pr* is 1 and *sf* is +0 without exception.

## 9.5 Augmented arithmetic operations

Language standards should define for binary formats, to be implemented according to 9.1, this clause's operations as is appropriate to the language. These homogeneous operations produce a pair of *sourceFormat* results denoted *(sourceFormat, sourceFormat)*. This standard recommends only the operations for binary formats because the requirements to address augmenting decimal format arithmetic are not yet determined.

This standard specifies a single rounding direction to be used in the operations in this subclause, defined as roundTiesTowardZero: the floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with smaller magnitude shall be delivered. However, an infinitely precise result with magnitude greater than $b^{emax} \times (b - \frac{1}{2} b^{1-p})$ shall round to $\infty$ with no change in sign; here *emax* and $p$ are determined by the destination format (see 3.3). Thus, roundTiesTowardZero carries all overflows (see 7.4) to $\infty$ with the sign of the intermediate result. An infinitely precise result with magnitude equal to $b^{emax} \times (b - \frac{1}{2} b^{1-p})$ shall round to $b^{emax} \times (b - b^{1-p})$ with no change in sign.

In the specification in this subclause, roundTiesTowardZero($x$ operation $y$) denotes the infinitely precise result of $x$ operation $y$ rounded using roundTiesTowardZero, where operation is +, −, or ×.

— *(sourceFormat, sourceFormat)* **augmentedAddition**(*source*, *source*)

The operation **augmentedAddition**($x$, $y$) computes both the infinitely precise sum $x + y$ rounded to *sourceFormat* using roundTiesTowardZero and the error in rounding the sum when the rounded $x + y$ is finite. If roundTiesTowardZero($x + y$) is a finite number, **augmentedAddition** produces roundTiesTowardZero($x + y$) and $x + y$ − roundTiesTowardZero($x + y$), where if $x + y$ − roundTiesTowardZero($x + y$) equals zero, it is returned with the sign of roundTiesTowardZero($x + y$).

This operation's exceptional behavior is the same as that of **addition** (see 5.4.1) using roundTiesTowardZero, with the following additional specifications. The operation propagates a NaN as both results if any input is a NaN (see 6.2.3). If roundTiesTowardZero($x + y$) is infinite, both produced results are the result of roundTiesTowardZero($x + y$) and the operation signals like **addition**($x$, $y$) using roundTiesTowardZero. If the operation signals the invalid operation exception, it produces the same quiet NaN for both outputs. If $x + y$ − roundTiesTowardZero($x + y$) is non-zero and lies strictly between $\pm b^{emin}$, the underflow exception shall be signaled. Under default exception handling, the operation signals inexact only when roundTiesTowardZero($x + y$) overflows; the operation's subnormal and zero results are exact.

— *(sourceFormat, sourceFormat)* **augmentedSubtraction**(*source*, *source*)

The operation **augmentedSubtraction**($x$, $y$) computes both the infinitely precise difference $x - y$ rounded to *sourceFormat* using roundTiesTowardZero and the error in rounding the difference when the rounded $x - y$ is finite. If roundTiesTowardZero($x - y$) is a finite number, **augmentedSubtraction** produces roundTiesTowardZero($x - y$) and $x - y$ − roundTiesTowardZero($x - y$), where if $x - y$ − roundTiesTowardZero($x - y$) equals zero, it is returned with the sign of roundTiesTowardZero($x - y$).

This operation's exceptional behavior is the same as that of **subtraction** (see 5.4.1) using roundTiesTowardZero, with the following additional specifications. The operation propagates a NaN as both results if any input is a NaN (see 6.2.3). If roundTiesTowardZero($x - y$) is infinite, both produced results are the result of roundTiesTowardZero($x - y$) and the operation signals like **subtraction**($x$, $y$) using roundTiesTowardZero. If the operation signals the invalid operation exception, it produces the same quiet NaN for both outputs. If $x - y$ − roundTiesTowardZero($x - y$) is non-zero and lies strictly between $\pm b^{emin}$, the underflow exception shall be signaled. Under default exception handling, the operation signals inexact only when roundTiesTowardZero($x - y$) overflows; the operation's subnormal and zero results are exact.

— *(sourceFormat, sourceFormat)* **augmentedMultiplication**(*source*, *source*)

The operation **augmentedMultiplication**($x$, $y$) computes both the infinitely precise product $x \times y$ rounded to *sourceFormat* using roundTiesTowardZero and the error in rounding the product when the rounded $x \times y$ is finite, under the conditions described below. If roundTiesTowardZero($x \times y$) is a finite number and $x \times y -$ roundTiesTowardZero($x \times y$) can be represented exactly as a finite number in sourceFormat, **augmentedMultiplication** produces roundTiesTowardZero($x \times y$) and $x \times y -$ roundTiesTowardZero($x \times y$), where if $x \times y -$ roundTiesTowardZero($x \times y$) equals zero, it is returned with the sign of roundTiesTowardZero($x \times y$).

This operation's exceptional behavior is the same as that of **multiplication** (see 5.4.1) using roundTiesTowardZero, with the following additional specifications. The operation propagates a NaN as both results if any input is a NaN (see 6.2.3). If roundTiesTowardZero($x \times y$) is infinite, both produced results are the result of roundTiesTowardZero($x \times y$). If the operation signals the invalid operation exception, it produces the same quiet NaN for both outputs. If $x \times y -$ roundTiesTowardZero($x \times y$) is non-zero and lies strictly between $\pm b^{emin}$, the underflow exception shall be signaled. If $x \times y -$ roundTiesTowardZero($x \times y$) is finite and non-zero and cannot be represented exactly in *sourceFormat* (because some non-zero digits lie strictly between $\pm b^{(emin - p + 1)}$), the results are roundTiesTowardZero($x \times y$) and the infinitely precise result of $x \times y -$ roundTiesTowardZero($x \times y$) rounded to *sourceFormat* using roundTiesTowardZero. Default exception handling raises the underflow flag and signals the inexact exception in this case. Otherwise, under default exception handling, the operation signals inexact only when roundTiesTowardZero($x \times y$) overflows.

## 9.6 Minimum and maximum operations

Language standards should define the following homogeneous general-computational operations for all supported arithmetic formats:

— *sourceFormat* **minimum**(*source, source*)
*sourceFormat* **minimumNumber**(*source, source*)
*sourceFormat* **maximum**(*source, source*)
*sourceFormat* **maximumNumber**(*source, source*)

**minimum**($x$, $y$) is $x$ if $x < y$, $y$ if $y < x$, and a quiet NaN if either operand is a NaN, according to 6.2. For this operation, −0 compares less than +0. Otherwise (*i.e.*, when $x = y$ and signs are the same) it is either $x$ or $y$.

**minimumNumber**($x$, $y$) is $x$ if $x < y$, $y$ if $y < x$, and the number if one operand is a number and the other is a NaN. For this operation, −0 compares less than +0. If $x = y$ and signs are the same it is either $x$ or $y$. If both operands are NaNs, a quiet NaN is returned, according to 6.2. If either operand is a signaling NaN, an invalid operation exception is signaled, but unless both operands are NaNs, the signaling NaN is otherwise ignored and not converted to a quiet NaN as stated in 6.2 for other operations.

**maximum**($x$, $y$) is $x$ if $x > y$, $y$ if $y > x$, and a quiet NaN if either operand is a NaN, according to 6.2. For this operation, +0 compares greater than −0. Otherwise (*i.e.*, when $x = y$ and signs are the same) it is either $x$ or $y$.

**maximumNumber**($x$, $y$) is $x$ if $x > y$, $y$ if $y > x$, and the number if one operand is a number and the other is a NaN. For this operation, +0 compares greater than −0. If $x = y$ and signs are the same it is either $x$ or $y$. If both operands are NaNs, a quiet NaN is returned, according to 6.2. If either operand is a signaling NaN, an invalid operation exception is signaled, but unless both operands are NaNs, the signaling NaN is otherwise ignored and not converted to a quiet NaN as stated in 6.2 for other operations.

— *sourceFormat* **minimumMagnitude**(*source, source*)
*sourceFormat* **minimumMagnitudeNumber**(*source, source*)
*sourceFormat* **maximumMagnitude**(*source, source*)
*sourceFormat* **maximumMagnitudeNumber**(*source, source*)

**minimumMagnitude**($x$, $y$) is $x$ if $|x|<|y|$, $y$ if $|y|<|x|$, otherwise **minimum**($x$, $y$).

**minimumMagnitudeNumber**($x$, $y$) is $x$ if $|x|<|y|$, $y$ if $|y|<|x|$, otherwise **minimumNumber**($x$, $y$).

**maximumMagnitude**($x$, $y$) is $x$ if $|x|>|y|$, $y$ if $|y|>|x|$, otherwise **maximum**($x$, $y$).

**maximumMagnitudeNumber**($x$,$y$) is $x$ if $|x|>|y|$, $y$ if $|y|>|x|$, otherwise **maximumNumber**($x$, $y$).

The preferred exponent is Q($x$) if $x$ is the result, Q($y$) if $y$ is the result.

NOTE—The quantum of the result might differ among implementations when $x$ and $y$ are different representations of the same cohort in decimal floating-point numbers.

## 9.7 NaN payload operations

Language standards should define the following homogeneous quiet-computational operations to provide generic access to payloads. These operations signal no exceptions.

— *sourceFormat* **getPayload**(*source*)

If the source operand is a NaN, the result is the payload as a non-negative floating-point integer. If the source operand is not a NaN, the result is −1.

The preferred exponent is 0.

— *sourceFormat* **setPayload**(*source*)

If the source operand is a non-negative floating-point integer whose value is one of an implementation-defined set of admissible payloads for the operation, the result is a quiet NaN with that payload. Otherwise, the result is +0, with a preferred exponent of 0.

— *sourceFormat* **setPayloadSignaling**(*source*)

If the source operand is a non-negative floating-point integer whose value is one of an implementation-defined set of admissible payloads for the operation, the result is a signaling NaN with that payload. Otherwise, the result is +0, with a preferred exponent of 0.

NOTE—An implementation may restrict the payloads that can be set. Thus **getPayload** might return a value that is not an admissible operand for **setPayload** or **setPayloadSignaling**. (A program can check by applying the **isNaN** operation to the result of **setPayload** or **setPayloadSignaling**.)

# 10. Expression evaluation

## 10.1 Expression evaluation rules

Clause 5 of this standard specifies the result of a single arithmetic operation. Every operation has an explicit or implicit destination. For numerical results, one rounding occurs to fit the exact result into a destination format. That result is reproducible in that the same operation applied to the same operands under the same attributes produces the same result on all conforming implementations in all languages.

Programming language standards might define syntax for expressions that combine one or more operations of this standard, producing a result to fit an explicit or implicit final destination. When a variable with a declared format is a final destination, as in format conversion to a variable, that declared format of that variable governs its rounding. The format of an implicit destination, or of an explicit destination without a declared format, is defined by language standard expression evaluation rules.

A programming language standard specifies one or more rules for expression evaluation. A rule for expression evaluation encompasses:

- The order of evaluation of operations.
- The formats of implicit intermediate results.
- When assignments to explicit destinations round once, and when twice (see below).
- The formats of parameters to generic and non-generic operations.
- The formats of results of generic operations.

Language standards might permit the user to select different language-defined rules for expression evaluation, and might allow implementations to define additional expression evaluation rules and specify the default expression evaluation rule; in these cases language standards should define preferredWidth attributes as specified below.

Some language standards implicitly convert operands of floating-point operations to a common format. Typically, operands are promoted to the widest format of the operands or a preferredWidth format. However, if the common format is not a superset of the operand formats, then the conversion of an operand to the common format might not preserve the values of the operands. Examples include:

- Converting a fixed-point or integer operand to a floating-point format with less precision.
- Converting a floating-point operand from one radix to another.
- Converting a floating-point operand to a format with the same radix but with either less range or less precision.

Language standards should disallow, or provide warnings for, mixed-format operations that would cause implicit conversion that might change operand values.

## 10.2 Assignments, parameters, and function values

The last operation of many expressions is an assignment to an explicit final destination variable. As a part of expression evaluation rules, language standards shall specify when the next to last operation is performed by rounding at most once to the format of the explicit final destination, and when by rounding as many as two times, first to an implicit intermediate format, and then to the explicit final destination format. The latter case does not correspond to any single operation in Clause 5 but implies a sequence of two such operations.

In either case, implementations shall never use an assigned-to variable’s wider precursor in place of the assigned-to variable’s stored value when evaluating subsequent expressions.

When a function has explicitly-declared formal parameter types in scope, the actual parameters shall be rounded if necessary to those explicitly-declared types. When a function does not have explicitly-declared formal parameter types in scope, or is a generic operation, the actual parameters shall be rounded according to language-defined rules.

When a function explicitly declares the type of its return value, the return value shall be rounded to that explicitly-declared type. When the return value type of a function is implicitly defined by language standard rules, the return value shall be rounded to that implicitly-defined type.

## 10.3 preferredWidth attributes for expression evaluation

Language standards defining generic operations, supporting more than one arithmetic format in a particular radix, and defining or allowing more than one way to map expressions in that language into the operations of this standard, should define preferredWidth attributes for each such radix. preferredWidth attributes are explicitly enabled by the user and specify one aspect of expression evaluation: the implicit destination format of language-defined generic operations.

In this standard, a computational operation that returns a numeric result first produces an unrounded result as an exact number of infinite precision. That unrounded result is then rounded to a destination format. For certain language-defined generic operations, that destination format is implied by the widths of the operands and by the preferredWidth attribute currently in effect.

The following preferredWidth attributes disable and enable widening of operations in expressions that might be as simple as $z=x+y$ or that might involve several operations on operands of different formats.

— **preferredWidthNone** attribute: Each such language standard should define, and require implementations to provide, means for users to specify a preferredWidthNone attribute for a block. Destination width is the maximum of the operand widths: generic operations with floating-point operands and results of the same radix round results to the widest format among the operands.

— **preferredWidthFormat** attributes: Each such language standard should define, and require implementations to provide, means for users to specify a preferredWidthFormat attribute for a block. The destination width is typically the maximum of the width of the preferredWidthFormat and operand widths: affected operations with floating-point operands and results (of the same radix) round results to the widest format among the operands and the preferredWidthFormat. Affected operations do not narrow their operands, which might be widened expressions. preferredWidthFormat affects only destinations in the radix of that format.

preferredWidth attributes do not affect the width of the final rounding to an explicit destination with a declared format, which is always rounded to that format. preferredWidth attributes do not affect explicit format conversions within expressions; they are always rounded to the format specified by the conversion.

## 10.4 Literal meaning and value-changing optimizations

Language standards should define the literal meaning of the source code of a program that translates to operations of this standard. The literal meaning is contained in the order of operations (controlled by precedence rules and parentheses), destination formats (implicit and explicit) for operations, and the scope of attribute or dynamic mode specifications. A language standard should require that by default, when no optimizations are enabled and no alternate exception handling is enabled, language implementations preserve the literal meaning of the source code. That means that language implementations do not perform value-changing transformations that change the numerical results or the flags raised.

A language implementation preserves the literal meaning of the source code by, for example:

- Preserving the order of operations defined by explicit sequence or parenthesization.
- Preserving the formats of explicit and implicit destinations.
- Applying the properties of real numbers to floating-point expressions only when they preserve numerical results and flags raised:
  - Applying the commutative law only to operations, such as **addition** and **multiplication**, for which neither the numerical values of the results, nor the representations of the results, depend on the order of the operands.
  - Applying the associative or distributive laws only when they preserve numerical results and flags raised.
  - Applying the identity laws ($0+x$ and $1\times x$) only when they preserve numerical results and flags raised.
- Preserving the order of operations affected by attributes or dynamic modes with respect to operations that modify attributes or dynamic modes; most computational operations are affected by attributes or dynamic modes.
- Preserving the order of operations that restore, lower, or raise status flags with respect to operations that test or save status flags; most computational operations can raise status flags.

The following value-changing transformations, among others, preserve the literal meaning of the source code:

- Applying the identity property $0+x$ when $x$ is not zero and is not a signaling NaN and the result has the same exponent as $x$.
- Applying the identity property $1\times x$ when $x$ is not a signaling NaN and the result has the same exponent as $x$.
- Changing the payload or sign bit of a quiet NaN.
- Changing the order in which different flags are raised.
- Changing the number of times a flag is raised when it is raised at least once.

A language standard should also define, and require implementations to provide, attributes that allow and disallow value-changing optimizations, separately or collectively, for a block. These optimizations might include, but are not limited to:

- Applying the associative or distributive laws.
- Synthesis of a **fusedMultiplyAdd** operation from a **multiplication** and an **addition**.
- Synthesis of a *formatOf* operation from an operation and a conversion of the result of the operation.
- Use of wider intermediate results in expression evaluation.

Programmers can allow these optimizations when the corresponding changes in numerical values or status flags are acceptable.

## 11. Reproducible floating-point results

Reproducible floating-point numerical and status flag results are possible for reproducible operations, with reproducible attributes, operating on reproducible formats, defined for each language as follows:

- A reproducible operation is one of the operations described in Clause 5 or is a supported operation from 9.2, 9.3, 9.5 or 9.6.
- A reproducible attribute is an attribute that is required by a language standard in all implementations (see Clause 4).
- A reproducible status flag is one raised by the invalid operation, division by zero, or overflow exceptions (see 7.2, 7.3, and 7.4).
- A reproducible format is an arithmetic format that is also an interchange format (see Clause 3).

Programs that can be reliably translated into an explicit or implicit sequence of reproducible operations on reproducible formats produce reproducible results. That is, the same numerical and reproducible status flag results are produced.

Reproducible results require cooperation from language standards, language processors, and users. A language standard should support reproducible programming. Any conforming language standard supporting reproducible programming shall:

- Support the reproducible-results attribute.
- Support a reproducible format by providing all the reproducible operations for that format.
- Provide means to explicitly or implicitly specify any sequence of reproducible operations on reproducible formats supported by that language.

and shall explicitly define:

- Which language element corresponds to which supported reproducible format.
- How to specify in the language each reproducible operation on each supported reproducible format.
- One or more unambiguous expression evaluation rules that shall be available for user selection on all conforming implementations of that language standard, without deferring any aspect to implementations. If a language standard permits more than one interpretation of a sequence of operations from this standard it shall provide a means of specifying an unambiguous evaluation of that sequence (such as by prescriptive parentheses).
- A reproducible-results attribute, as described in 4.1, with values to indicate when reproducible results are required or reproducible results are not required. Language standards define the default value. When the user selects reproducible results required:
  - Execution behavior shall preserve the literal meaning (see 10.4) of the source code.
  - Conversions to and from external decimal character sequence shall not limit the maximum supported precision H (see 5.12.2).
  - Language processors shall indicate where reproducibility of operations that can affect the results of floating-point operations can not be guaranteed.
  - Only default exception handling (see Clause 7) shall be used.

  If a language supports separately compiled routines (*e.g.*, library routines for common functions) there must be some mechanism to ensure reproducible behavior.

Users obtain the same floating-point numerical and reproducible status flag results, on all platforms supporting such a language standard, by writing programs that:

- Use the reproducible results required attribute.
- Use only floating-point formats that are reproducible formats.
- Use only reproducible floating-point operations explicitly, or implicitly via expressions.
- Use only attributes required in all implementations for rounding, and preferredWidth.

— Use only integer and non-floating-point formats supported in all implementations of the language standard, and only in ways that avoid signaling integer arithmetic exceptions and other implement-ation-defined exceptions.

and that

— Do not use value-changing optimizations (see 10.4).
— Do not exceed system limits.
— Do not use **fusedMultiplyAdd**(0, ∞, *c*) or **fusedMultiplyAdd**(∞, 0, *c*) where *c* is a quiet NaN.
— Do not use signaling NaNs.
— Do not depend on the quantum of a decimal result for the minimum and maximum operations of 9.6 when *x* and *y* are equal.
— Do not depend on quiet NaN propagation, payloads, or sign bits.
— Do not depend on the underflow and inexact exceptions and flags.
— Do not depend on the quantum of the results of operations on decimal formats in Table 9.1.
— Do not depend on encodings.

# Annex A

(informative)

# Bibliography

Bibliographical references are resources that provide additional or helpful material but do not need to be understood or used to implement this standard. Reference to these resources is made for informational use only.

[B1] Ahrens, P., Nguyen, H. D., and Demmel, J. "Efficient Reproducible Floating Point Summation and BLAS", EECS Department, UC Berkeley, Technical Report No. UCB/EECS-2016-121, June 18, 2016.[1]

[B2] ANSI INCITS 4–1986 Information Systems—Coded Character Sets—7-bit American National Standard Code for Information Interchange (7-Bit ASCII).[2]

[B3] Boldo, S., and Muller, J.-M., "Some functions computable with a fused-mac", *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-2366-8, pp. 52–58, IEEE Computer Society, 2005. doi:10.1109/ARITH.2005.39[3]

[B4] Bruguera, J. D., and Lang, T., "Floating-point Fused Multiply-Add: Reduced Latency for Floating-Point Addition", *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-2366-8, pp. 42–51, IEEE Computer Society, 2005. doi:10.1109/ARITH.2005.22

[B5] Coonen, J. T., "Contributions to a Proposed Standard for Binary Floating-point Arithmetic", PhD thesis, University of California, Berkeley, 1984.

[B6] Cowlishaw, M. F., "Densely Packed Decimal Encoding", *IEE Proceedings—Computers and Digital Techniques*, Vol. 149 #3, ISSN 1350-2387, pp. 102–104, IEE, London, 2002. doi:10.1049/ip-cdt:20020407

[B7] Cowlishaw, M. F., "Decimal Floating-Point: Algorism for Computers", *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-1894-X, pp. 104–111, IEEE Computer Society, 2003. doi:10.1109/ARITH.2003.1207666

[B8] Demmel, J. W., and Li., X., "Faster numerical algorithms via exception handling", *IEEE Transactions on Computers*, 43(8): pp. 983–992, 1994. doi:10.1109/12.295860

[B9] de Dinechin, F., Ershov, A., and Gast, N., "Towards the post-ultimate libm", *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-2366-8, pp. 288–295, IEEE Computer Society, 2005. doi:10.1109/ARITH.2005.46

[B10] de Dinechin, F., Lauter, C. Q., and Muller, J.-M., "Fast and correctly rounded logarithms in double-precision", *RAIRO – Theoretical Informatics and Applications*, 41, pp. 85–102, EDP Sciences, 2007. doi:10.1051/ita:2007003

[B11] Higham, N. J., *Accuracy and Stability of Numerical Algorithms*, 2nd edition, ISBN 0-89871-521-0, Society for Industrial and Applied Mathematics (SIAM), 2002.

[B12] IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems (previously designated IEC 559:1989).[4]

[B13] ISO/IEC 9899:2018(E) Programming languages—C (C17).[5]

[B14] ISO/IEC TS 18661-1:2014, Information technology—Programming languages, their environments, and system software interfaces—Floating-point extensions for C—Part 1: Binary floating-point arithmetic.

[1] Available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html.
[2] ANSI publications are available from the American National Standards Institute (http://www.ansi.org/).
[3] IEEE publications are available from The Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).
[4] IEC publications are available from the International Electrotechnical Commission (http://www.iec.ch/). IEC publications are also available in the United States from the American National Standards Institute (http://www.ansi.org).
[5] ISO/IEC publications are available from the ISO Central Secretariat (http://www.iso.org/). ISO/IEC publications are available in the United States from the American National Standards Institute (http://www.ansi.org/).

[B15] 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.

[B16] 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.

[B17] ISO/IEC TS 18661-4:2015, Information Technology—Programming languages, their environments, and system software interfaces—Floating-point extensions for C—Part 4: Supplementary functions.

[B18] ISO/IEC TS 18661-5:2016, Information Technology—Programming languages, their environments, and system software interfaces—Floating-point extensions for C—Part 5: Supplementary attributes.

[B19] Kahan, W., "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit", *The State of the Art in Numerical Analysis*, (Eds. Iserles and Powell), Clarendon Press, Oxford, 1987.

[B20] Lefèvre, V., "New results on the distance between a segment and $Z^2$. Application to the exact rounding", *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-2366-8, pp. 68–75, IEEE Computer Society, 2005. doi:10.1109/ARITH.2005.32

[B21] Lefèvre, V., and Muller, J.-M., "Worst cases for correct rounding of the elementary functions in double precision", *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-1150-3, pp. 111–118, IEEE Computer Society, 2001. doi:10.1109/ARITH.2001.930110

[B22] Markstein, P., *IA-64 and Elementary Functions: Speed and Precision*, ISBN 0-13-018348-2, Prentice Hall, Upper Saddle River, NJ, 2000.

[B23] Montoye, R. K., Hokenek, E., and Runyon, S. L., "Design of the IBM RISC System/6000 floating-point execution unit", *IBM Journal of Research and Development*, 34(1), pp. 59–70, 1990. doi:10.1147/rd.341.0059

[B24] Muller, J.-M., *Elementary Functions: Algorithms and Implementation*, 2nd edition, Chapter 10, ISBN 0-8176-4372-9, Birkhäuser, 2006.

[B25] Overton, M. L., *Numerical Computing with IEEE Floating Point Arithmetic*, ISBN 0-89871-571-7, Society for Industrial and Applied Mathematics (SIAM), 2001.

[B26] Priest, D. M., "Algorithms for arbitrary precision floating point arithmetic". *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, ISBN 0-8186-9151-4, pp. 132–143, IEEE Computer Society, 1991. doi:10.1109/ARITH.1991.145549

[B27] Sayed, W. S., and Fahmy, H. A. H., "What are the Correct Results for the Special Values of the Operands of the Power Operation?", *ACM Transactions on Mathematical Software*, ISSN:0098-3500, 42(2), pp. 14:1–14:17, 2016

[B28] Schwarz, E. M., Schmookler, M. S., and Trong, S. D., "Hardware Implementations of Denormalized Numbers", *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-1894-X, pp. 70–78, IEEE Computer Society, 2003. doi:10.1109/ARITH.2003.1207662

[B29] Stehlé, D., Lefèvre, V., and Zimmermann, P., "Searching Worst Cases of a One-Variable Function Using Lattice Reduction", *IEEE Transactions on Computers*, 54(3), pp. 340–346, 2005. doi:10.1109/TC.2005.55

[B30] The Unicode Standard, Version 5.0, The Unicode Consortium, Addison-Wesley Professional, 27 October 2006, ISBN 0-321-48091-0.

# Annex B

(informative)

# Program debugging support

## B.1 Overview

Implementations of this standard vary in the priorities they assign to characteristics like performance and debuggability (the ability to debug). This annex describes some programming environment features that should be provided by implementations that intend to support maximum debuggability. On some implementations, enabling some of these abilities might be very expensive in performance compared to fully optimized code.

Debugging includes finding the origins of and reasons for numerical sensitivity or exceptions, finding programming errors such as accessing uninitialized storage that are only manifested as incorrect numerical results, and testing candidate fixes for problems that are found.

## B.2 Numerical sensitivity

Debuggers should be able to alter the attributes governing handling of rounding or exceptions inside subprograms, even if the source code for those subprograms is not available; dynamic modes might be used for this purpose. For instance, changing the rounding direction or precision during execution might help identify subprograms that are unusually sensitive to rounding, whether due to ill-condition of the problem being solved, instability in the algorithm chosen, or an algorithm designed to work in only one rounding-direction attribute. The ultimate goal is to determine responsibility for numerical misbehavior, especially in separately-compiled subprograms. The chosen means to achieve this ultimate goal is to facilitate the production of small reproducible test cases that elicit unexpected behavior.

## B.3 Numerical exceptions

Debuggers should be able to detect, and pause the program being debugged, when a prespecified exception is signaled within a particular subprogram, or within specified subprograms that it calls. To avoid confusion, the pause should happen soon after the event which precipitated the pause. After such a pause, the debugger should be able to continue execution as if the exception had been handled by an alternate handler if specified, or otherwise by the default handler. The pause is associated with an exception and might not be associated with a well-defined source-code statement boundary; insisting on pauses that are precise with respect to the source code might well inhibit optimization.

Debuggers should be able to raise and lower status flags.

Debuggers should be able to examine all the status flags left standing at the end of a subprogram's or whole program's execution. These capabilities should be enhanced by implementing each status flag as a reference to a detailed record of its origin and history. By default, even a subprogram presumed to be debugged should at least insert a reference to its name in a status flag and in the payload of any new quiet NaN produced as a floating-point result of an invalid operation. These references indicate the origin of the exception or NaN.

Debuggers should be able to maintain tables of histories of quiet NaNs, using the NaN payload to index the tables.

Debuggers should be able to pause at every floating-point operation, without disrupting a program's logic for dealing with exceptions. Debuggers should display source code lines corresponding to machine instructions whenever possible.

For various purposes a signaling NaN could be used as a reference to a record containing a numerical value extended by an exception history, wider exponent, or wider significand. Consequently debuggers should be able to cause bitwise operations like **negate**, **abs**, and **copySign**, which are normally silent, to detect signaling NaNs. Furthermore the signaling attribute of signaling NaNs should be able to be enabled or

disabled globally or within a particular scope, without disrupting or being affected by a program's logic for default or alternate handling of other invalid operation exceptions.

## B.4 Programming errors

Debuggers should be able to define some or all NaNs as signaling NaNs that signal an exception every time they are used. In formats with superfluous bit patterns not generated by arithmetic, such as non-canonical significand fields in decimal formats, debuggers should be able to enable signaling-NaN behavior for data containing such bit patterns.

Debuggers should be able to set uninitialized storage and variables, such as heap and common space, to specific bit patterns such as all-zeros or all-ones which are helpful for finding inadvertent usages of such variables; those usages might prove refractory to static analysis if they involve multiple aliases to the same physical storage.

More helpful, and requiring correspondingly more software coordination to implement, are debugging environments in which all floating-point variables, including automatic variables each time they are allocated on a stack, are initialized to signaling NaNs that reference symbol table entries describing their origin in terms of the source program. Such initialization would be especially useful in an environment in which the debugger is able to pause execution when a prespecified exception is signaled or flag is raised.

# Annex C

(informative)

## List of Operations

# Consensus

## WE BUILD IT.

**Connect with us on:**

**Facebook:** https://www.facebook.com/ieeesa

**Twitter:** @ieeesa

**LinkedIn:** http://www.linkedin.com/groups/IEEESA-Official-IEEE-Standards-Association-1791118

**IEEE-SA Standards Insight blog:** http://standardsinsight.com

**YouTube:** IEEE-SA Channel

IEEE
standards.ieee.org
Phone: +1 732 981 0060 Fax: +1 732 562 1571