SIGPLAN NOTICES

A Monthly Publication of the Special Interest Group on Programming Languages

VOLUME 22 NUMBER 2 FEBRUARY 1987

CONTENTS:

Language Journals and Newsletters List

As a service to its members, SIGPLAN is compiling a list of journals and newsletters which deal specifically with programming languages and language issues. Once the list has been compiled, it will be published in **SIGPLAN Notices**. If you are the publisher or editor of such a publication, or if you are connected with such a publication in some other capacity and believe it unlikely that the editor or publisher will see this notice, please send the following information by **1 March 1987** to

Steven S. Muchnick
Attn: Pubslist
Sun Microsystems, Inc. MS 5-40
2550 Garcia Avenue
Mountain View, CA 94043

Name of publication:

Name of publisher:

Short description of purpose:

Year of first publication:

Number of times published per year:

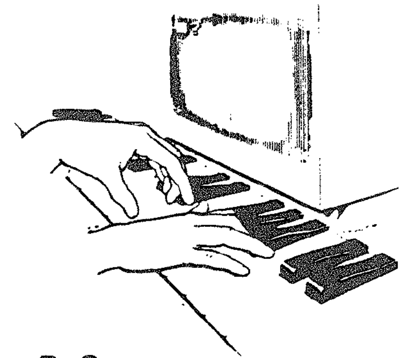
Cost (if any):

How to subscribe:

Organizational affiliation of publication (if any):

How to submit material for publication:

CALL FOR PAPERS



Second Workshop on Empirical Studies of Programmers

December 8-9, 1987 Washington, D. C.

In Cooperation With:
Software Psychology Society
University of Michigan
MCC
Yale University

The First Workshop on Empirical Studies of Programmers was held in June of 1986. Attendees unanimously agreed that it was a most useful, exciting, and important meeting. We are pleased to announce that the Second Workshop will take place in December,1987. In order to facilitate interaction, attendance will be limited to 100 persons. We hereby request those working in this area to submit high quality papers to this Second Workshop.

Suggested Topics:
Topics of interest include, but are not limited to the following:

* Cognitive models of all aspects of programming, e.g., design, generation, comprehension, debugging, maintenance
* The relation of programming to problem-solving
* The use of programming tools, environments, and documentation
* The effects of style, and control and data structures, on program comprehension, production and maintenance
* Evaluations of programming methodologies
* Programming and the non-professional programmer
* Assessments of programmer abilities
* Studies of programmers in "programming-in-the-large"

Information For Authors:
Six copies of a double-spaced, 15-25 page manuscript should be submitted to:
Dr. Gary Olson
Cognitive Science and
Machine Intelligence Laboratory
904 Monroe Street
The University of Michigan
Ann Arbor, MI 48109

Accepted papers will appear in a volume which will be available at the workshop.

Important Dates:
Submission Deadline: May 1, 1987
Acceptance Notification: July 1, 1987
Final Version Due: August 1, 1987
Conference Date: December 8-9, 1987

Conference Committee:

Conference Co-Chairs:
Elliot Soloway
Dept. of Computer Science
Yale University
New Haven, CT 06520

Bill Curtis
MCC
9430 Research Blvd.
Austin, TX 78759

Program Co-Chairs:
Gary Olson
Cognitive Science and
Machine Intelligence Laboratory
The University of Michigan
904 Monroe Street
Ann Arbor, MI 48109

Sylvia Sheppard
Computer Technology Associates
7501 Forbes Blvd.
Lanham, MD 29796

Local Arrangements:
Stan Rifkin
Master Systems
P.O. Box 7108
McLean, VA 22106

Program Committee:
Victor Basili
Deborah Boehm-Davis
Ruven Brooks
Marc Eisenstadt
Thomas Green
Jean-Michel Hoc
S. Sitharama Iyengar
Clayton Lewis
Tony Norcio
Roy Pea
Ben Shneiderman
Kathleen Swigger
Marvin Zelkowitz
Nicholas Zvengintzov
Stuart Zweben

CALL FOR PAPERS

Sixth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC87)

Vancouver, British Columbia, Canada
August 10-12, 1987

Original research contributions are sought that address fundamental issues in the theory and practice of distributed and concurrent systems. Topics of interest include, but are not limited to, the following aspects of concurrent and distributed systems:

* Principles of distributed computation derived from practical experience with working systems
* Algorithms and complexity
* Specification, semantics, and verification
* Programming languages and programming language constructs
* Fault tolerance

Important Dates:

Jan. 30, 1987:	Abstracts due.
Apr. 10, 1987:	Authors informed of acceptance or rejection.
May 15, 1987:	A final copy of each accepted paper due, typed on special forms for inclusion in the conference proceedings.

Please send *eleven* copies of a detailed abstract (not the complete paper), with the address, telephone number, and *net address* (if available) of a contact author on the cover page, to the Program Chair:

Fred B. Schneider
Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853

The abstract should be no more than *10 double-spaced typewritten pages*. It must include a clear description of the problem being discussed, comparisons with extant work, and a section on major original contributions. There should be enough detail provided for the program committee to make a decision.

Submissions that arrive late or are too long are likely to be rejected without consideration of their merits.

Conference Chair: David Kirkpatrick, University of British Columbia (*kirk@ubc.csnet*)

Publicity Chair: Tiko Kameda, Simon Fraser University (*tiko%lccr.sfu.cdn@ubc.csnet*)

The Program Committee:

Andrew Birrell, DEC
Danny Dolev, Hebrew University
Nissim Francez, Technion
Eli Gafni, UCLA
Vassos Hadzilacos, Toronto
Leslie Lamport, DEC
Barbara Liskov, MIT
Michael Merritt, AT&T Bell Laboratories
Fred Schneider, Cornell
Eli Upfal, IBM Almaden

FOR IMMEDIATE RELEASE JULY 10, 1986

MAJOR CONFERENCE PLANNED ON HUMAN -- COMPUTER INTERACTION

The major North American conference focusing on the improvement of interaction between humans and computers will be held in Toronto, April 5 - 9, 1987.

The event will be unique because it will bring together specialists in computer graphics and human-computer interaction to present research results, discuss issues of mutual concern, and take part in specialized training courses. "The conference will focus on helping make computers easier to use", said conference Co-Chair Ron Baecker, Co-director of the Dynamic Graphics Project at the University of Toronto.

The Conference, CHI + GI '87, encourages participation from both academia and industry in the wide variety of disciplines involved in computer graphics and human-computer interaction.

Of interest to practitioners, researchers, teachers, students, computer artists, and others working in the field, the conference will include tutorials, interactive poster sessions, videotape presentations, demonstrations, and an electronic theatre evening featuring the latest in spectacular computer graphics.

The Conference is a combination of CHI'87 (Human Factors in Computing Systems) and GI '87 (Graphics Interface). The annual CHI conference (sponsored by the ACM (Association for Computing Machinery) Special Interest Group on Computers and Human Interaction, SIGCHI) is the leading forum for the presentation of original designs and research in all aspects of human-computer interaction. The annual Graphics Interface conference, sponsored by the Canadian Man-Computer Communications Society (CMCCS), is the oldest regularly scheduled computer graphics conference.

Toronto is a major North American centre for research on and application of enhanced computer human interfaces and computer graphics. More than 1,500 delegates are expected to attend the conference at the Toronto Hilton Harbour Castle Hotel, April 5 - 9, 1987.

- 30 -

A copy of the Call for Participation is included with this press release.

For Further Information:
Wendy Walker
CHI + GI 1987 Conference Office
Computer Systems Research Institute
University of Toronto
2002 - 10 Kings College Road
Toronto, Ontario, Canada M5S 1A4

Phone: 416-978-5184
Electronic mail: WWalker.CHI@Xerox.

Nominations for SIGPLAN Executive Committee

The SIGPLAN nominating committee (Mary S. Van Deusen and Steven S. Muchnick) have put together the following slate for the SIGPLAN 1987 election:

Chair:	Mark Scott Johnson and Robert F. Mathis
Vice-chair:	Teri F. Payton and David S. Wise
Sec/Treasurer:	Robert H. Halstead and Peter S. Mager
Members-at-large:	Frances E. Allen, Stuart I. Feldman, David A. Fisher, Brent T. Hailpern, Julian A. Padget

ACM and SIGPLAN by-laws require that all members of SIGPLAN be notified of their right to nominate candidates by petition. The ballot does not discriminate between candidates selected by the nominating committee and those who choose to run by petition. Members wishing to petition for candidacy must, by **March 15, 1987**, inform the following three individuals of their intention to petition.

Pat Ryan
ACM
11 West 42nd Street
New York, New York 10036

Teri F. Payton
SIGPLAN Secretary
SDC, A Burroughs Company
PO Box 517
Paoli, Pennsylvania 19301

Doris K. Lidtke
SIGBOARD Area E Director
Computer and Information Science
Towson State University
Baltimore, Maryland 21204

Upon notification of intent to petition, ACM headquarters will send you the appropriate petition forms. You must acquire 103 signatures of voting SIGPLAN members on these forms (one percent of SIGPLAN membership) and return the forms to ACM headquarters (Pat Ryan) by **April 15, 1987** for verification of voting member status.

PASCAL PERVERSIONS

Philip Machanick, Craig Levieux and Paul Dadswell
Computer Science Department, University of the Witwatersrand
1 Jan Smuts Ave, BRAAMFONTEIN 2001, South Africa.

 # 1 in a series

```
PROGRAM perverse(output);
  VAR integer : real;

  PROCEDURE aha;
    VAR real : -maxint..maxint;
  BEGIN
  real    := 5;
  integer := 5.4;
  write('real',real,' integer=',integer);
  END;

BEGIN aha; END.
```

 # 2

```
program MOREPERVERSE (OUTPUT,INPUT);
  const FALSE = TRUE;
begin
if FALSE then WRITE('true')
         else WRITE('false');
end.
```

and # 3

```
                    PROGRAM bizzare (output);
                       CONST truth = false;

                                  PROCEDURE a;
                        CONST false = true;

                                PROCEDURE b;
                      CONST true = truth;
                                     BEGIN
                            CASE 5 < 4 OF
      true   : write('5 is less than 4');
  false : write('5 is not less than 4');
                                    END;
                                      END;

                                       BEGIN
                                          b;
                                        END;

                                         BEGIN
                                            a;
                                          END.
```

All of the above programs compiled and executed successfully on the IBM Pascal/VS compiler running the "standard" option under CMS.

Q'Nial

N.I.A.L. Systems Limited P.O. Box 280, Alexandria Bay, New York 13607-0280 1-800-267-0660

October 16, 1986

Dr. Richard Wexelblat, Editor
SIGPLAN Notices
Philips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510

Dear Dr. Wexelblat,

In the note from Edward Cherlin of APL Market News dated June 12, 1986, which you kindly printed in the SIGPLAN notices, there was an unfortunate typographical error in our telephone number.

The correct number for toll-free calls from the U.S. is:

1-800-267-0660

Persons who call that number may request a free reprint of Mr. Cherlin's review of Q'Nial which was recently published in APL Market News.

As well, we have moved to a new office and use P.O. Box 2128, Kingston ON K7L 5J8 as the mailing address.

A "starter" version of Q'Nial is now available for the IBM PC for the grand sum of $99.00 U.S. This price includes the documentation and shipping expenses. This version implements most of the array theory but does not include some of the advanced features found in the regular Q'Nial.

An "enhanced" Q'Nial with an Artificial Intelligence Toolkit will be available in December '86.

We appreciate your assistance in advising the scientific community of this interesting product.

Sincerely,

William H. Jenkins

William H. Jenkins,
President.

Twente University of Technology

DEPARTMENT OF COMPUTER SCIENCE
P.O. BOX 217 - 7500 AE ENSCHEDE
THE NETHERLANDS.

Dr. Richard L. Wexelblat
Editor SIGPLAN Notices

Your reference Our reference Phone 053- Date 30 Sept 1986

Dear Dr. Wexelblat:

In my contribution to the Sept 1986 issue of SIGPLAN Notices entitled "A Useful Application of Formal Procedure Parameters" a small error slipped through. The procedure as given prints the path from root to leaf except for the last item. It can be corrected by replacing the call to `path´ in the body of `paths´ by the call `localpath´.

I would like to apologize for this error and I also would like to thank the readers of SIGPLAN Notices who brought it to my attention.

Sincerely,

Peter van Eijk

uucp: decvax!mcvax!utrcu1!infpve

ANSI/IEEE
Std 754-1985

An American National Standard

IEEE Standard for Binary Floating-Point Arithmetic

Sponsor

Standards Committee of the IEEE Computer Society

Approved March 21, 1985

IEEE Standards Board

Approved July 26, 1985

American National Standards Institute

The Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street, New York, NY 10017, USA

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least once every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
345 East 47th Street
New York, NY 10017
USA

Foreword

(This Foreword is not a part of ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.)

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. Draft 8.0 of this standard was published to solicit public comments.[1] Implementation techniques can be found in *An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic* by Jerome T. Coonen,[2] which was based on a still earlier draft of the proposal.

This standard defines a family of commercially feasible ways for new systems to perform binary floating-point arithmetic. The issues of retrofitting were not considered. Among the desiderata that guided the formulation of this standard were

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

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

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

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

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

(6) Enable rather than preclude further refinements and extensions.

[1] *Computer Magazine* vol 14, no 3, March 1981.
[2] *Computer Magazine* vol 13, no 1, January 1980.

Members of the Floating-Point Working Group of the Microprocessor Standards Subcommittee and those who participated by correspondence were as follows:

David Stevenson, *Chairman*

Andrew Allison
William Ames
Mike Arya
Janis Baron
Steve Baumel
Dileep Bhandarkar
Joel Boney
E.H. Bristol
Werner Buchholz
Jim Bunch
Ed Burdick
Gary R. Burke
Paul Clemente
W.J. Cody
Jerome T. Coonen
Jim Crapuchettes
Itzhak Davidesko
Wayne Davison
R.H. Delp
James Demmel
Donn Denman
Alvin Despain
Augustin A. Dubrulle
Tom Eggers
Philip J. Faillace
Richard Fateman
David Feign
Don Feinberg
Stuart Feldman
Eugene Fisher
Paul F. Flanagan
Gordon Force
Lloyd Fosdick
Robert Fraley
Howard Fullmer
Daniel D. Gajski
David M. Gay
C.W. Gear
Martin Graham
David Gustavson
Guy K. Haas
Kenton Hanson
Chuck Hastings
David Hough
John Edward Howe
Thomas E. Hull
Suren Irukulla
Richard E. James III
Paul S. Jensen
W. Kahan
Howard Kaikow
Richard Karpinski
Virginia Klema
Les Kohn
Dan Kuyper
M. Dundee Maples
Roy Martin
William H. McAllister
Colin McMaster
Dean Miller
Webb Miller
John C. Nash
Dan O'Dowd
Cash Olsen
A. Padegs
John F. Palmer
Beresford Parlett
Dave Patterson
Mary H. Payne
Tom Pittman
Lew Randall
Robert Reid
Christian Reinsch
Frederic N. Ris
Stan Schmidt
Van Shahan
Robert L. Smith
Roger Stafford
G.W. Stewart
Robert Stewart
Harold S. Stone
W.D. Strecker
Robert Swarz
George Taylor
James W. Thomas
Dar-Sun Tsien
Greg Walker
John Steven Walther
Shlomo Waser
P.C. Waterman
Charles White

When the IEEE Standards Board approved this standard on March 21, 1985, it had the following membership:

John E. May, *Chairman*

John P. Riganati, *Vice Chairman*

Sava I. Sherr, *Secretary*

James H. Beall
Fletcher J. Buckley
Rene Castenschiold
Edward Chelotti
Edward J. Cohen
Paul G. Cummings
Donald C. Fleckenstein
Jay Forster
Daniel L. Goldberg
Kenneth D. Hendrix
Irvin N. Howell, Jr
Jack Kinn
Joseph L. Koepfinger*
Irving Kolodny
R.F. Lawrence
Lawrence V. McCall
Donald T. Michael*
Frank L. Rose
Clifford O. Swanson
J. Richard Weger
W.B. Wilkens
Charles J. Wylie

*Member emeritus

Contents

An American National Standard

IEEE Standard for Binary Floating-Point Arithmetic

1. Scope

1.1 Implementation Objectives. 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 programmer or 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

(1) Basic and extended floating-point number formats

(2) Add, subtract, multiply, divide, square root, remainder, and compare operations

(3) Conversions between integer and floating-point formats

(4) Conversions between different floating-point formats

(5) Conversions between basic format floating-point numbers and decimal strings

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

1.3 Exclusions. This standard does not specify

(1) Formats of decimal strings and integers

(2) Interpretation of the sign and significand fields of NaNs

(3) Binary ↔ decimal conversions to and from extended formats

2. Definitions

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

binary floating-point number. A bit-string characterized by three components: a sign, a signed exponent, and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent. In this standard a bit-string is not always distinguished from a number it may represent.

denormalized number. A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

destination. The location for the result of a binary or unary operation. 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.

exponent. The component of a binary floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.

fraction. The field of the significand that lies to the right of its implied binary point.

mode. A variable that a user may set, sense, save, and restore to control the execution of subsequent arithmetic operations. The default mode is the mode that a program can assume to be in effect unless an explicitly contrary statement is included in either the program or its specification. The following mode shall be implemented: rounding, to control the direction of rounding errors. In certain implementations, rounding precision may be required, to shorten the precision of results.
The implementor may, at his option, implement the following modes: traps disabled/enabled, to handle exceptions.

NaN. Not a number, a symbolic entity encoded in floating-point format. There are two types of NaNs (6.2). Signaling NaNs signal the invalid operation exception (7.1) whenever they appear as operands. Quiet NaNs propagate through almost every arithmetic operation without signaling exceptions.

result. The bit string (usually representing a number) that is delivered to the destination.

significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

shall. The use of the word *shall* signifies that which is obligatory in any conforming implementation.

should. The use of the word *should* signifies that which is strongly recommended as being in keeping with the intent of the standard, although architectural or other constraints beyond the scope of this standard may on occasion render the recommendations impractical.

status flag. A variable that may take two states, set and clear. A user may clear a flag, copy it, or restore it to a previous state. When set, a status flag may contain additional system-dependent information, possibly inaccessible to some users. The operations of this standard may as a side effect set some of the following flags: inexact result, underflow, overflow, divide by zero, and invalid operation.

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. Formats

This standard defines four floating-point formats in two groups, basic and extended, each having two widths, single and double. The standard levels of implementation are distinguished by the combinations of formats supported.

3.1 Sets of Values. This section concerns only the numerical values representable within a format, not the encodings. The only values representable in a chosen format are those specified by way of the following three integer parameters:

p = the number of significand bits (precision)
$E_{\max}$ = the maximum exponent
$E_{\min}$ = the minimum exponent

Each format's parameters are given in Table 1. Within each format only the following entities shall be provided:

Numbers of the form $(-1)^s 2^E (b_0 \bullet b_1 b_2 \cdots b_{p-1})$

where

s = 0 or 1
E = any integer between $E_{\min}$ and $E_{\max}$, inclusive
b_i = 0 or 1

Two infinities, $+\infty$ and $-\infty$
At least one signaling NaN
At least one quiet NaN

The foregoing description enumerates some values redundantly, for example, $2^0(1 \bullet 0) = 2^1(0 \bullet 1) = 2^2(0 \bullet 01) = \cdots$. However, the encodings of such nonzero values may be redundant only in extended formats (3.3). The nonzero values of the form $\pm 2^{E_{\min}}(0 \bullet b_1 b_2 \cdots b_{p-1})$ are called denormalized. Reserved exponents may be used to encode NaNs, $\pm\infty$, ± 0, and denormalized num-

Table 1
Summary of Format Parameters

Parameter	Format			
	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
E_{max}	$+127$	$\geq +1023$	$+1023$	$\geq +16383$
E_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent *bias*	$+127$	unspecified	$+1023$	unspecified
Exponent width in bits	8	≥ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

bers. 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 signs are significant in some circumstances, such as division by zero, and not in others. In this standard, 0 and ∞ are written without a sign when the sign is not important.

3.2 Basic Formats. Numbers in the single and double formats are composed of the following three fields:

(1) 1-bit sign s
(2) Biased esponent $e = E+bias$
(3) Fraction $f = \bullet\, b_1 b_2 \cdots b_{p-1}$

The range of the unbiased exponent E shall include every integer between two values E_{min} and E_{max}, inclusive, and also two other reserved values $E_{min}-1$ to encode ± 0 and denormalized numbers, and $E_{max}+1$ to encode $\pm\infty$ and NaNs. The foregoing parameters are given in Table 1. Each nonzero numerical value has just one encoding. The fields are interpreted as follows:

3.2.1 Single. A 32-bit single format number X is divided as shown in Fig 1. The value v of X is inferred from its constituent fields thus

(1) If $e = 255$ and $f \neq 0$, then v is NaN regardless of s
(2) If $e = 255$ and $f = 0$, then $v = (-1)^s \infty$
(3) If $0 < e < 255$, then $v = (-1)^s 2^{e-127}(1 \bullet f)$
(4) If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-126}(0 \bullet f)$ (denormalized numbers)
(5) If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

3.2.2 Double. A 64-bit double format number X is divided as shown in Fig 2. The value v of X is inferred from its constituent fields thus

Fig 1
Single Format

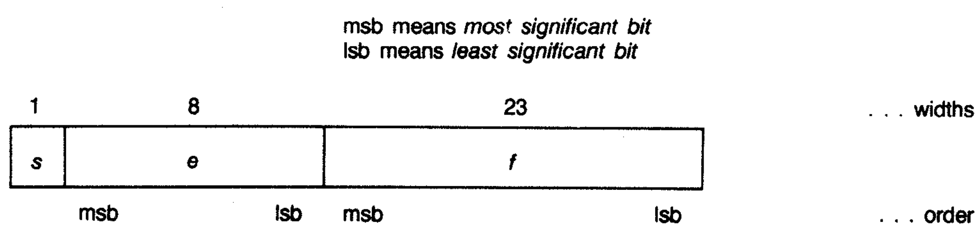
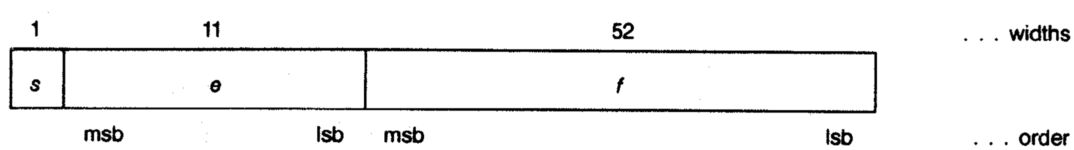


Fig 2
Double Format



(1) If $e = 2047$ and $f \neq 0$, then v is NaN regardless of s

(2) If $e = 2047$ and $f = 0$, then $v = (-1)^s\infty$

(3) If $0 < e < 2047$, then $v = (-1)^s 2^{e-1023}(1 \bullet f)$

(4) If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-1022}(0 \bullet f)$ (denormalized numbers)

(5) If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

3.3 Extended Formats. The single extended and double extended formats encode in an implementation-dependent way the sets of values in 3.1 subject to the constraints of Table 1. This standard allows an implementation to encode some values redundantly, provided that redundancy be transparent to the user in the following sense: an implementation either shall encode every nonzero value uniquely or it shall not distinguish redundant encodings of nonzero values. An implementation may also reserve some bit strings for purposes beyond the scope of this standard. When such a reserved bit string occurs as an operand the result is not specified by this standard.

An implementation of this standard is not required to provide (and the user should not assume) that single extended have greater range than double.

3.4 Combinations of Formats. All implementations conforming to this standard shall support the single format. Implementations should support the extended format corresponding to the widest basic format supported, and need not support any other extended format.[3]

4. Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception (7.5). Except for binary$\longleftrightarrow$decimal conversion (whose weaker conditions are specified in 5.6), every operation specified in Section 5 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 section.

The rounding modes affect all arithmetic operations except comparison and remainder. The rounding modes may affect the signs of zero sums (6.3), and do affect the thresholds beyond which overflow (7.3) and underflow (7.4) may be signaled.

4.1 Round to Nearest. An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered. However, an infinitely precise result with magnitude at least $2^{E_{\max}}(2-2^{-p})$ shall round to ∞ with no change in sign; here $E_{\max}$ and p are determined by the destination format (see Section 3) unless overridden by a rounding precision mode (4.3).

4.2 Directed Roundings. An implementation shall also provide three user-selectable directed rounding modes: round toward $+\infty$, round toward $-\infty$, and round toward 0.

When rounding toward $+\infty$ the result shall be the format's value (possibly $+\infty$) closest to and no less than the infinitely precise result. When rounding toward $-\infty$ the result shall be the format's value (possibly $-\infty$) closest to and no greater than the infinitely precise result. When rounding toward 0 the result shall be the format's value closest to and no greater in magnitude than the infinitely precise result.

4.3 Rounding Precision. Normally, a result is rounded to the precision of its destination. However, some systems deliver results only to double or extended destinations. On such a system the user, which may be a high-level language compiler, shall be able to specify that a result be rounded instead to single precision, though it may be stored in the double or extended format with its wider exponent range.[4] Similarly, a system that delivers results only to double extended destinations shall permit the user to specify rounding to single or double precision. Note that to meet the specifications in 4.1, the result cannot suffer more than one rounding error.

[3] Only if upward compatibility and speed are important issues should a system supporting the double extended format also support single extended.

[4] Control of rounding precision is intended to allow systems whose destinations are always double or extended to mimic, in the absence of over/underflow, the precisions of systems with single and double destinations. An implementation should not provide operations that combine double or extended operands to produce a single result, nor operations that combine double extended operands to produce a double result, with only one rounding.

5. Operations

All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the square root, find the remainder, round to integer in floating-point format, convert between different floating-point formats, convert between floating-point and integer formats, convert binary $\leftrightarrow$ decimal, and compare. Whether copying without change of format is considered an operation is an implementation option. Except for binary $\leftrightarrow$ decimal conversion, each of the operations 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 Sections 4 and 7). Section 6 augments the following specifications to cover ± 0, $\pm\infty$, and NaN; Section 7 enumerates exceptions caused by exceptional operands and exceptional results.

5.1 Arithmetic. An implementation shall provide the add, subtract, multiply, divide, and remainder operations for any two operands of the same format, for each supported format; it should also provide the operations for operands of differing formats. The destination format (regardless of the rounding precision control of 4.3) shall be at least as wide as the wider operand's format. All results shall be rounded as specified in Section 4.

When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - y \times n$, where n is the integer nearest the exact value x/y; whenever $|n - x/y| = \frac{1}{2}$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x. Precision control (4.3) shall not apply to the remainder operation.

5.2 Square Root. The square root operation shall be provided in all supported formats. The result is defined and has a positive sign for all operands ≥ 0, except that $\sqrt{-0}$ shall be -0. The destination format shall be at least as wide as the operand's. The result shall be rounded as specified in Section 4.

5.3 Floating-Point Format Conversions. It shall be possible to convert floating-point numbers between all supported formats. If the conversion is to a narrower precision, the result shall be rounded as specified in Section 4. Conversion to a wider precision is exact.

5.4 Conversion Between Floating-Point and Integer Formats. It shall be possible to convert between all supported floating-point formats and all supported integer formats. Conversion to integer shall be effected by rounding as specified in Section 4. Conversions between floating-point integers and integer formats shall be exact unless an exception arises as specified in 7.1.

5.5 Round Floating-Point Number to Integer Value. It shall be possible to round a floating-point number to an integral valued floating-point number in the same format. The rounding shall be as specified in Section 4, with the understanding that when rounding to nearest, if the difference between the unrounded operand and the rounded result is exactly one half, the rounded result is even.

5.6 Binary $\leftrightarrow$ Decimal Conversion. Conversion between decimal strings in at least one format and binary floating-point numbers in all supported basic formats shall be provided for numbers throughout the ranges specified in Table 2. The integers M and N in Tables 2 and 3 are such that the decimal strings have values $\pm M \times 10^{\pm N}$. On input, trailing zeros shall be appended to or stripped from M (up to the limits specified in Table 2) so as to minimize N. When the destination is a decimal string, its least significant digit should be located by format specifications for purposes of rounding.

When the integer M lies outside the range specified in Tables 2 and 3, that is, when $M \geq 10^9$ for single or 10^{17} for double, the implementor may, at his option, alter all significant digits after the ninth for single and seventeenth for double to other decimal digits, typically 0.

Conversions shall be correctly rounded as specified in Section 4 for operands lying within

Table 2
Decimal Conversion Ranges

Format	Decimal to Binary		Binary to Decimal	
	Max M	Max N	Max M	Max N
Single	10^9-1	99	10^9-1	53
Double	$10^{17}-1$	999	$10^{17}-1$	340

the ranges specified in Table 3. Otherwise, for rounding to nearest, the error in the converted result shall not exceed by more than 0.47 units in the destination's least significant digit the error that is incurred by the rounding specifications of Section 4, provided that exponent over/underflow does not occur. In the directed rounding modes the error shall have the correct sign and shall not exceed 1.47 units in the last place.

Conversions shall be monotonic, that is, increasing the value of a binary floating-point number shall not decrease its value when converted to a decimal string; and increasing the value of a decimal string shall not decrease its value when converted to a binary floating-point number.

When rounding to nearest, conversion from binary to decimal and back to binary shall be the identity as long as the decimal string is carried to the maximum precision specified in Table 2, namely, 9 digits for single and 17 digits for double.[5]

If decimal to binary conversion over/underflows, the response is as specified in Section 7. Over/underflow and NaNs and infinities encountered during binary to decimal conversion should be indicated to the user by appropriate strings. NaNs encoded in decimal strings are not specified in this standard.

To avoid inconsistencies, the procedures used for binary ↔ decimal conversion should give the same results regardless of whether the conversion is performed during language translation (interpretation, compilation, or assembly) or during program execution (run-time and interactive input/output).

Table 3
Correctly Rounded Decimal Conversion Range

Format	Decimal to Binary		Binary to Decimal	
	Max *M*	Max *N*	Max *M*	Max *N*
Single	10^9-1	13	10^9-1	13
Double	$10^{17}-1$	27	$10^{17}-1$	27

5.7 Comparison. It shall be possible to compare floating-point numbers in all supported formats, even if the operands' formats differ. Comparisons are exact and never overflow nor 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 $+0 = -0$).

The result of a comparison shall be delivered in one of two ways at the implementor's option: 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. In addition to the true-false response, an invalid operation exception (7.1) shall be signaled when, as indicated in Table 4, last column, *unordered* operands are compared using one of the predicates involving < or > but not ? (Here the symbol ? signifies *unordered*).

Table 4 exhibits the twenty-six functionally distinct useful predicates named, in the first column, using three notations: *ad hoc*, FORTRAN-like, and mathematical. It shows how they are obtained from the four condition codes and tells which predicates cause an invalid operation exception when the relation is *unordered*. The entries T and F indicate whether the predicate is true or false when the respective relation holds.

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 4 reverses the true/false sense of its associated entries, but leaves the last column's entry unchanged.[6]

Implementations that provide predicates shall provide the first six predicates in Table 4 and should provide the seventh, and a means of logically negating predicates.

[5]The properties specified for conversions are implied by error bounds that depend on the format (single or double) and the number of decimal digits involved: the 0.47 mentioned is a worst-case bound only. For a detailed discussion of these error bounds and economical conversion algorithms that exploit the extended format, *see* COONEN, JEROME T. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic. Ph.D. Thesis,* University of California, Berkeley, CA, 1984.

[6]There may appear to be two ways to write the logical negation of a predicate, one using NOT explicitly and the other reversing the relational operator. For example, the logical negation of $(X = Y)$ may be written either $\mathrm{NOT}(X = Y)$ or $(X \mathbin{?}{<}{>} Y)$; in this case both expressions are functionally equivalent to $(X \neq Y)$. However, this coincidence does not occur for the other predicates. For example, the logical negation of $(X < Y)$ is just $\mathrm{NOT}(X < Y)$, the reversed predicate $(X \mathbin{?}{>}{=} Y)$ is different in that it does not signal an invalid operation exception when X and Y are *unordered*.

Table 4
Predicates and Relations

Predicates			Relations				Exception
Ad hoc	FORTRAN	Math	Greater Than	Less Than	Equal	Unordered	Invalid If Unordered
=	.EQ.	=	F	F	T	F	No
?<>	.NE.	≠	T	T	F	T	No
>	.GT.	>	T	F	F	F	Yes
>=	.GE.	≥	T	F	T	F	Yes
<	.LT.	<	F	T	F	F	Yes
<=	.LE.	≤	F	T	T	F	Yes
?	unordered		F	F	F	T	No
<>	.LG.		T	T	F	F	Yes
<=>	.LEG.		T	T	T	F	Yes
?>	.UG.		T	F	F	T	No
?>=	.UGE.		T	F	T	T	No
?<	.UL.		F	T	F	T	No
?<=	.ULE.		F	T	T	T	No
?=	.UE.		F	F	T	T	No
NOT(>)			F	T	T	T	Yes
NOT(>=)			F	T	F	T	Yes
NOT(<)			T	F	T	T	Yes
NOT(<=)			T	F	F	T	Yes
NOT(?)			T	T	T	F	No
NOT(<>)			F	F	T	T	Yes
NOT(<=>)			F	F	F	T	Yes
NOT(?>)			F	T	T	F	No
NOT(?>=)			F	T	F	F	No
NOT(?<)			T	F	T	F	No
NOT(?<=)			T	F	F	F	No
NOT(?=)			T	T	F	F	No

6. Infinity, NaNs, and Signed Zero

6.1 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. Infinites shall be interpreted in the affine sense, that is, $-\infty<$(every finite number)$<+\infty$.

Arithmetic on ∞ is always exact and therefore shall signal no exceptions, except for the invalid operations specified for ∞ in 7.1. The exceptions that do pertain to ∞ are signaled only when

(1) ∞ is created from finite operands by overflow (7.3) or division by zero (7.2), with corresponding trap disabled

(2) ∞ is an invalid operand (7.1).

6.2 Operations with NaNs. Two different kinds of NaN, signaling and quiet, shall be supported in all operations. Signaling NaNs afford values 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 implementor's discretion, afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the NaNs be preserved through arithmetic operations and floating-point format conversions.

Signaling NaNs shall be reserved operands that signal the invalid operation exception (7.1) for every operation listed in Section 5. Whether copying a signaling NaN without a change of format signals the invalid operation exception is the implementor's option.

Every operation involving a signaling NaN or invalid operation (7.1) shall, if no trap occurs and if a floating-point result is to be delivered, deliver a quiet NaN as its result.

Every operation involving one or two input NaNs, none of them signaling, shall signal no exception but, if a floating-point result is to be delivered, shall deliver as its result a quiet NaN, which should be one of the input NaNs. Note that format conversions might be unable to deliver the same NaN. Quiet NaNs do have effects similar to signaling NaNs on operations that do not deliver a floating-point result; these operations, namely comparison and conversion to a format that has no NaNs, are discussed in 5.4, 5.6, 5.7, and 7.1.

6.3 The Sign Bit. This standard does not interpret the sign of an NaN. Otherwise, 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 the round floating-point number to integral value operation 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 modes except round toward $-\infty$, in which mode that sign shall be $-$. However, $x + x = x - (-x)$ retains the same sign as x even when x is zero.

Except that $\sqrt{-0}$ shall be -0, every valid square root shall have a positive sign.

7. Exceptions

There are five types of exceptions that shall be signaled when detected. The signal entails setting a status flag, taking a trap, or possibly doing both. With each exception should be associated a trap under user control, as specified in Section 8. The default response to an exception shall be to proceed without a trap. This standard specifies results to be delivered in both trapping and nontrapping situations. In some cases the result is different if a trap is enabled.

For each type of exception the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset 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.

The only exceptions that can coincide are inexact with overflow and inexact with underflow.

7.1 Invalid Operation. The invalid operation exception is signaled if an operand is invalid for the operation to be performed. The result, when the exception occurs without a trap, shall be a quiet NaN (6.2) provided the destination has a floating-point format. The invalid operations are

(1) Any operation on a signaling NaN (6.2)

(2) Addition or subtraction—magnitude subtraction of infinites such as, $(+\infty) + (-\infty)$

(3) Multiplication—$0 \times \infty$

(4) Division—$0/0$ or ∞/∞

(5) Remainder— x REM y, where y is zero or x is infinite

(6) Square root if the operand is less than zero

(7) Conversion of a binary floating-point number to an integer or decimal format when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled

(8) Comparison by way of predicates involving $<$ or $>$, without ?, when the operands are *unordered* (5.7, Table 4)

7.2 Division by Zero. If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signaled. The result, when no trap occurs, shall be a correctly signed ∞ (6.3).

7.3 Overflow. The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (Section 4) were the exponent range unbounded. The result, when no trap occurs, shall be determined by the rounding mode and the sign of the intermediate result as follows:

(1) Round to nearest carries all overflows to ∞ with the sign of the intermediate result

(2) Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result

(3) Round toward $-\infty$ carries positive overflows to the format's largest finite number, and carries negative overflows to $-\infty$

(4) Round toward $+\infty$ carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$

Trapped overflows on all operations except conversions shall deliver to the trap handler the result obtained by dividing the infinitely precise result by 2^{α} and then rounding. The bias adjust α is 192 in the single, 1536 in the double, and

$3 \times 2^{n-2}$ in the extended format, when n is the number of bits in the exponent field.[7] Trapped overflow on conversion from a binary floating-point format shall deliver to the trap handler a result in that or a wider format, possibly with the exponent bias adjusted, but rounded to the destination's precision. Trapped overflow on decimal to binary conversion shall deliver to the trap handler a result in the widest supported format, possibly with the exponent bias adjusted, but rounded to the destination's precision; when the result lies too far outside the range for the bias to be adjusted, a quiet NaN shall be delivered instead.

7.4 Underflow. Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between $\pm 2^{E_{\min}}$ which, because it is so tiny, may cause some other exception later such as overflow upon division. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers. The implementor may choose how these events are detected, but shall detect these events in the same way for all operations. Tininess may be detected either

(1) *After rounding*—when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E_{\min}}$

(2) *Before rounding*—when a nonzero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm 2^{E_{\min}}$.

Loss of accuracy may be detected as either

(3) *A denormalization loss*—when the delivered result differs from what would have been computed were exponent range unbounded.

(4) *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 7.5).

When an underflow trap is not implemented, or is not enabled (the default case), underflow shall be signaled (by way of the underflow flag) only when both tininess and loss of accuracy have been detected. The method for detecting tininess and loss of accuracy does not affect the delivered result which might be zero, denormalized, or $\pm 2^{E_{\min}}$. When an underflow trap has been implemented and is enabled, underflow shall be signaled when tininess is detected regardless of loss of accuracy. Trapped underflows on all operations except conversion shall deliver to the trap handler the result obtained by multiplying the infinitely precise result by 2^{α} and then rounding. The bias adjust α is 192 in the single, 1536 in the double, and $3 \times 2^{n-2}$ in the extended format, where n is the number of bits in the exponent field.[8] Trapped underflows on conversion shall be handled analogously to the handling of overflows on conversion.

7.5 Inexact. If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination or, if an inexact trap occurs, to the trap handler.

8. Traps

A user should be able to request a trap on any of the five exceptions by specifying a handler for it. He should be able to request that an existing handler be disabled, saved, or restored. He should also be able to determine whether a specific trap handler for a designated exception has been enabled. When an exception whose trap is disabled is signaled, it shall be handled in the manner specified in Section 7. When an exception whose trap is enabled is signaled the execution of the program in which the exception occurred shall be suspended, the trap handler previously specified by the user shall be activated, and a result, if specified in Section 7, shall be delivered to it.

8.1 Trap Handler. A trap handler should have the capabilities of a subroutine that can return a value to be used in lieu of the exceptional operation's result; this result is undefined unless delivered by the trap handler. Similarly, the flag(s) corresponding to the exceptions being signaled with their associated traps enabled may be undefined unless set or reset by the trap handler.

[7] The bias adjust is chosen to translate over/underflowed values as nearly as possible to the middle of the exponent range so that, if desired, they can be used in subsequent scaled operations with less risk of causing further exceptions.

[8] Note that a system whose underlying hardware always traps on underflow, producing a rounded, bias-adjusted result, shall indicate whether such a result is rounded up in magnitude in order that the correctly denormalized result may be produced in system software when the user underflow trap is disabled.

When a system traps, the trap handler should be able to determine

(1) Which exception(s) occurred on this operation

(2) The kind of operation that was being performed

(3) The destination's format

(4) In overflow, underflow, and inexact exceptions, the correctly rounded result, including information that might not fit in the destination's format

(5) In invalid operation and divide by zero exceptions, the operand values

8.2 Precedence. If enabled, the overflow and underflow traps take precedence over a separate inexact trap.

Appendix
Recommended Functions and Predicates

(This Appendix is not a part of ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.)

The following functions and predicates are recommended as aids to program portability across different systems, perhaps performing arithmetic very differently. They are described generically, that is, the types of the operands and results are inherent in the operands. Languages that require explicit typing will have corresponding families of functions and predicates.

Some functions, such as the copy operation $y := x$ without change of format, may at the implementor's option be treated as nonarithmetic operations which do not signal the invalid operation exception for signaling NaNs; the functions in question are (1), (2), (6), and (7).

(1) Copysign(x,y) returns x with the sign of y. Hence, abs(x) = copysign(x,1.0), even it x is NaN.

(2) $-x$ is x copied with its sign reversed, not $0-x$; the distinction is germane when x is ± 0 or NaN. Consequently, it is a mistake to use the sign bit to distinguish signaling NaNs from quiet NaNs.

(3) Scalb(y,N) returns $y \times 2^N$ for integral values N without computing 2^N.

(4) Logb(x) returns the unbiased exponent of x, a signed integer in the format of x, except that logb(NaN) is a NaN, logb(∞) is $+\infty$, and logb(0) is $-\infty$ and signals the division by zero exception. When x is positive and finite the expression scalb[x,$-$logb(x)] lies strictly between 0 and 2; it is less than 1 only when x is denormalized.

(5) Nextafter(x,y) returns the next representable neighbor of x in the direction toward y. The following special cases arise: if $x = y$, then the result is x without any exception being signaled; otherwise, if either x or y is a quiet NaN, then the result is one or the other of the input NaNs. Overflow is signaled when x is finite but nextafter(x,y) is infinite; underflow is signaled when nextafter(x,y) lies strictly between $\pm 2^{E_{\min}}$, in both cases, inexact is signaled.

(6) Finite(x) returns the value TRUE if $-\infty < x < +\infty$, and returns FALSE otherwise.

(7) Isnan(x), or equivalently $x \neq x$, returns the value TRUE if x is a NaN, and returns FALSE otherwise.

(8) $x <> y$ is TRUE only when $x < y$ or $x > y$, and is distinct from $x \neq y$, which means NOT($x = y$) (Table 4).

(9) Unordered(x,y), or $x?y$, returns the value TRUE if x is unordered with y, and returns FALSE otherwise (Table 4).

(10) Class(x) tells which of the following ten classes x falls into: signaling NaN, quiet NaN, $-\infty$, negative normalized nonzero, negative denormalized, -0, $+0$, positive denormalized, positive normalized nonzero, $+\infty$. This function is never exceptional, not even for signaling NaNs.

Acknowledgements

Appreciation is expressed to the following companies and organizations for contributing the time, talent, and resources of their employees to make possible the development of this standard:

Altos Computer Systems
Analytical Mechanics
Apple Computer
Argonne National Laboratory
Beckman Instruments
Bell AT&T Laboratories
Computervision
Data General Corp
Digital Equipment Corp
ELXSI
ESL Inc
Fairchild Microprocessor
Four Phase Systems
The Foxboro Company
Hewlett Packard Co
IBM Corp
Intel Corp
Itty Bitty Computers
Kylex
Lawrence Livermore Laboratory
Leibniz-Rech/Bay. Akad. Wiss.
Lockheed Research Laboratory
M & E Associates
Massachusetts Institute of Technology
Menlo Computer Associates
MMI
Motorola, Inc
National Semiconductor, Corp
Oregon Software
Parasitic Engineering
Prime Computer
Signetics Corp
Sperry Univac
Stanford Linear Accelerator Center
Stewart Research Ent.
The Systems Group
Tandem
University of Arizona
University of California
University of Colorado
University of Illinois
University of Maryland
University of Massachusetts
University of Ottawa
University of Toronto
Volition Systems
Zilog