September 17, 2006

# DRAFT Standard for Floating-Point Arithmetic P754

## Draft 1.2.1.1

Modified at 23:14 on September 17, 2006

**Sponsor:**
Microprocessor Standards Committee

**Abstract:** This standard specifies interchange and non-interchange formats and methods for binary and decimal floating-point arithmetic in computer programming environments. Exception conditions are defined and default handling of these conditions is specified.

It is intended that an implementation of a floating-point system conforming to this standard can be realized entirely in software, entirely in hardware, or in any combination of software and hardware 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, sequence of operations, and destination formats, all under user control.

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

*Copyright © 2006 by the IEEE*
*Three Park Avenue*
*New York, New York 10016-5997, USA*
*All rights reserved.*
*This document is an unapproved draft of a proposed IEEE Standard. As such, this document is subject to change. USE AT YOUR OWN RISK! Because this is an unapproved draft, this document must not be utilized for any conformance/compliance purposes. Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of international standardization consideration. Prior to adoption of this document, in whole or in part, by another standards development organization permission must first be obtained from the Manager, Standards Intellectual Property, IEEE Standards Activities Department. Other entities seeking permission to reproduce this document, in whole or in part, must obtain permission from the Manager, Standards Intellectual Property, IEEE Standards Activities Department.*

*IEEE Standards Activities Department*
*Manager, Standards Intellectual Property*
*445 Hoes Lane*
*Piscataway, NJ 08854, USA*

## Patent statement

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 with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention. A patent holder or patent applicant has filed a statement of assurance that it will grant licenses under these rights without compensation or under reasonable rates and nondiscriminatory, reasonable terms and conditions to applicants desiring to obtain such licenses. The IEEE makes no representation as to the reasonableness of rates, terms, and conditions of the license agreements offered by patent holders or patent applicants. Further information may be obtained from the IEEE Standards Department.

## Change history

The following table shows the change history for this document.

| Version | Date | Author | Description |
|---|---|---|---|
| 0.992 | — | DH | Original content, passed from David Hough for styles&names changes. |
| 0.99223 | 07/03/06 | DVJ | Revised by David V James:<br>Applied MSC templates to existing text, including cross-references.<br>Included acronym clause and bibliography annex.<br>Changed to MSC/IEEE {shall, should, may, expected} definitions.<br>Provided PAR specified scope and purpose (required).<br>Applied names from previous motion (others in the same vein).<br>Reformatted indents for clarity.<br>Changed bullet lists to dash lists.<br>Changed numbered lists to: a) 1) i) ordering.<br>Uniform italics of *x*, *y*, and *z*.<br>Limited fonts to Arial, Times New Roman, Symbol, & Courier.<br>Eliminated historical annex (expected to be voted out). |
| 1.0 | 07/13/06 | DH | Results of style review 2006 July 13. |
| 1.1 | 07/19/06 | DH | Results of general meeting 2006 July 19:<br>18 Delete 5.3.2 Scaled-product operation, add A.3. Scaled-product operations<br>19 Define multiple-exception semantics<br>21 Clarify conversion to signed or unsigned integer format<br>22 Reduce recommended options for signaling NaN character sequences<br>20 Clarify conversion to unsigned integer format - plan A - After rounding |
| 1.1.1 | 07/27/06 | DH | Results of style review 2006 July 27. |
| 1.1.3 | 08/05/06 | STC | Results of style review 2006 August 3. |
| 1.1.4 | 08/12/06 | STC | Results of general meeting 2006 August 9, and email comments. |
| 1.1.5 | 08/15/06 | DH | Results of style review 2006 August 15. |
| 1.1.6 | 08/17/06 | DH | Results of style review 2006 August 17. |
| 1.2 | 09/12/06 | STC | Results of style review 2006 September 12 and general meeting 2006 September 7. |
| 1.2.1 | 09/17/06 | STC | Results of style review 2006 September 14. |


Your copyright notice would normally appear here.
This is an unapproved draft, subject to change.

## Introduction

[This introduction is not a part of DRAFT Standard for Floating-Point Arithmetic P754.]

This standard is a product of the Floating-Point Working Group of the Microprocessor Standards Subcommittee of the Standards Committee of the IEEE Computer Society. This work was sponsored by the Technical Committee on Microprocessors and Minicomputers.

PURPOSE: 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, sequence of operations, and destination formats, 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.
b) Enhance the capabilities and safety available to users and programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs. However, we recognize that utility and safety are sometimes antagonists.
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. When restricted to a declared subset of the standard, these programs should produce identical results on all conforming systems.
d) Provide direct support for
    1) Execution-time diagnosis of anomalies
    2) Smoother handling of exceptions
    3) Interval arithmetic at a reasonable cost
e) Provide for development of
    1) Standard elementary functions such as exp and cos
    2) Very high precision (multiword) arithmetic
    3) Coupling of numerical and symbolic algebraic computation
f) Enable rather than preclude further refinements and extensions.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## Participants

At the time this standard was completed, the working group had the following membership:

Dan Zuras, Chair

Aiken, Alex
Applegate, Matthew
Bailey, David
Bass, Steve
Bhandarkar, Dileep
Bhat, Mahesh
Bindel, David
Boldo, Sylvie
Canon, Stephen
Carlough, Steven
Cornea, Marius
Cowlishaw, Mike
Crawford, John
Darcy, Joe
Das Sarma, Debjit
Daumas, Marc
Davis, Bob
Davis, Mark
Delp, Dick
Demmel, Jim
Erle, Mark
Fahmy, Hossam
Fasano, J.P.
Fateman, Richard
Feng, Eric
Ferguson, Warren
Fit-Florea, Alex
Fournier, Laurent
Freitag, Chip
Godard, Ivan
Golliver, Roger
Gustafson, David
Hack, Michel
Harrison, John
Hauser, John
Hida, Yozo
Hinds, Chris
Hoare, Graydon
Hough, David
Huck, Jerry
Hull, Jim
Ingrassia, Michael
James, David
James, Rick
Kahan, William
Kapernick, John
Karpinski, Richard
Kidder, Jeff
Koev, Plamen
Li, Ren-Cang
Liu, Zhishun Alex
Mak, Raymond
Markstein, Peter
Matula, David
Melquiond, Guillaume
Mori, Nobuyoshi
Morin, Ricardo
Nedialkov, Ned
Nelson, Craig
Oberman, Stuart
Okada, Jon
Ollmann, Ian
Parks, Michael
Pittman, Tom
Postpischil, Eric
Riedy, Jason
Schwarz, Eric
Scott, David
Senzig, Don
Sharapov, Ilya
Shearer, Jim
Siu, Michael
Smith, Ron
Stevens, Chuck
Tang, Peter
Taylor, Pamela
Thomas, Jim
Thompson, Brandon
Thrash, Wendy
Toda, Neil
Trong, Son Dao
Tsai, Leonard
Tsen, Charles
Tydeman, Fred
Wang, Liang Kai
Westbrook, Scott
Winkler, Steve
Wood, Anthony
Yalcinalp, Umit
Zemke, Fred
Zimmermann, Paul
Zuras, Dan


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

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

| | | |
|---|---|---|
| To Be Supplied By IEEE | Etc. | Etc. |

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## Table of contents

# Table of contents




Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## List of figures



Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## List of tables




Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# DRAFT Standard for Floating-Point Arithmetic P754

## 1. Overview

### 1.1 Scope

This standard specifies interchange and non-interchange formats and methods for binary and decimal floating-point arithmetic in computer programming environments. Exception conditions are defined and default handling of these conditions is specified.

It is intended that an implementation of a floating-point system conforming to this standard can be realized entirely in software, entirely in hardware, or in any combination of software and hardware. It is the environment the user of the system sees that conforms or fails to conform to this standard. Hardware components that require software support to conform shall not be said to conform apart from such software.

### 1.2 Inclusions

This standard specifies:

— Formats for binary and decimal floating-point data for computation and data interchange.

— Addition, subtraction, multiplication, division, fusedMultiplyAdd, squareRoot, compare, and other operations.

— Conversions between integer and floating-point formats.

— Conversions between different floating-point formats.

— Conversions between floating-point ~~numbers~~ data in internal formats and external representations as character sequences.

— Floating-point exceptions and their handling, including nonnumbers (NaNs).

### 1.3 Exclusions

This standard does not specify:

— Formats of integers and external representations of numbers as character sequences.

— Interpretation of the sign and significand fields of NaNs.

### 1.4 Purpose

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, sequence of operations, and destination formats, all under user control.

### 1.5 Language-defined/implementation-defined

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 languages may choose to leave some behaviors to implementations to define.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

**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 will behave identically with respect to such language-defined behaviors. Languages that aspire toward reproducible results on all platforms are expected to specify more behaviors than languages that aspire toward maximum performance on all platforms.

Because this standard requires facilities that are not currently available in common programming languages, such languages might not be able to fully support this standard if they are no longer evolving themselves as standards. 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 specification 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 Annexes

The normative part of this standard is accompanied by several non-normative annexes:

— Annex B and Annex C contain recommendations for programming languages.

— Annex D, Annex E, and Annex G incorporate the working group's consensus on directions that future standard revisions should address. By providing these in preliminary form, the working group hopes that language designers, standards bodies, and implementers will develop and implement specifications that application programmers can exploit.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 2. References

The following referenced documents are indispensable for the application of this standard:

ANSI/IEEE Std 754–1985, IEEE Standard for Binary Floating-Point Arithmetic.[1]

ISO/IEC 9899, Second edition 1999-12-01, ~~Programing~~ Programming languages—C[2]

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[2]ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# 3. Terms and definitions

## 3.1 Conformance levels

Several keywords are used to differentiate between different levels of requirements and optionality, as follows:

**3.1.1 expected:** Describes the behavior of the hardware or software in the design models assumed by this specification. Other hardware and software design models may also be implemented.

**3.1.2 may:** Indicates a course of action permissible within the limits of the standard with no implied preference ("may" means "is permitted to").

**3.1.3 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").

**3.1.4 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").

## 3.2 Glossary of terms

**3.2.1 basic format:** One of the five sets of floating-point representations, three binary and two decimal, whose encodings are specified by this standard.

**3.2.2 biased exponent:** The sum of the exponent *e* and a constant (bias) chosen to make the biased exponent's range nonnegative.

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

**3.2.4 canonical encoding:** The preferred encoding of a floating-point representation in a format admitting more than one encoding for that representable value floating-point datum. Applied to declets, significands of finite numbers, infinities, and NaNs, especially in decimal formats.

**3.2.5 cohort:** In a given format, the set of floating-point representations of floating-point numbers with the same numerical value. +0 and −0 are in separate cohorts.

**3.2.6 computational operation:** An operation producing a floating-point result or capable of signaling a floating-point exception. Comparisons are computational operations.

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

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

**3.2.9 declet:** An encoding of three decimal digits into ten bits using the densely-packed decimal encoding scheme. Of the 1024 possible declets, 1000 canonical declets are produced by computational operations, while 24 non-canonical declets are not produced by computational operations, but are accepted in operands.

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

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

**3.2.12 exception:** An event that occurs when an operation has no outcome suitable for every reasonable application. That operation might signal one or more exceptions by invoking the default or, if explicitly

 Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

requested by the programmer, a language-defined alternate handling. Note that “event,” “exception,” and “signal” are defined in diverse ways in different programming environments.

**3.2.13 exponent:** The component of a finite floating-point representation ~~number~~ that signifies the integer power to which the radix is raised in determining the value of that floating-point representation ~~number~~. 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 significand length in digits.

**3.2.14 extended format:** A non-interchange format with wider precision and range that extends a supported basic format.

**3.2.15 external character sequence:** A representation of a number or NaN ~~floating-point datum~~ ~~number~~ as a sequence of characters, intended to be interpreted ~~more~~ readily by people ~~humans than by computers~~, including the character sequences in floating-point literals in program text.

**3.2.16 floating-point datum:** A floating-point number or nonnumber (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.

**3.2.17 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.

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

**3.2.19 format:** A set of representations of numerical values and symbols, perhaps accompanied by an encoding. ~~implemented in conformance with this standard.~~

**3.2.20 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.

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

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

**3.2.23 mode:** An implicit parameter to operations of this standard, which the user may set, test, save, and restore. The term mode may refer to the mode parameter (as in "rounding direction mode") or its value (as in "roundTowardZero mode").

**3.2.24 NaN:** Not a Number, a symbolic floating-point datum ~~symbolic entity symbol encoded in floating-point format.~~ There are two types of NaN representations: quiet and signaling. Most operations propagate **quiet NaN**s without signaling exceptions, and signal the invalid exception when given a **signaling NaN** operand. ~~Quiet NaNs propagate through most operations without signaling exceptions, while in most operations signaling NaNs signal the invalid operation exception when they appear as operands.~~

**3.2.25 narrower/wider format:** If the set of ~~numerical representable entities~~ 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.

**3.2.26 non-computational operation**: An operation producing no floating-point result and never signaling any floating-point exception.

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

**3.2.28 payload:** The diagnostic information contained in a NaN, encoded in part of its trailing significand field.

**3.2.29 prevailing mode:** The value of a mode governing a particular instance of execution of a computational operation of this standard. Languages specify how the prevailing mode is determined.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

**3.2.30 quantum:** The quantum of ~~the representation of a floating-point number~~ a finite floating-point representation is the value of a unit in the last position of its significand.

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

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

**3.2.33 result:** ~~The bit string (usually representing a floating-point datum)~~ The floating-point representation or encoding that is delivered to the destination.

**3.2.34 signal:** When an operation 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 by the programmer, a language-defined alternate handling.

**3.2.35 significand:** A component of a finite ~~an unencoded binary or decimal~~ 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.

**3.2.36 status flag:** A variable that may take two states, raised or lowered. When raised, a status flag may 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, divide-by-zero and invalid.

**3.2.37 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 cannot use the full precision available to normal numbers of the same format. Supersedes IEEE Std 754–1985's *denormalized number*.

**3.2.38 supported format:** A format provided in the programming environment and implemented in conformance with the requirements of this standard. Thus, a programming environment may provide more formats than it supports, as only those implemented in accordance with the standard are said to be supported.

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

**3.2.40 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.

**3.2.41 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.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# 4. Abbreviations and acronyms

This document contains the following abbreviations and acronyms:

NOTE DVJ: Consider listing here NaN, qNaN, sNaN, … .

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# 5. Formats

## 5.1 Overview: formats and conformance

This clause defines several kinds of standard floating-point formats, in two radices, 2 and 10. All the formats specified by this standard are fixed-width. The precision and range of a fixed-width format are determinable from the program text, and the corresponding encoding is usually defined so that all members have the same size in storage.

Formats defined by this standard are interchange or non-interchange:

- **interchange formats** are formats with encodings defined in this standard. They are widely available for storage and for data interchange among platforms. The format names used in this standard are not usually those used in programming environments. Interchange formats defined by this standard are basic or storage:
  - **basic formats** are interchange formats, available for arithmetic. This standard defines three basic binary floating-point formats in lengths of 32, 64, and 128 bits, and two basic decimal floating-point formats in lengths of 64 and 128 bits. A programming environment conforms to this standard, in a particular radix, by implementing one or more of the basic formats of that radix. The choice of standard formats is language-defined or, if the relevant language standard is silent or defers to the implementation, implementation-defined. A conforming implementation of a basic format shall:
    - provide means to initialize and store that format,
    - provide all the operations of this standard for that format,
    - provide conversions between that basic format and all other implemented standard formats.
  - **storage formats** are narrow interchange formats. This standard defines one binary storage floating-point format of 16 bits length, and one decimal storage floating-point format of 32 bits length. To support a storage format, this standard only requires that conversions be provided between that storage format and all other supported formats of the same radix. Languages permitting computation upon storage formats should perform such computations in wider formats.
- **non-interchange formats** are formats with no encodings defined in this standard. None are required by this standard. If implemented they are available for arithmetic, but they might not be suitable for interchanging data among platforms.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 5.2 Specification levels

Floating-point arithmetic is a systematic approximation of real arithmetic, as illustrated in Table 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 1—Relationships between different specification levels for a particular format**

| | | |
|---|---|---|
| Level 1 | **{–∞ … — 0 — … +∞}** | Extended real numbers. |
| many-to-one ↓ | rounding | ↑ one-to-many |
| Level 2 | **{–∞ … –0}** ∪ {+0 … +∞} ∪ NaN | Floating-point data—<br>an algebraically ~~completed~~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 6) maps an extended real number to a *~~representation of a~~ floating-point datum* included in that format. A ~~representable entity~~ floating-point datum, which can be a signed zero, finite non-zero number, signed infinity, or not-a-number, can be mapped to one or more *~~floating-point~~ 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 ~~representations~~ data to more than one bit string. Multiple NaN bit strings may be used to store retrospective diagnostic information (see 8.2).

## 5.3 Sets of floating-point data

This subclause specifies the sets of ~~entities~~ floating-point data representable within floating-point formats; the encodings for ~~those~~ representations of floating-point data in interchange formats are discussed in 5.4 and 5.5. 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 significant digits (precision)

— $emax$ = the maximum exponent

— $emin$ = the minimum exponent

Shall be either $1 - emax$ or $-emax$.
Should be $1 - emax$.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

The values of these parameters for each interchange format are given in Table 2; constraints on these parameters for extended formats are given in Table 7. Table 2 refers to interchange formats by the number of bits in their encoding. Within each format, the following entities floating-point data shall be provided:

— 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 $emin \leq e \leq emax$
  — $m$ is a number represented by a digit string of the form
    $d_0.d_1d_2\ldots d_{p-1}$ where $d_i$ is an integer digit $0 \leq d_i < b$ (therefore $0 \leq m < b$)
— Two infinities, $+\infty$ and $-\infty$
— Quiet and signaling NaNs

These are the only entities floating-point data provided. Binary interchange formats have just one representation each for +0 and −0, but decimal formats have many.

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: then 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 $emin \leq q+p-1 \leq emax$
  — $c$ is a number represented by a digit string of the form
    $d_0d_1d_2\ldots d_{p-1}$ where $d_i$ is an integer digit $0 \leq d_i < b$ ($c$ is therefore an integer with $0 \leq c < b^p$).

This view of the significand as an integer, $c$, with its corresponding exponent $q$, describes exactly the same set of zero and non-zero values non-zero floating-point numbers as the view in scientific form. (For non-zero 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 representable entities 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. Subnormal numbers are distinguished from normal numbers because of reduced precision and, in binary, because of different encoding methods. Every finite representable floating-point number is an integral multiple of the smallest subnormal magnitude $b^{emin} \times b^{1-p}$.

For any variable that has the value zero, the sign bit $s$ provides an extra bit of 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 8.3). In this standard, 0 and ∞ are written without a sign when the sign is not important.

**Table 2—Interchange format parameters defining floating-point numbers**

| | **Binary format (*b*=2)** | | | | **Decimal format (*b*=10)** | | |
|---|---|---|---|---|---|---|---|
| **parameter** | **binary16 storage** | **binary32 basic** | **binary64 basic** | **binary128 basic** | **decimal32 storage** | **decimal64 basic** | **decimal128 basic** |
| *p* digits | 11 | 24 | 53 | 113 | 7 | 16 | 34 |
| *emax* | +15 | +127 | +1023 | +16383 | +96 | +384 | +6144 |
| *emin* | -14 | -126 | -1022 | -16382 | -95 | -383 | -6143 |


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 5.4 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 5.1, the value of the significand $m$ is maximized by decreasing $e$ until either $e=emin$ or $m\geq 1$. After this normalization process is done, if $e=emin$ and $m<1$, the floating-point number is subnormal. Subnormal numbers (and zero) are encoded with a reserved biased exponent value.

~~Numbers~~ Floating-point data in the binary interchange formats are encoded in the following three fields ordered as shown in Figure 5.1:

a) 1-bit sign $S$

b) $w$-bit biased exponent $E=e+bias$

c) $(t=p-1)$-bit trailing significand digit string $T=d_1 d_2 \ldots d_{p-1}$; the leading bit of the logical significand, $d_0$, is implicitly encoded in the biased exponent $E$.

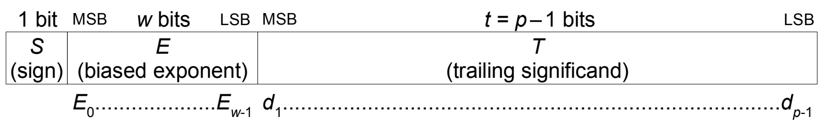


**Figure 5.1—Binary interchange floating-point format**

MSB is most significant bit; LSB is least significant bit. The values of $w$, $bias$, and $t$ for the binary interchange formats are listed in Table 3.

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 $\pm 0$ and subnormal numbers

— The reserved value $2^w-1$ to encode $\pm\infty$ and NaNs.

The ~~floating point~~ representation of the floating-point datum, $r$ and ~~representable entity~~ value of the floating-point datum represented, $v$ are inferred from the constituent fields thus:

a) If $E=2^w-1$ and $T\neq 0$, then $r$ is qNaN or sNaN and $v$ is NaN regardless of $S$.

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 ~~corresponding representable entity~~ 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 ~~corresponding representable entity~~ 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 8.3).

**Table 3—Binary interchange format encoding parameters**

| Format name | parameter | binary16 | binary32 | binary64 | binary128 |
|---|---|---|---|---|---|
| Storage width | — | 16 | 32 | 64 | 128 |
| Trailing significant width | $t$ | 10 | 23 | 52 | 112 |
| Biased exponent field width | $w$ | 5 | 8 | 11 | 15 |
| Bias | $E-e$ | 15 | 127 | 1023 | 16383 |

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 5.5 Decimal interchange format encodings

Unlike in a binary floating-point interchange format, in a decimal floating-point interchange format a ~~representable~~ floating-point number may have multiple representations. The set of ~~floating-point~~ 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 the maximum exponent value, then $(s, q, c)$ and $(s, q+1, c \div 10)$ 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 7.10). The cohorts of different floating-point ~~representations~~ numbers may 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 only has one representation. (An $n$-digit floating-point number may have fewer than $p-n+1$ members in its cohort if ~~the floating-point number~~ 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 operands also select a member of the result's cohort according to the specification in 7.11. Traditional decimal applications make use of the additional information cohorts convey.

~~Numbers~~ Representations of floating-point data in the decimal interchange formats are encoded in the following three fields, whose detailed layouts are described later.

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 ~~of the floating-point number~~ 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 $T$ which 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.

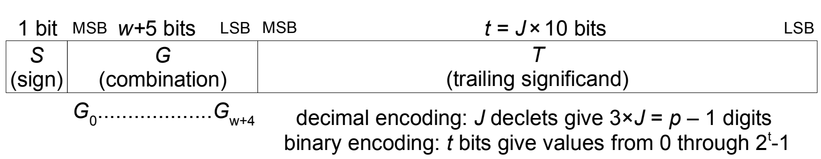


**Figure 5.2—Decimal interchange floating-point formats**

MSB is most significant bit; LSB is least significant bit.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

The values of $w$, $bias$, and $t$ for the decimal interchange formats are listed in Table 4:

**Table 4—Decimal interchange format encoding parameters**

| Format name | parameter | decimal32 | decimal64 | decimal128 |
|---|---|---|---|---|
| Storage width | — | 32 | 64 | 128 |
| Trailing significant width | $t$ | 20 | 50 | 110 |
| Combination field width | $w$+5 | 11 | 13 | 17 |
| Bias | $E$-$q$ | 101 | 398 | 6176 |

The ~~floating point~~ representation of the floating-point datum, $r$, and ~~representable entity~~ value of the floating-point datum represented, $v$, are inferred from the constituent fields, thus:

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 $T$ constitutes the NaN's payload, which can be used to distinguish various NaNs.

   The NaN 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 is ignored (it is treated as if it were zero). A NaN is in its preferred (canonical) representation 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 ~~infinity~~ representations 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 from the combination field $G$ and the trailing significand field $T$ and the biased exponent $E$ is encoded in the combination field. The encoding within these fields depends on whether the significand uses the decimal or the binary encoding.

   1) If the significand uses the *decimal* encoding, 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 first 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 first 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…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 first 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 5.5 and 5.6. Computational operations produce only the 1000 canonical declets, but also accept the 24 non-canonical declets in operands.

   2) Alternatively, if the significand uses the *binary* encoding, 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 $T$.

      In both cases i) and ii), the maximum value of the binary-encoded significand is the same as that of the equivalent 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

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

and the value used for *c* is zero. Computational operations produce only canonical significands, but also accept non-canonical significands in operations.

**Decoding densely-packed decimal:** Table 5 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 "don't care". Thus all 1024 possible 10-bit patterns shall be accepted and mapped into 1000 possible 3-digit combinations with some redundancy.

**Table 5—Decoding 10-bit densely-packed decimal to 3 decimal digits**

| $\mathbf{b}_{(6)}, \mathbf{b}_{(7)}, \mathbf{b}_{(8)}, \mathbf{b}_{(3)}, \mathbf{b}_{(4)}$ | $\mathbf{d}_{(1)}$ | $\mathbf{d}_{(2)}$ | $\mathbf{d}_{(3)}$ |
|---|---|---|---|
| 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 | $4b_{(0)} + 2b_{(1)} + b_{(2)}$ | $8 + b_{(5)}$ | $4b_{(7)} + 2b_{(8)} + b_{(9)}$ |
| 1 1 0 x x | $8 + b_{(2)}$ | $4b_{(3)} + 2b_{(4)} + b_{(5)}$ | $4b_{(7)} + 2b_{(8)} + b_{(9)}$ |
| 1 1 1 0 0 | $8 + b_{(2)}$ | $8 + b_{(5)}$ | $4b_{(7)} + 2b_{(8)} + b_{(9)}$ |
| 1 1 1 0 1 | $8 + b_{(2)}$ | $4b_{(3)} + 2b_{(4)} + b_{(5)}$ | $8 + b_{(9)}$ |
| 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 6 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 $\mathbf{b}_{(9)}$. Computational operations generate only the 1000 canonical 10-bit patterns defined by Table 6.

**Table 6—Encoding 3 decimal digits to 10-bit densely-packed decimal**

| $\mathbf{d}_{(1,0)}, \mathbf{d}_{(2,0)}, \mathbf{d}_{(3,0)}$ | $\mathbf{b}_{(0)}, \mathbf{b}_{(1)}, \mathbf{b}_{(2)}$ | $\mathbf{b}_{(3)}, \mathbf{b}_{(4)}, \mathbf{b}_{(5)}$ | $\mathbf{b}_{(6)}$ | $\mathbf{b}_{(7)}, \mathbf{b}_{(8)}, \mathbf{b}_{(9)}$ |
|---|---|---|---|---|
| 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 "don't care") are not generated in the result of a computational operation. However, as listed in Table 5, these 24 bit patterns do map to values in the range 0-999 ~~representations of valid decimal numbers~~. The bit pattern in a NaN significand can affect how the NaN is propagated (see 8.2).


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 5.6 Non-interchange formats

Like interchange formats, non-interchange formats are characterized by the parameters *b*, *p*, *emax*, and *emin*, and define representations for all ~~encompass all representations of~~ floating-point data (see 5.1). But unlike interchange formats, bit string encodings of noninterchange formats are not specified by this standard. Their encodings should be defined so that all members use the same amount of storage.

This standard does not require an implementation to provide any noninterchange format, but an implementation that does not support the widest basic format should support an *extended* non-interchange format that extends the widest basic format that is supported.

Table 7 specifies the minimum precision and exponent range of such extended formats:

**Table 7—Extended format parameters for floating-point numbers**

| | **Extended formats associated with:** | | |
|---|---|---|---|
| **Parameter** | **binary32** | **binary64** | **decimal64** |
| *p* digits ≥ | 32 | 64 | 20 |
| *emax* ≥ | 1023 | 16383 | 6144 |
| *emin* ≤ | −1022 | −16382 | −6143 |

**Note**—the minimum exponent range is that of the next wider basic format, while the minimum precision is intermediate between the widest supported basic format and the next wider basic format.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# 6. Modes and rounding

## 6.1 Mode specification

A mode is an implicit parameter to operations of this standard. All implementations shall provide the rounding direction modes (see 6.2) and should provide alternate exception handling modes (see Clause 9). With constant-mode specification, a user may specify a constant value for a mode parameter. With dynamic-mode specification, a user may specify that the mode parameter assumes the value of a dynamic mode variable. Modes in this standard may be supported with constant-mode specification or dynamic-mode specification, or both, as defined by the language. Mode specification is intended to be by means of translation directives, such as pragmas.

For constant-mode specification, the implementation provides language-defined means to specify a constant value for the mode parameter for all standard operations in a language-defined syntactic unit of the program. Whether and how external function calls are affected by a constant-mode specification for their immediately containing static scope is language defined.

For dynamic mode specification, the implementation provides language-defined means to specify that the mode parameter assumes the value of a dynamic mode variable for all standard operations in a language-defined syntactic unit of the program. The implementation initializes a dynamic mode variable to the default value for the 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 7.7.6 and 7.7.7.

In the absence of any explicit specification in the program, it is language-defined whether the mode parameter assumes the default mode value or the value of a dynamic mode variable.

The following aspects of dynamic mode variables are language (or implementation) defined:

— the effect of changing the value of the mode variable in an asynchronous event, such as in another thread or signal handler,

— whether the value of the mode variable can be determined by non-programmatic means, such as a debugger.

## 6.2 Rounding direction modes

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while perhaps signaling the inexact exception (see 9.6), underflow, or overflow. 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 modes in this clause.

The rounding direction mode affects all computational operations that might be inexact. Non-zero floating-point results always have the same sign as the unrounded result.

The rounding direction mode may affect the signs of zero sums (see 8.3), and does affect the thresholds beyond which overflow (see 9.4) and underflow (see 9.5) are signaled.

Implementations supporting both decimal and binary formats shall provide separate rounding direction modes for binary and decimal. Operations returning results in internal floating-point format use the rounding direction mode associated with the radix of the results. Operations converting from an operand in internal floating-point format to a result in integer format or external character sequence format use the rounding direction mode associated with the radix of the operand.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

### 6.2.1 Rounding direction modes to nearest

In these modes an infinitely precise result with magnitude at least $b^{emax}(b - \frac{1}{2} b^{1-p})$ shall round to $\infty$ with no change in sign; here *emax* and *p* are determined by the destination format (see Clause 5.0). With:

— roundTiesToEven, the ~~representable~~ floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest ~~representable~~ floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with an even least significant digit shall be delivered.

An implementation of this standard shall provide roundTiesToEven. It shall be the default rounding direction mode for results in binary formats. The default rounding direction mode for results in decimal formats is language-defined, but should be roundTiesToEven.

— roundTiesToAway, the ~~representable~~ floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest ~~representable~~ floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with larger magnitude shall be delivered.

A decimal implementation of this standard shall provide roundTiesToAway as a user-selectable rounding direction mode.

### 6.2.2 Directed rounding modes

An implementation shall also provide three other user-selectable rounding direction modes, the directed rounding modes roundTowardPositive, roundTowardNegative, and roundTowardZero. With:

— roundTowardPositive, the result shall be the format's ~~representable~~ floating-point number (possibly $+\infty$) closest to and no less than the infinitely precise result.

— roundTowardNegative, the result shall be the format's ~~representable~~ floating-point number (possibly $-\infty$) closest to and no greater than the infinitely precise result.

— roundTowardZero, the result shall be the format's ~~representable~~ floating-point number closest to and no greater in magnitude than the infinitely precise result.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# 7. Operations

## 7.1 Overview

All conforming implementations of this standard shall provide the operations listed in this chapter. Each of the computational operations specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's format (see Clause 6 and Clause 9). Clause 8 augments the following specifications to cover ±0, ±∞, and NaN; Clause 9 enumerates exceptions caused by exceptional operands and exceptional results.

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

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

— general-computational operations produce floating-point results, round all results according to Clause 6, and might signal the floating-point exceptions of Clause 9,

— 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 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 format of the operands.

Languages might permit other kinds of operations and combinations of operations into expressions. By their expression evaluation rules, languages specify when and how such operations and expressions are mapped into the operations of this standard.

In the operation descriptions that follow, operand 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.

*formatOf* indicates that the name of the operation specifies the floating-point destination *format*, which might be different from the floating-point operands' format. There are *formatOf* versions of these operations for every supported non-storage floating-point 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 called *int, intFormatOf, characterSequence,* or *conversionSpecification*. Languages with both signed and unsigned integer types should support both signed and unsigned *int* and *intFormatOf* operands and results.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 7.2 Decimal exponent calculation

As discussed in 5.3, a floating-point number may 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 ~~representable entity~~ value ~~*v*~~ of a floating-point result (and hence its cohort) is determined ~~only~~ by the operation and the operands' ~~representable entities~~ ~~floating-point~~ values; it is never dependent on the representation ~~of floating-point data~~ or ~~the~~ 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 certain computational operations, if the result is inexact, the cohort member of least possible exponent is used to get the longest possible significand; 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.

For other computational operations, whether or not the result is exact, the cohort member is selected based on the preferred exponent for a result of that operation.

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. Thus for finite $x$, depending on the representation of zero, $0+x$ might result in a different member of $x$'s cohort.

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

## 7.3 Homogeneous general-computational operations

### 7.3.1 General operations

Implementations shall provide the following homogeneous general-computational operations for all supported non-storage floating-point formats; they never propagate non-canonical results. Their destination format is indicated as *sourceFormat*:

— *sourceFormat* **roundToIntegralTiesToEven**(*source*)
*sourceFormat* **roundToIntegralTiesToAway**(*source*)
*sourceFormat* **roundToIntegralTowardZero**(*source*)
*sourceFormat* **roundToIntegralTowardPositive**(*source*)
*sourceFormat* **roundToIntegralTowardNegative**(*source*)
See 7.9. The preferred exponent is max(Q($x$), 0).

— *sourceFormat* **roundToIntegralExact**(*source*)
See 7.9. The preferred exponent is max(Q($x$), 0).

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

**nextUp**($x$) is the least ~~representable~~ 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 NaN, then the result is according to 8.2.

The preferred exponent is the least possible.

**nextDown**($x$) is –nextUp(–$x$).

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

— *sourceFormat* **nextAfter**(*source, source*)

**nextAfter**(*x*, *y*) is the next ~~representable neighbor of~~ floating-point number that neighbors *x* in the direction toward *y*, in the format of *x*:

— If either *x* or *y* is NaN, then the result is according to 8.2.

— If $x=y$, then **nextAfter**(*x,y*) is copySign(*x,y*).

— If $x<y$, then **nextAfter**(*x,y*) is nextUp(*x*); if $x>y$, then **nextAfter**(*x,y*) is nextDown(*x*). Overflow is signaled when *x* is finite but **nextAfter**(*x*, *y*) is infinite; underflow is signaled when **nextAfter**(*x*, *y*) lies strictly between $\pm b^{emin}$; in both cases, inexact is signaled.

The preferred exponent is Q(*x*).

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

When $y\neq 0$, the remainder $r=$**remainder**(*x*, *y*) is defined regardless of the rounding direction mode 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*.

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

— *sourceFormat* **minNum**(*source, source*)
*sourceFormat* **maxNum**(*source, source*)
*sourceFormat* **minNumMag**(*source, source*)
*sourceFormat* **maxNumMag**(*source, source*)

**minNum**(*x*,*y*) is *x* if $x<y$, *y* if $y<x$, the floating-point number if one operand is a floating-point number and the other a NaN. Otherwise it is either *x* or *y*.

**maxNum**(*x*,*y*) is *y* if $x<y$, *x* if $y<x$, the floating-point number if one operand is a floating-point number and the other a NaN. Otherwise it is either *x* or *y*.

**minNumMag**(*x*,*y*) is *x* if $|x|<|y|$, *y* if $|y|<|x|$, otherwise minNum(*x*,*y*).

**maxNumMag**(*x*,*y*) is *x* if $|x|>|y|$, *y* if $|y|>|x|$, otherwise maxNum(*x*,*y*).

The preferred exponent is Q(*x*) if *x* is ~~returned~~ the result, Q(*y*) if *y* is ~~returned~~ the result.

### 7.3.2 Decimal operation

Implementations supporting decimal formats shall provide the following homogeneous general-computational operation for all supported non-storage decimal floating-point formats. It never propagates non-canonical results. The destination format is indicated as *sourceFormat*:

— *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 which has the same numerical value as *x* and the same quantum as *y*. If the exponent is being increased, rounding according to the prevailing rounding direction mode might occur: the result is a different floating-point representation ~~number~~ and inexact 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, invalid is signaled and the result is NaN. If one or both operands are NaN the rules in 8.2 are followed. Otherwise if only one operand is infinite then invalid is signaled and the result is 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*).


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

### 7.3.3 *logBFormat* operations

Implementations shall provide the following general-computational operations for all supported non-storage floating-point formats. For each supported non-storage floating-point format, languages define an associated *logBFormat* to contain the integral values of logB($x$). The *logBFormat* might be a floating-point format or an integer format. The *logBFormat* shall 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, as well as scale factors produced by scaled-product operations (E.4).

If *logBFormat* is a floating-point format, then the following operations are homogeneous. If *logBFormat* is an integer format, then the first operand and the floating-point result of scaleB are of the same format.

— *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 when $x$ is positive and finite,

$1 \le \text{scaleB}(x, -\mathbf{logB}(x)) < b$.

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

The preferred exponent is 0.

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

**scaleB**($x$, $N$) is $x \times b^N$ for integral values $N$. The result ~~of **scaleB**~~ is computed as if the exact product were formed and then rounded to the destination format, subject to the prevailing rounding direction mode.

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

## 7.4 formatOf general-computational operations

### 7.4.1 Arithmetic operations

Implementations shall provide the following *formatOf* general-computational operations, for destinations of all supported non-storage floating-point formats, and, for each destination format, for operands of all supported non-storage floating-point formats with the same radix as the destination format. These operations never propagate non-canonical results.

— *formatOf*-**addition**(*source1, source2*)
*formatOf*-**subtraction**(*source1, source2*)
*formatOf*-**multiplication**(*source1, source2*)
*formatOf*-**division**(*source1, source2*)

For inexact decimal results, the preferred exponent is the least possible. For exact decimal results, the preferred exponent is min(Q(x), Q(y)) for addition and subtraction, Q(x) + Q(y) for multiplication, and Q(x) – Q(y) for the division x/y.

— *formatOf*-**squareRoot**(*source*)

The squareRoot operation is defined and has a positive sign for all operands ≥ 0, except that **squareRoot**(–0) shall be –0.

For inexact decimal ~~format~~ results, the preferred exponent is the least possible. For exact decimal ~~format~~ results, 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. No underflow, overflow, or inexact exception (Clause

Copyright © 2006 IEEE. All rights reserved.

This is an unapproved draft, subject to change.

7) can arise due to the multiplication, but only due to the addition; and so fusedMultiplyAdd differs from a multiplication operation followed by an addition operation.

For inexact decimal results, the preferred exponent is the least possible. For exact decimal results, the preferred exponent is min(Q($x$)+Q($y$), Q(z)).

— *formatOf*-**convert(***int*)

It shall be possible to convert from all supported signed and unsigned integer formats to all supported non-storage floating-point 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 default result is determined according to the prevailing rounding direction mode, and an inexact or floating-point overflow exception arises as specified in Clause 9, just as with arithmetic operations.

The preferred exponent is 0.

Implementations shall provide the following *intFormatOf* general-computational operations for destinations of all of a language-defined set of integer formats and for operands of all supported non-storage floating-point formats.

— *intFormatOf*-**convertToIntegerTiesToEven**(*source*)
*intFormatOf*-**convertToIntegerTowardZero**(*source*)
*intFormatOf*-**convertToIntegerTowardPositive**(*source*)
*intFormatOf*-**convertToIntegerTowardNegative**(*source*)
*intFormatOf*-**convertToIntegerTiesToAway**(*source*)
See 7.8 for details.

— *intFormatOf*-**convertToIntegerExactTiesToEven**(*source*)
*intFormatOf*-**convertToIntegerExactTowardZero**(*source*)
*intFormatOf*-**convertToIntegerExactTowardPositive**(*source*)
*intFormatOf*-**convertToIntegerExactTowardNegative**(*source*)
*intFormatOf*-**convertToIntegerExactTiesToAway**(*source*)
See 7.8 for details.

### 7.4.2 Conversion operations for all formats

Implementations shall provide the following *formatOf* conversion operations from all supported floating-point formats to all supported floating-point formats, including storage formats. Some format conversion operations produce results in a different radix than the operands.

— *formatOf*-**convert**(*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 6. 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 ~~format~~ results, the preferred exponent is the maximum possible.

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

— *formatOf*-**convertFromDecimalCharacter**(*decimalCharacterSequence*)
See 7.12.3. The preferred exponent is Q(*decimalCharacterSequence*).

— *decimalCharacterSequence* **convertToDecimalCharacter**(*source, conversionSpecification*)
See 7.12.3. The *conversionSpecification* specifies the precision and formatting of the *decimalCharacterSequence* result.

### 7.4.3 Conversion operations for binary formats

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


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

— *formatOf*-**convertFromHexCharacter**(*hexCharacterSequence*)
See 7.12.2.

— *hexCharacterSequence* **convertToHexCharacter**(*source, conversionSpecification*)
See 7.12.2. The *conversionSpecification* specifies the precision and formatting of the *hexCharacterSequence* result.

## 7.5 Homogeneous quiet-computational operations

### 7.5.1 Sign operations

Implementations shall provide the following homogeneous quiet-computational sign operations for all supported non-storage floating-point formats. They might propagate non-canonical encodings. They are performed as if on strings of bits, treating floating-point numbers and NaNs alike, and hence signal no exception.

~~The preferred exponent is Q(*x*).~~

— *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.

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

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

The preferred exponent is Q(*x*).

— *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 of *y*.

The preferred exponent is Q(*x*).

### 7.5.2 Decimal re-encoding operations

For each supported decimal format (if any), the implementation shall provide the following operations to convert between the internal decimal format and the two standard encodings for that format. These operations enable portable programs that are independent of the implementation's encoding for decimal types to access data represented with either standard encoding.

— *decimalEncodingType* **encodeDecimal**(*decimalType*):
encodes the value of the operand using decimal encoding

— *decimalType* **decodeDecimal**(*decimalEncodingType*):
decodes the decimal-encoded operand

— *binaryEncodingType* **encodeBinary**(*decimalType*):
encodes the value of the operand using the binary encoding

— *decimalType* **decodeBinary(***binaryEncodingType*):
decodes the binary-encoded operand

where *decimalEncodingType* is a language-defined type for storing decimal-encoded decimal floating-point data ~~numbers~~, *binaryEncodingType* is a language-defined type for storing binary-encoded decimal floating-point data ~~numbers~~, and *decimalType* is the type of the given decimal floating-point format.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 7.6 Signaling-computational operations

### 7.6.1 Comparisons

Implementations shall provide the following comparison operations, for all supported non-storage floating-point operands of the same radix:

— *boolean* **compareEqual**(*source1,source2*)
*boolean* **compareNotEqual**(*source1,source2*)
*boolean* **compareGreater**(*source1,source2*)
*boolean* **compareGreaterEqual**(*source1,source2*)
*boolean* **compareLess**(*source1,source2*)
*boolean* **compareLessEqual**(*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* **compareUnordered**(*source1,source2*)
*boolean* **compareQuietNotGreater(***source1,source2*)
*boolean* **compareQuietLessUnordered**(*source1,source2*)
*boolean* **compareQuietNotLess(***source1,source2*)
*boolean* **compareQuietGreaterUnordered**(*source1,source2*)
*boolean* **compareOrdered(***source1,source2*)

See 7.11 for details.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

### 7.6.2 Exception signaling-computational operations

This operation signals the exceptions specified by its operand, invoking either default or, if explicitly requested by the programmer, a language-defined alternate handling:

— *void* **signalException**(*exceptionGroupType*):
signals the exceptions specified in the *exceptionGroupType* operand, which can represent any subset of the exceptions.

Whether signalException additionally signals the inexact exception whenever it signals overflow or underflow is language defined. If signalException signals overflow and inexact or underflow and inexact, then it signals overflow or underflow before inexact. Otherwise, the order in which the exceptions are signaled is unspecified.

## 7.7 Non-computational operations

### 7.7.1 Conformance predicates

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

— *boolean* **is754**(*void*)

**is754**() is true if and only if this programming environment conforms to ANSI-IEEE Std 754-1985.

— *boolean* **is754R**(*void*)

**is754R**() is true if and only if this programming environment conforms to this standard.

### 7.7.2 General operations

Implementations shall provide the following non-computational operations for all supported non-storage floating-point formats. They are never exceptional, even for signaling NaNs.:

— *boolean* **isSigned**(*source*)

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

— *boolean* **isNormal**(*source*)

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

— *boolean* **isFinite**(*source*)

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

— *boolean* **isZero**(*source*)

**isZero**(*x*) is true if and only if $x = \pm 0$.

— *boolean* **isSubnormal**(*source*)

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

— *boolean* **isInfinity**(*source*)

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

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

— *int* **radix**(*source*)
**radix**(*x*) is the radix *b* of the format of *x*, 2 or 10.

— *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* **totalOrder**(*source, source*)
**totalOrder**(*x, y*) is defined in 7.10.

— *boolean* **totalOrderMag**(*source, source*)
**totalOrderMag**(*x, y*) is **totalOrder**(abs(*x*),abs(*y*)).

### 7.7.3 Decimal operation

Implementations supporting decimal formats shall provide the following non-computational operation for all supported non-storage decimal floating-point 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, i.e. Q(*x*) = Q(*y*), and false otherwise. sameQuantum(NaN, NaN) and sameQuantum(∞, ∞) are true; if exactly one operand is infinite or exactly one operand is NaN, sameQuantum is false. sameQuantum signals no exception.

### 7.7.4 Operations on subsets of flags

Implementations shall provide the following non-computational operations that act upon multiple status flags collectively:

— *void* **lowerFlag**(*exceptionGroupType*):
lowers (clears) the flags corresponding to the exceptions specified in the *exceptionGroupType* operand, which can represent any subset of the exceptions.

— *boolean* **testFlag**(*exceptionGroupType*):
queries whether any of the flags corresponding to the exceptions specified in the *exceptionGroupType* operand, which can represent any subset of the exceptions, are raised.

— *void* **restoreFlag**(*flagsType*, *exceptionGroupType*):
restores the flags corresponding to the exceptions specified in the *exceptionGroupType* operand, which can represent any subset of the exceptions, to their state represented in the *flagsType* operand (see saveFlags in 7.7.5).


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

### 7.7.5 Operations on all flags

Implementations shall provide the following non-computational operations that act upon all status flags collectively:

— *flagsType* **saveFlags**(*void*)

returns a representation of the state of all the flags. The return values of the **saveFlags** operation are for use as the first operand to the restoreFlag operation in the same program; this standard does not require support for any other use.

### 7.7.6 Operations on individual modes

Implementations shall provide the following non-computational operations for each supported MODE (see clause 6):

— *MODEtype* **getMODE**(*void*)
get prevailing value of MODE. Under constant specification for MODE, **getMODE** returns the constant value. Under dynamic specification for MODE, **getMODE** returns the current value of the dynamic MODE variable. Elsewhere, the return value is language defined (and may be unspecified).

For the rounding direction modes, the **getMODE** operations are:

— *binaryRoundingDirectionType* **getBinaryRoundingDirection**(*void*)

— *decimalRoundingDirectionType* **getDecimalRoundingDirection**(*void*)

With constant MODE specification, the value of the mode is set by the specification directive itself. Implementations supporting constant specification for MODE (as defined by the language) shall provide for constant specification of the default and each specific value of the mode.

Implementations supporting dynamic specification for MODE shall provide the following non-computational operation:

— *void* **setMODE**(*MODEtype*)
set the value of the dynamic mode variable. The operand may be any of the language-defined representations for the default and each specific value of MODE. The effect of this operation if used outside the static scope of a dynamic specification for MODE is language defined (and may be unspecified).

For the rounding direction modes, the **setMODE** operations are:

— *void* **setBinaryRoundingDirection**(*binaryRoundingDirectionType*)

— *void* **setDecimalRoundingDirection**(*decimalRoundingDirectionType*)

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

### 7.7.7 Operations on all modes with dynamic specification

Implementations supporting dynamic specification for modes shall provide the following non-computational operations for all dynamic-specifiable modes collectively:

— *modeGroupType* **saveModes(***void*)
save values of all dynamic-specifiable modes as a group

— *void* **restoreModes**(*modeGroupType*)
restore values of all dynamic-specifiable modes as a group

— *void* **defaultModes**(*void*)
set all dynamic-specifiable modes to default values

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.

The effect of these operations if used outside the scope of a dynamic specification for a dynamic-specifiable mode is language defined (and may be unspecified).

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

Implementations shall provide conversion operations from all supported non-storage floating-point formats to all supported [where is supported-provided-implemented explained?] signed and unsigned integer formats (Jim, I can't figure out how to say what you want said here without making this sentance both vacuous & tautalogical). Integral values are converted exactly from floating-point formats to integer formats whenever the value is representable in both formats.

Conversion to integer shall be effected by rounding as specified in Clause 6, but the rounding direction is indicated by the operation name.

When the rounded-to-integral floating-point value of the conversion operation's operand is not representable in the destination format because of overflow, and overflow cannot otherwise be indicated, the invalid exception shall be signaled.

When the rounded-to-integral floating-point value of the conversion operation's operand differs from its operand value, yet is representable in the destination format, the inexact exception might be signaled in certain circumstances:

The inexact exception should be signaled if an inexact conversion was implicitly invoked by a language's rules for conversions (including conversions during a copy) ~~for assignments~~ or expressions involving mixed types.

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

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

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

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

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

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

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


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

— *intFormatOf*-**convertToIntegerExactTowardPositive**(*x*)
rounds *x* to an integral value toward positive infinity.
— *intFormatOf*-**convertToIntegerExactTowardNegative**(*x*)
rounds *x* to an integral value toward negative infinity,
— *intFormatOf*-**convertToIntegerExactTiesToAway**(*x*)
rounds *x* to the nearest integral value, with halfway cases rounded away from zero.

## 7.9 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 6, 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 implied by the operation name and does not depend on a rounding direction mode. These operations do not signal any exception except for signaling NaN input.

— *sourceFormat* **roundToIntegralTiesToEven**(*x*)
rounds *x* to the nearest integral value, with halfway cases rounding to even.
— *sourceFormat* **roundToIntegralTiesToAway**(*x*)
rounds *x* to the nearest integral value, with halfway cases rounding away from zero.
— *sourceFormat* **roundToIntegralTowardZero**(*x*)
rounds *x* to an integral value toward zero.
— *sourceFormat* **roundToIntegralTowardPositive**(*x*)
rounds *x* to an integral value toward positive infinity.
— *sourceFormat* **roundToIntegralTowardNegative**(*x*)
rounds *x* to an integral value toward negative infinity.

For the following operation, the rounding direction is the prevailing rounding direction mode. This operation signals invalid for signaling NaN, and for a numerical operand, signals inexact if the result is not identical to the operand.

— *sourceFormat* **roundToIntegralExact**(*x*) rounds *x* to an integral value according to the prevailing rounding direction mode.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 7.10 Details of totalOrder predicate

For each supported non-storage floating-point format, an implementation shall provide certain predicates that define orderings among all operands in a particular format.

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 ~~entities~~ floating-point datum:

      i) if $x$ and $y$ have negative sign,
      totalOrder($x$, $y$) if and only if the exponent of $x \geq$ the exponent of $y$

      ii) otherwise
      totalOrder($x$, $y$) if and only if the exponent of $x \leq$ the exponent of $y$

   Note that 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 ~~representation~~ floating-point datum, as when one or both encodings are non-canonical.

d) if $x$ and $y$ are unordered numerically because $x$ or $y$ is NaN:

   1) totalOrder(−NaN, floating-point number) is true where −NaN represents a NaN with negative sign bit

   2) totalOrder(floating-point number, +NaN) is true where +NaN represents a NaN with positive sign bit

   3) if $x$ and $y$ are both NaNs, then totalOrder reflects a total ordering based on

      i) negative sign bit < positive sign bit

      ii) signaling < quiet for +NaN, reverse for −NaN

      iii) lesser payload < greater payload for +NaN, reverse for −NaN

Neither signaling nor quiet NaNs signal an exception.

For canonical $x$ and $y$, totalOrder($x,y$) and totalOrder($y,x$) are both true only if $x$ and $y$ are bitwise identical.

## 7.11 Details of comparison predicates

For every supported non-storage floating-point format, it shall be possible to compare ~~two numbers~~ one floating-point datum to another in that format. Additionally, floating-point ~~numbers~~ data represented in different formats shall be comparable as long as the operands' formats have the same radix.

Comparisons are exact and never overflow or underflow. Four mutually exclusive relations are possible: *less than*, *equal*, *greater than*, and *unordered*. The last case arises when at least one operand is 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*.

Languages define how the result of a comparison shall be delivered, in one of two ways: either as a condition code identifying one of the four relations listed above, or as a true-false response to a predicate that names the specific comparison desired.

Table 8, Table 9, and Table 10 exhibit twenty functionally distinct useful predicates and negations with various ad-hoc and traditional names and symbols. Each predicate is true if any of the its indicated condition codes is true. The condition code "?" indicates an *unordered* relation. Table 9 lists four unordered-signaling predicates and their negations that cause an invalid operation exception when the relation is unordered. That invalid exception defends against unexpected quiet NaNs arising in programs written using the standard predicates {<, ≤, ≥, >} and their negations, without considering the possibility of a quiet NaN operand. Programs that explicitly take account of the possibility of quiet NaN operands may use the unordered-quiet predicates in Table 10 which do not signal such an invalid exception.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

Note that predicates come in pairs, each a logical negation of the other; applying a prefix such as NOT to negate a predicate in Table 8, Table 9, and Table 10 reverses the true/false sense of its associated entries, but does not change whether *unordered* relations cause an invalid operation exception.

The unordered-quiet predicates in Table 8, intended for use by all programs, do not signal an exception on quiet NaN operands:

### Table 8—Required unordered-quiet predicate and negation

| **Unordered-quiet predicate** | | **Unordered-quiet negation** | |
|---|---|---|---|
| **True relations** | **Names** | **True relations** | **Names** |
| EQ | compareEqual<br>= | LT GT UN | compareNotEqual<br>?<><br>NOT(=)<br>≠ |

The unordered-signaling predicates in Table 9, intended for use by all programs *not* written to take into account the possibility of NaN operands, signal an invalid exception on quiet NaN operands:

### Table 9—Required unordered-signaling predicates and negations

| **Unordered-signaling predicate** | | **Unordered-signaling negation** | |
|---|---|---|---|
| **True relations** | **Names** | **True relations** | **Names** |
| GT | compareGreater<br>> | EQ LT UN | compareSignalingNotGreater<br>NOT(>) |
| GT EQ | compareGreaterEqual<br>>=<br>≥ | LT UN | compareSignalingLessUnordered<br>NOT(>=) |
| LT | compareLess<br>< | EQ GT UN | compareSignalingNotLless<br>NOT(<) |
| LT EQ | compareLessEqual<br><=<br>≤ | GT UN | compareSignalingGreaterUnordered<br>NOT(<=) |

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

The unordered-quiet predicates in Table 10, intended for use by all programs written to take into account the possibility of NaN operands, do not signal an exception on quiet NaN operands:

**Table 10—Required unordered-quiet predicates and negations**

| Unordered-quiet predicate | | Unordered-quiet negation | |
|---|---|---|---|
| **True relations** | **Names** | **True relations** | **Names** |
| GT | compareQuietGreater<br>!<=<br>isGreater | EQ LT UN | compareQuietNotGreater<br>?<=<br>NOT(!<=) |
| GT EQ | compareQuietGreaterEqual<br>!<<br>isGreaterEqual | LT UN | compareQuietLessUnordered<br>?<<br>NOT(!<) |
| LT | compareQuietLess<br>!>=<br>isLess | EQ GT UN | compareQuietNotLess<br>?>=<br>NOT(!>=) |
| LT EQ | compareQuietLessEqual<br>!><br>isLessEqual | GT UN | compareQuietGreaterUnordered<br>?><br>NOT(!>) |
| UN | compareUnordered<br>?<br>isUnordered | LT EQ GT | compareOrdered<br><=><br>NOT(?) |

There are two ways to write the logical negation of a predicate, one using NOT explicitly and the other reversing the relational operator. Thus in programs written without considering the possibility of a NaN operand, the logical negation of the unordered-signaling predicate (X < Y) is just the unordered-signaling predicate NOT(X < Y); the unordered-quiet reversed predicate (X ?>= Y) is different in that it does not signal an invalid operation exception when X and Y are *unordered*. In contrast, the logical negation of (X = Y) may be written either NOT(X = Y) or (X ?<> Y); in this case both expressions are functionally equivalent to (X != Y).

## 7.12 Details of conversion between internal floating-point and external character sequences

This Clause specifies conversions between internal formats and external character sequence formats. Conversions between internal formats of different radices are correctly rounded and set exceptions correctly as described in 7.4.2.

Implementations shall provide conversions from each supported internal format to an external decimal character sequence, exact for decimal and using roundTiesToEven for binary, with sufficient information that the external character sequence can be converted back to the same internal format and recover the representation of the original floating-point number unchanged.

Implementations shall support conversions between all supported binary internal formats and one or more external character sequence formats representing numbers with hexadecimal digits. Implementations shall support at least one conversion specification that converts all floating-point numbers in all supported binary internal formats to external hexadecimal character sequences, with sufficient information that the external character sequence can be converted back to the same internal format and recover the representation of the original floating-point number unchanged.~~ sufficiently precisely to represent the binary internal binary floating-point number exactly.~~

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 internal 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 provide means, to permit constants to be translated at execution time with the modes in effect at execution time and exceptions generated at execution time.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

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

Any external character sequence created on output to represent a zero, infinity, or NaN, shall represent a zero, infinity, or NaN on input as well. Some character sequence formatting specifications reproduce internal ~~format~~ floating-point numbers exactly, in roundTiesToEven mode, when those numbers are converted to character sequences and then those sequences are converted back to internal ~~format~~ floating-point numbers. For those specifications, zeros, infinities, and NaNs are reproduced exactly as well. Signs of zeros and infinities are preserved.

Issues of character codes (ASCII, Unicode, etc.) are language-defined. The representation of infinities, NaNs, and zeros by external character sequences is, in part, language defined. Representations of infinities and NaNs should be the same for hexadecimal and decimal character sequences.

Conversion of an infinity in internal 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 an internal floating-point format shall produce an infinity (with the same sign as the input).

Conversion of a quiet NaN in internal 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.

Conversion of a signaling NaN in internal 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 exception should be signaled.

Conversion of external character sequences “nan”, regardless of case, with an optional preceding sign, to an internal floating-point format shall produce a quiet NaN.

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

Languages should provide an optional conversion of NaNs in internal format to external character sequences that appends to the basic NaN character sequences a suffix that can represent the NaN payload (see 8.2). The form and interpretation of the payload suffix is language defined. The language should require that any such optional output sequences be recognized as input in conversion of external character sequences to internal formats.

### 7.12.2 External hexadecimal character sequences representing finite numbers

Implementations supporting binary formats shall provide conversions between all interchange and non-interchange binary formats and ~~an~~ external hexadecimal character ~~sequence format~~ sequences. External hexadecimal character sequences for finite numbers are of the form specified by C99 subclauses:

6.4.4.2 floating constants,
20.1.3 strtod,
7.19.6.2 fscanf (a, e, f, g), and
7.19.6.1 fprintf (a, A).

The “0x” may be omitted in contexts where the only character sequence data is hexadecimal. When converting to hexadecimal 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 character sequences with an explicit precision specification, and conversions from hexadecimal character sequences to internal binary formats, are correctly rounded according to the prevailing binary rounding direction mode.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

### 7.12.3 External decimal character sequences representing finite numbers

~~Conversion parameters ***m*** and ***n*** are specified below according to the widest internal format supported in a radix. For each supported radix, an implementation shall define integer $\mu \geq (\boldsymbol{m}+3)$ and integer $\eta \geq \boldsymbol{n}$, and shall provide conversions between all interchange and non-interchange formats and at least one external character sequence format that represents all decimal floating-point numbers of the form $M \times 10^N$ where integers $M$ and $N$ satisfy $|M| \leq 10^\mu - 1$ and $|N + \mu - m - 3| \leq \eta$.~~

The conversion parameter $m$ is specified below according to the widest internal format supported in a radix. For each supported radix, and implementation shall define an integer $\mu \geq (m+3)$ and an integer $\eta$ (further specified below) and shall provide conversion between all interchange and non-interchange formats in the radix and at least one external character sequence format that represents all decimal numbers with up to $\mu$ significant digits and with exponents (of 10) in the range $[-\eta, \eta]$.

In internal to decimal-string conversions, if more than $\mu$ result digits are requested, the input shall be converted with correct rounding to $\mu$ digits, and extra digits shall be generated as zeros.

If more than $\mu$ digits are given for decimal-string to internal conversions, the result of the conversion shall be as if it were carried out in two steps: First round the given decimal number to $\mu$ decimal digits, and then convert the resulting $\mu$-digit number to the target floating-point format, in both cases rounding correctly according to the prevailing rounding mode.

**Table 11—Decimal conversion parameter~~s~~ when widest supported format is basic**

| Widest basic format | *m* for binary formats | *m* for decimal formats | ~~*n* for either radix~~ |
|---|---|---|---|
| 32-bit | 9 | — | ~~99~~ |
| 64-bit | 17 | 16 | ~~999~~ |
| 128-bit | 36 | 34 | ~~999~~ |

Table 11 specifies the parameters when the widest supported format in a particular radix is basic. When the widest implemented format is not basic:

— If the widest implemented binary format can encompass $p$ significant bits, then
$m$ is $1+\text{ceiling}(p \times \log_{10}(2))$ ~~and~~
~~***n*** is $10^{\text{ceiling}(\log_{10}(\log_{10}(2) \times emax))} - 1$. [Hack: formula wrong?]~~

— If the widest implemented decimal format can encompass $p$ significant digits, then
$m$ is $p$ ~~and~~
~~***m*** is $10^{\text{ceiling}(\log_{10}(emax))} - 1$. [Hack: formula wrong?]~~

$\eta$ shall be sufficiently large to represent the result of converting any number in internal format to the external decimal character sequence format with up to $\mu$ significant digits.

Implementations should provide other decimal character sequence formats as well. All conversions to and from decimal character sequence formats, within the conversion parameter limits above, are correctly rounded according to the prevailing rounding direction mode. For conversions between binary formats and decimal character sequences, the inexact exception shall be signaled correctly for conversions of no more than $\mu$ digits.

For internal to decimal-string conversions, the inexact bit shall be set correctly.

For decimal-string to internal conversions, if more than $\mu$ digits were given, and any of those extra digits were non-zero, the inexact bit shall be set.

The table entries for $m$ are the number of significant digits to be produced when converting internal binary to decimal character sequences, when no precision is specified by the program or the language.

As a consequence of the foregoing, conversions shall be monotonic: increasing the value of an internal floating-point number shall not decrease its value after conversion to an external character sequence, and increasing the value of a external character sequence shall not decrease its value after conversion to an internal floating-point number.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

When the destination is an external ~~representation~~ character sequence, language specifications locate its least significant digit for purposes of rounding. The result format's values are the ~~decimal~~ numbers representable within that language specification. The number of significant digits is determined by that specification, and in the case of fixed-point conversion by the source value as well.

If external to internal conversion over/underflows, the response is as specified in Clause 9. Over/underflow encountered during internal to external conversion should be indicated to the user by appropriate character sequences.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# 8. Infinity, NaNs, and sign bit

## 8.1 Infinity arithmetic

Infinity arithmetic shall be construed as the limiting case 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 with infinite operands or results are usually exact and signal no exceptions, except when

— ∞ is an invalid operand (see 9.2),
— ∞ is created from finite operands by overflow (see 9.4) or division by zero (see 9.3),
— remainder(subnormal, ∞) signals underflow,
— nextAfter(x, ∞) signals underflow and inexact if the result would be subnormal,
— nextAfter(max normal, ∞) signals overflow and inexact if the result would be infinite.

## 8.2 Operations with NaNs

Two different kinds of NaN, signaling and quiet, shall be supported in all operations. Signaling NaNs afford representations for uninitialized variables and arithmetic-like enhancements (such as complex-affine infinities or extremely wide range) that are not the subject of the 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 computational operations.

Signaling NaNs shall be reserved operands that signal the invalid operation exception (see 9.1) for every general-computational and signaling-computational operation.

Under default exception handling, any operation signaling an invalid exception for which a floating-point result is to be delivered, shall deliver a quiet NaN.

Every general-computational and quiet-computational operation involving one or more input NaNs, none of them signaling, shall signal no exception, except fusedMultiplyAdd (see 9.2) . For an operation with quiet NaN inputs other than max and min operations, if a floating-point result is to be delivered, the result shall be a quiet NaN, which should be one of the input NaNs. If the trailing significand field of a decimal input NaN is canonical then the bit pattern of that field shall be preserved if that NaN is chosen as the result NaN. Note that format conversions, including conversions between internal 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 7.4, 7.6, and 9.1.

### 8.2.1 ~~Binary encodings of~~ NaN encodings in binary formats

This clause further specifies the encodings of NaNs as bit strings when they are the results of operations. When encoded, all NaNs have a sign bit and a pattern of bits necessary to identify the encoding as a NaN and which determines its kind (sNaN vs. qNaN). The remaining bits, which are in the trailing field, encode the payload, which might be diagnostic information (see 8.2).

All binary NaN bitstrings have all the bits of the biased exponent field E set to 1 (see 5.4). A quiet NaN bitstring should be encoded with the first bit ($d_1$) of the trailing significand field T being 1. A signaling NaN bitstring should be encoded with the first bit of the trailing significand field being 0. Some other bit of the trailing significand field must not be zero ~~be non-zero~~ to distinguish the NaN from infinity.

In the preferred encoding, a signaling NaN should be quieted by setting $d_1$ to 1, leaving the remaining bits of T unchanged.

For binary formats, the payload is the $p-2$ least significant bits of the trailing significand field.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

### 8.2.2 NaN encodings in decimal 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 unchanged (see 5.5).

Any computational operation which produces, propagates, or quiets a decimal format NaN shall set the bits $G_6$ through $G_{w+4}$ of $G$ to 0, and shall generate only a canonical trailing significand field.

For decimal formats, the payload is the trailing significand field.

### 8.2.3 NaN propagation

An operation which propagates NaNs and has a single NaN as an input should produce a NaN with the payload of the input NaN.

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. This standard does not specify which of the input NaNs will provide the payload.

Invalid operations, and conversions of a quiet NaN to a floating-point format of the same or a different radix, should return a quiet NaN which should provide some language-defined diagnostic information.

Furthermore, a conversion of a canonical 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 encoding in any way.

There should be means to read and write NaN payloads from and to external character sequences (see 7.12.1).

## 8.3 The sign bit

When either an input or result is NaN, this standard does not interpret the sign of a NaN. Note however that operations on bitstrings – copy, negate, abs, copySign – specify the sign bit of a NaN result, sometimes based upon the sign bit of a NaN operand. The logical predicate totalOrder is 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 roundToIntegral and roundToIntegralExact (see 7.3.1) is the sign of the 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 + in all rounding direction modes except roundTowardNegative; in that mode, the sign of an exact zero sum (or difference) shall be –. However, $x+x = x–(–x)$ retains the same sign as $x$ even when $x$ is zero.

When $(a\times b)+c$ would vanish in exact arithmetic, the sign of fusedMultiplyAdd($a$,$b$,$c$) shall be determined by the rules above for a sum of operands.

Except that squareRoot(–0) shall be –0, every valid squareRoot shall have a positive sign.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# 9. Default exception handling

## 9.1 Overview: exceptions and flags

There are five types of exceptions that shall be signaled. This clause specifies default nonstop exception handling, which usually entails raising a status flag, delivering a default result, and continuing execution. A language might define modes for alternate exception handling and means for programmers to invoke them.

For each type of exception the implementation shall provide a status flag that shall be raised when the corresponding exception is signaled. It shall be lowered only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time. (See 7.7.4 and 7.7.5)

A program that does not inherit status flags from another source, begins execution with all status flags lowered.

Languages should specify defaults in the absence of any explicit program 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 flags have scope greater than within an invoked function, whether and when an asynchronous event, such as a raising or lowering it in another thread or signal handler, affects the flag tested within that invoked function

— whether a flag's state can be determined by non-programmatic means (such as a debugger) within that invoked function

— whether flags raised in invoked subfunctions set flags in invoking functions,

— whether flags raised in invoking functions set flags in invoked subfunctions.

— whether to allow, and if so the means, to declare flags to 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

  — ~~the flags standing at the beginning of execution of an invoked subfunction are the flags that were standing in the invoking function at the time the subfunction was invoked~~

  — on return from or termination of an invoked subfunction, the flags standing in an invoking function are the flags that were standing in the subfunction at the time of return or termination

  — ~~when a function terminates other than by returning to its immediate invoking function, the flags standing will be those standing at the time of the function termination~~

An invocation of the signal-exception operation of 7.6.2, may signal any combination of exceptions. For an invocation of any other operation ~~specified~~ required by this standard, at most ~~only~~ two exceptions might be signaled, in just these combinations: overflow followed by inexact, and underflow followed by inexact.

The inexact exception is signaled if the overflow exception receives default handling, and might be signaled if the underflow exception receives default handling (see 9.5).

In general, when an operation signals more than one exception, none of which have alternate exception handling enabled, each signaled exception will receive its default handling.

When an operation signals more than one exception, some or all of which have alternate exception handling enabled, alternate exception handling will be invoked for the most important exception, and languages define whether other signaled exceptions receive default handling, alternate handling, or are ignored. Exceptions are listed in this clause in order of decreasing importance (invalid most important, inexact least important).

For the computational operations defined in this standard, exceptions are defined below to be signaled if and only if certain conditions arise. That is not meant to imply whether those exceptions are signaled by operations not specified by this standard such as complex arithmetic or elementary transcendental functions. Those and other operations, not specified by this standard, should signal those exceptions according to the definitions below for standard operations, but that may not always be economical. Standard exceptions for nonstandard functions are language-defined.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## 9.2 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 invalid exception operation shall be a quiet NaN (see 8.2). The invalid exception operations in this standard are:

a) any general-computational or signaling-computational operation on a signaling NaN (see 8.2);

b) multiplication: ~~0 × ∞ or ∞ × 0;~~ 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(∞,∞); ~~0/0 or ∞/∞;~~

f) remainder: remainder(*x*,*y*), ~~*x* REM *y*~~, where *y* is zero or *x* is infinite and neither is 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 invalid exception operations are:

~~i) conversion of an internal floating-point number to an integer (see 7.8) or external representation as a character sequence (see 7.12.1) when overflow, infinity, or NaN precludes a correctly-rounded representation in the destination and this cannot otherwise be indicated;~~

j) conversion of an internal floating-point number to an ~~unsigned~~ integer format, when the source is NaN, infinity, or a value which would convert to an integer ~~less than zero~~ outside the range of the result format under the prevailing rounding mode. ~~to an unsigned integer format and this cannot otherwise be indicated;~~

k) comparison by way of unordered-signaling predicates listed in Table 9, when the operands are *unordered*;

l) when *logBFormat* is an integer format, then logB(NaN), logB(∞), and logB(0) (see 7.3.3) .

## 9.3 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. In particular, the division by zero exception shall be signaled if the divisor is zero and the dividend is a finite non-zero number. The default result shall be a correctly signed ∞ (see 8.3).

When *logBFormat* is a floating-point format, logB(0) is −∞ and signals the division by zero exception.

## 9.4 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 (Clause 6) were the exponent range unbounded. The default result shall be determined by the rounding direction mode and the sign of the intermediate result as follows:

a) roundTiesToEven and roundTiesToAway carries
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 −∞

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

d) roundTowardPositive carries
negative overflows to the format's most negative finite number, and carries
positive overflows to $+\infty$

However nextAfter($x$,y) signals overflow and inexact if and only if nextAfter is infinite and differs from the finite number $x$.

## 9.5 Underflow

The underflow exception is signaled when a tiny non-zero result would be created strictly between $\pm b^{emin}$ which, because it is so tiny, may cause some other exception later such as overflow upon division. The implementer may choose how tininess is detected, but shall detect tininess in the same way for all operations of a given radix (in the case of a conversion operation, the radix from which the rounding mode is taken). Tininess may be detected either

a) *After rounding* - when a non-zero result computed as though the exponent range were unbounded would lie strictly between $\pm b^{emin}$
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 method for detecting tininess does not affect the default rounded result delivered which might be zero, subnormal, or $\pm b^{emin}$.

Loss of accuracy shall be detected as an inexact result - when the delivered result differs from what would have been computed were both exponent range and precision unbounded. (This is the condition called inexact in 9.6).

The default exception handling for underflow is to deliver a rounded result, raise the underflow flag, and signal the inexact exception, if and only if both tininess and loss of accuracy have been detected; if no loss of accuracy occurs, no flag is raised.

However nextAfter($x$,$y$) signals underflow and inexact if and only if ~~nextAfter~~ the result is strictly between $\pm b^{emin}$ and differs from $x$.

## 9.6 Inexact

If the rounded result of an operation is not exact or if it overflows with default handling then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination.

nextAfter($x$,$y$) signals inexact if and only if nextAfter also signals overflow or underflow.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# Annexes

# Annex A
# (informative)
# Bibliography

The following documents may be helpful to the reader:

United States Patent 6,437,715, Cowlishaw, August 20, 2002: Decimal to binary coder/decoder.

Densely-Packed Decimal Encoding, Michael F. Cowlishaw, IEE Proceedings - Computers and Digital Techniques, Vol. 149 #3, ISSN 1350-2387, pp102-104, IEE, London, May 2002.

Decimal Floating-Point: Algorism for Computers, Michael F. Cowlishaw, Proceedings of the 16th IEEE Symposium on Computer Arithmetic, ISBN 0-7695-1894-X, pp104-111, IEEE, June 2003.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# Annex B
# (informative)
# Expression evaluation

## B.1 Overview

~~The operations specified previously in Clause 7 are rounded to a destination format of Clause 5, according to a rounding direction method of Clause 6, and raise exceptions according to Clause 9. This Annex pertains to languages for which every variable and constant is typed and therefore every floating-point variable has one of the formats of this standard.~~

Every operation has an implicit or explicit destination. When a variable is a final destination, as in conversion to a variable, the format of that variable governs its rounding. The format of an anonymous destination is defined by language expression evaluation rules.

Some languages implicitly convert operands of standard floating-point operations to a common format. Typically, operands are promoted to the widest format of the operands or a Widento format (see Annex C). However, if the common format is not a superset of the operand formats, then the conversion 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 less range or precision

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

**Widento methods**

Annex C prescribes Widento methods for widening operations in expressions. Widening, which should be available in every implementation supporting more than one floating-point format in a radix, is performed as specified by the user, and thus is not an optimization in the usual sense. Widening occurs before optimization is considered.

**Reproducible results**

Languages should provide means for programmers to specify reproducible results—~~identical~~ results that are identical on all platforms supporting that language and this standard, for operations completely specified by this standard.

## B.2 Optimization

As part of support for this standard, a language should require that execution behavior preserve the literal meaning of the source code and not change the numerical results or exceptions signaled. However, the language should define, and require implementations to provide, means to allow or disallow the following optimizations, separately and collectively, for a language-defined syntactic unit of the program:

— 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 reassociation and wider intermediates to evaluate a sum reduction
— use of reassociation and wider intermediates to evaluate a product reduction


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## B.3 Assignments

Assignment of an expression to a variable should be implemented by further rounding the result value of the assigned expression to the width of the assigned-to variable. Implementations should never use an assigned-to variable's wider precursor in place of the assigned-to variable's stored value when evaluating subsequent expressions.

Actual parameters to non-generic function calls are like assignments, and are rounded to the type of the formal parameter if a declaration is in scope, and are rounded to a language-defined type otherwise. Languages define rules for actual parameters to generic functions.

Values to be returned by functions of declared types are like assignments and should be rounded to the declared type of the function. Languages define rules for types of generic function return values according to the function parameters.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# Annex C
# (informative)
# Widento methods for expression evaluation

In this standard, a computational operation 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-specified generic operations, that destination format is implied by the widths of the operands and by the **Widento method** currently in effect.

An implementation should provide a Widento method for each supported non-storage format.

The following Widento methods 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.

— noWidento method: A language should define, and require implementations to provide, means for users to specify a noWidento method, for a language defined syntactic unit of the program. 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, unless that format is a storage format; then the result should be rounded to the narrowest supported basic format.

— widentoFormat methods: A language that provides addition, subtraction, multiplication, division, and comparison as generic operators should define, and require implementations to provide, means for users to specify a WidentoFormat method for each supported format, except storage formats, for a language defined syntactic unit of the program. widentoFormat methods affect the aforementioned operators. Whether and which other generic operators or functions they affect is language defined. Table C.1 lists operators that are suitable for being affected by Widento methods. Destination width is the maximum of the width of the widentoFormat 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 widentoFormat. Affected operations (including comparisons) do not narrow their operands, which may be widened expressions. widentoFormat affects only expressions in the radix of format.

Widento methods do not affect the width of the final rounding to an explicit destination, which is always rounded to the declared format of that destination.

Widento methods do not affect explicit format conversions within expressions; they are always rounded to the format specified by the conversion.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

**Table C.1—Widento operations**

| Operation |
|---|
| *destination* **addition**(*source1, source2*)<br>*destination* **subtraction**(*source1, source2*)<br>*destination* **multiplication**(*source1, source2*)<br>*destination* **division**(*source1, source2*) |
| *destination* **squareRoot** (*source1*) |
| *destination* **fusedMultiplyAdd** (*source1, source2, source3*) |
| *destination* **minNum**(*source1, source2*)<br>*destination* **maxNum**(*source1, source2*)<br>*destination* **minNumMag**(*source1, source2*)<br>*destination* **maxNumMag**(*source1, source2*) |
| *boolean* **compareEqual***(source1,source2)*<br>*boolean* **compareNotEqual***(source1,source2)*<br>*boolean* **compareGreater***(source1,source2)*<br>*boolean* **compareGreaterEqual***(source1,source2)*<br>*boolean* **compareLess***(source1,source2)*<br>*boolean* **compareLessEqual***(source1,source2)*<br>*boolean* **compareSignalingNotGreater***(source1,source2)*<br>*boolean* **compareSignalingLessUnordered***(source1,source2)*<br>*boolean* **compareSignalingNotLess***(source1,source2)*<br>*boolean* **compareSignalingGreaterUnordered***(source1,source2)*<br>*boolean* **compareQuietGreater***(source1,source2)*<br>*boolean* **compareQuietGreaterEqual***(source1,source2)*<br>*boolean* **compareQuietLess***(source1,source2)*<br>*boolean* **compareQuietLessEqual***(source1,source2)*<br>*boolean* **compareUnordered***(source1,source2)*<br>*boolean* **compareQuietNotGreater***(source1,source2)*<br>*boolean* **compareQuietLessUnordered***(source1,source2)*<br>*boolean* **compareQuietNotLess***(source1,source2)*<br>*boolean* **compareQuietGreaterUnordered***(source1,source2)*<br>*boolean* **compareOrdered***(source1,source2)* |
| *destination* **f**(*source*) for all the functions **f** in Table D.1 |

Many languages define generic floating-point operations with operator symbols or functional form. These symbols and functions do not specify the destination format of the floating-point result; rather the prevailing Widento method and the formats of the operands imply the destination format to which the infinite-precision floating-point result is to be rounded. Thus *source1*, *source2*, and *source3* might be different floating-point formats. Non-canonical encodings are never propagated.

The Widento methods define the width of a generic operation to be the maximum of the widths of its operands and the width of the widentoFormat, if any is in effect. That "maximum" implies an ordering among the formats of the operands—one must be a subset of the other (see B.1).

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# Annex D
# (informative)
# Elementary transcendental functions

Means are known by which certain elementary transcendental functions may be computed correctly rounded, in all rounding direction modes — but in some cases over limited domains. Implementations should provide correctly-rounded versions of the functions listed in Table D.1, for binary32 format if supported and binary64 format if supported.

Implementations should also provide faithfully-rounded versions when these are significantly more efficient than correctly-rounded. For all other basic formats, these functions should be faithfully-rounded.

Faithful rounding is defined thus: let x denote an infinitely precise number to be rounded according to the prevailing rounding direction mode:

- In a rounding direction mode to nearest, if *x* is ~~representable~~ a floating-point number, *x* is the faithfully rounded result.
- In a rounding direction mode to nearest, if *x* is not ~~representable~~ a floating-point number, either of the two nearest ~~representable~~ floating-point numbers bracketing *x* is the faithfully rounded result.
- In a directed rounding mode, if *x* is ~~representable~~ a floating-point number, either *x* or the next ~~representable~~ floating-point number in the specified direction is the faithfully-rounded result.
- In a directed rounding mode, if *x* is not ~~representable~~ a floating-point number, either of the two ~~representable~~ floating-point numbers nearest *x* in the specified direction is the faithfully rounded result.

Furthermore, faithfully-rounded results should preserve important properties of the unrounded and correctly-rounded functions:

- exactly representable results,
- monotonicity,
- symmetry in rounding direction modes to nearest.

Because these functions are transcendental, they are almost always inexact; when results are inexact but no other exception is signaled, languages define whether the inexact exception is signaled, not signaled, or indeterminate, but the inexact exception should not be signaled for exact results.

For all functions, signaling NaN operands signal the invalid exception.

For expm1, log1p, sinPi, atanPi, sin, and atan, f(+0) is +0 and f(−0) is −0.

Languages should define which other mathematical functions should or should be provided in corrrectly-rounded and faithfully-rounded versions.

When a language specifies elementary transcendental functions, each implementation should document the worst-case accuracies achieved and indicate whether the accuracies are proven or measured for a subset of inputs.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

**Table D.1—Standardized transcendental functions**

| Operation | Function | Correctly-rounded domain | Exceptions |
|---|---|---|---|
| exp | $e^x$ | $[-\infty, +\infty]$ | overflow; underflow |
| expm1 | $e^x$-1 | $[-\infty, +\infty]$ | overflow; underflow |
| sinh | sinh($x$) | $[-\infty, +\infty]$ | overflow |
| cosh | cosh($x$) | $[-\infty, +\infty]$ | overflow |
| log<br>log2<br>log10 | $\log_e(x)$<br>$\log_2(x)$<br>$\log_{10}(x)$ | $(0, +\infty]$ | $x = 0$: division by zero;<br>$x < 0$: invalid |
| log1p | $\log_e(1+x)$ | $(-1, +\infty]$ | $x = -1$: division by zero;<br>$x < -1$: invalid |
| sinPi | $\sin(\pi \times x)$ | empty | $\|x\|=\infty$: invalid;<br>underflow |
| cosPi | $\cos(\pi \times x)$ | empty | $\|x\|=\infty$: invalid |
| atanPi | atan($x$)/$\pi$ | empty | underflow |
| sin | sin($x$) | $[-\pi, +\pi]$ | $\|x\|=\infty$: invalid;<br>underflow |
| cos | cos($x$) | $[-\pi, +\pi]$ | $\|x\|=\infty$: invalid |
| tan | tan($x$) | $[-\pi, +\pi]$ | $\|x\|=\infty$: invalid;<br>underflow, overflow |
| asin | asin($x$) | $[-1, +1]$ | $\|x\| > 1$: invalid |
| acos | acos($x$) | $[-1, +1]$ | $\|x\| > 1$: invalid |
| atan | atan($x$) | [–tan(P2), +tan(P2)]<br>for $\|x\|$ > tan(P2), see text below | underflow |

Some functions, such as cosPi and log, can underflow and/or overflow in an abnormal format with a huge precision and a small exponent field. These are not noted in Table D.1 and are not anticipated to occur in common practice.

All functions are faithfully rounded outside the correctly-rounded domain, except:

For atan, P2 is $\pi/2$ rounded toward zero in the format of $x$.

When |x| > tan(P2) in rounding direction modes to nearest, atan($x$) is copySign(P2,$x$) and might not be correctly rounded.

When |x| > tan(P2) in directed rounding direction modes, atan($x$) is correctly rounded to ±P2 or to ±nextUp(P2), in order to support interval arithmetic inclusion.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# Annex E
# (informative)
# Alternate exception handling modes

## E.1 Overview

Languages should define, and require implementations to provide, means for the user to attach alternate exception handling modes to blocks, language-defined syntactic units (see 6.2). Alternate exception handlers specify lists of exceptions and actions to be taken for each listed exception if it is signaled. Exception lists may contain:

— Any operation-specific exceptions (e.g. 0/0, $\infty - \infty$). The names are language-defined.

— One of the five exception classes: invalid, divideByZero, overflow, underflow, inexact.

— **allExceptions**: all of the aforementioned five exception classes

All implementations should provide alternate exception handling for the superclass **allExceptions**, the five exception classes, and operation-specific exceptions as well.

Languages should provide the non-resumable alternate exception handling modes listed in E.2, and the resumable alternate exception handling modes listed in E.3. The syntax and scope for such mode declarations are language-defined.

## E.2 Non-resumable alternate exception handling modes

Non-resumable-mode alternate exception handling attached to a block means: handle the implied exceptions according to the non-resumable mode specified, then abandon execution of the block attached to and resume execution elsewhere as indicated. Languages should define, and require implementations to provide, these non-resumable modes:

— {block} attached to a block: abandon execution of the attached block and execute the alternate block. The extent to which the original block is evaluated is language-defined, so the alternate handling block should make no assumptions about values of variables that might have been changed.

— transfer attached to a block: transfers control; no return possible. transfer is a language-specific idiom for non-resumable control transfer; conventional languages should offer several transfer idioms such as

  — **goto label:** label might be local or global according to the semantics of the language.

  — **break:** abandon the block controlled by this exception handling and go to the next block.

  — **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 subprogram.

When a block is interrupted for non-resumable alternate exception handling, none, some, or all of the variables assigned in that block may be in an undefined state. Some programming environments might choose to checkpoint all variables prior to executing the protected block, and then restore them prior to executing the alternate block; others leave the responsibility to the programmer to decide which variables should be checkpointed prior to entry and then to explicitly restore them in the alternate block as needed.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

## E.3 Resumable alternate exception handling modes

Resumable-mode alternate exception handling attached to a block means: handle the implied exceptions according to the resumable mode declared, and continue execution of the block attached to. Implementations should support the restoreDefaults mode and should support these other resumable modes:

— restoreDefaults attached to a block:
Restores the (static) default exception handling despite alternate exception handling that might be in effect in outer contexts.

— substitute($x$) (applicable to 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.

— substituteExor($x$) (applicable to any exception arising from multiplication or division):
Like substitute($x$), but replace the default result of such an exceptional operation, if not a NaN, with $|x|$ and attach the EXOR of the signs of the operands.

— abruptUnderflow:
Replace tiny results with zero (or minimum normal in directed rounding modes) results of appropriate signs, raise the underflow flag, and signal inexact.

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# Annex F
# (informative)
# Scaled-product operations

Implementations should provide the following reduction homogeneous computational operations for all supported non-storage floating-point formats. Unlike the rest of the operations in this standard, these operate on arrays of length *n*, and may evaluate products in any order and in any wider format, so results (including flags) might not be identical on different implementations. These operations may signal both inexact and invalid. These operations avoid overflow and underflow to compute a scaled product *pr* and a scale factor *sf*; the proper unscaled product could be recovered with scaleB(*pr*,*sf*) in the absence of over/underflow. The preferred exponent is 0.

— (*sourceFormat, logBformat*) **scaledProd** ( *source array, int* )
{*pr,sf*} = scaledProd( *p, n* ) where *p* is an array of length *n*; scaleB( *pr, sf* ) computes $\prod_{(i = 1,n)} p_i$

— (*sourceFormat, logBformat*) **scaledProdSum** ( *source array, source array, int* )
{*pr,sf*} = scaledProdSum( *p, q, n* ) where *p* and *q* are arrays of length *n*; scaleB( *pr, sf* ) computes $\prod_{(i = 1,n)} (p_i + q_i)$

— (*sourceFormat, logBformat*) **scaledProdDiff** ( *source array, source array, int* )
{*pr,sf*} = scaledProdDiff( *p, q, n* ) where *p* and *q* are arrays of length *n*; scaleB( *pr, sf* ) computes $\prod_{(i = 1,n)} (p_i - q_i)$


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

# Annex G
# (informative)
# Program debugging support

## G.1 Overview

Implementations of this standard vary in the relative priority they assign to characteristics like performance and debuggability (the ability to debug). Therefore this standard does not require all implementations to provide all the debugging support that would be desirable if debuggability were the most important desideratum. 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 may be very expensive in performance compared to fully optimized code.

High-level debugging includes tasks like

— finding where~~,~~
— finding why~~,~~
— testing program fixes~~.~~

in order to investigate

— numerical sensitivity~~,~~
— numerical exceptions~~,~~
— programming errors such as accessing uninitialized storage that are only manifested as incorrect numerical results.

## G.2 Numerical sensitivity

Debuggers should be able to alter the modes governing handling of exceptions inside subprograms, even if the source code for those subprograms is not available. For instance, changing the rounding direction or precision during execution may help identify subprograms that are unusually sensitive to roundoff, 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 mode. The higher-level goal is to determine responsibility for numerical misbehavior, especially in separately-compiled subprograms. The means to achieve that goal is to facilitate the production of small reproducible test cases that elicit unexpected behavior.

## G.3 Numerical exceptions

Debuggers should be able to detect and pause to the debugger when a prespecified exception is signaled within a particular subprogram, or within specified ~~undebugged~~ 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 may well inhibit optimization.

Debuggers should be able to raise and lower status flags.

Debuggers should be able to examine all the unrequited exceptions 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 pointer to a detailed record of its origin and history. By default, even a ~~debugged~~ subprogram presumed to be debugged should at least insert a ~~pointer~~ reference to its name, in an exception flag and in the payload of any

Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.

new quiet NaN produced as a floating-point result of an invalid operation. These ~~pointers~~ 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 symbolic link to a record containing a numerical value extended by an exception history, extra exponent, or extra significand. Consequently bitwise operations like negate, abs, and copySign, which are normally silent, should detect signaling NaNs. Furthermore the signaling attribute of signaling NaNs should be able to be enabled or disabled globally or within a particular context, without disrupting or being affected by a program's logic for default or alternate ~~invalid exception~~ handling of other invalid exceptions.

## G.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. Debbugers should be able to cause non-canonical significand fields to signal an exception. ~~Whether non-canonical significand fields signal an exception is language-defined.~~

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 may prove refractory to static analysis if they involve multiple aliases to the same physical storage. If all-ones bit patterns were defined to be signaling NaNs, then such usages might be isolated earlier.

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 point to symbol table entries describing their origin in terms of the source program.


Copyright © 2006 IEEE. All rights reserved.
This is an unapproved draft, subject to change.