INSIDE MACINTOSH

# PowerPC Numerics

**Addison-Wesley Publishing Company**

Reading, Massachusetts  Menlo Park, California  New York
Don Mills, Ontario  Wokingham, England  Amsterdam  Bonn
Sydney  Singapore  Tokyo  Madrid  San Juan
Paris  Seoul  Milan  Mexico City  Taipei

ISBN 0-201-40728-0
1 2 3 4 5 6 7 8 9-CRW-9897969594
First Printing, March 1994

The paper used in this book meets the EPA standards for recycled fiber.

**Library of Congress Cataloging-in-Publication Data**

Inside Macintosh. PowerPC Numerics / Apple Computer, Inc.
p. cm.
Includes index.
ISBN 0-201-40728-0
1. Macintosh (Computer) 2. PowerPC (Microprocessor) I. Apple Computer, Inc.
QA76.8.M3I5622 1994
005.265—dc20 93-51248
CIP

# Contents

# Figures, Tables, and Listings

# About This Book

This book, *Mac OS X Numerics*, is the reference for the numerics environment for Mac OS X. Mac OS X numerics is an environment in which floating-point operations are performed quickly and as accurately as possible. The core features used in Mac OS X numerics are not exclusive to Apple Computer; rather they are taken from IEEE Standard 754, the IEEE Standard for Binary Floating-point Arithmetic, augmented by IEEE standard ISO/IEC 9899, Programming Languages - C (more commonly referred to as referred to as C99). The numerics environment specified by the IEEE Standard 754 is called the IEEE standard numerics environment, or just IEEE standard numerics.

In one sense, IEEE standard numerics is an abstraction: a definition of an environment for computer numerics, independent of a specific computer. To have an instance of this environment, you need a language in which to describe operations and an implementation unit to carry them out. On Mac OS X, the IEEE standard numerics environment is implemented by the Libm math library, which complies with IEEE Standard 754 and C99.

The first part of this book describes the IEEE standard numerics definition, and the remaining parts describe how numerics is implemented in Mac OS X through Libm.

You should read this book if

- you want to create Mac OS X applications that use floating-point operations
- you want to learn more about IEEE Standard 754 and C99 standard for binary floating-point arithmetic

## What's in This Book

Part 1 describes the features shared by all IEEE standard numerics implementations and includes examples that show how to use IEEE standard numerics effectively. These examples are written in C, although other high-level languages might provide support for IEEE standard numerics. Read Part 1 to find out how Mac OS X implements IEEE Standard 754 in general or to learn more about the standard.

Part 2 explains the numeric implementation in compilers and in the Libm math library. This library is provided to implement both IEEE Standard 754 and C99. Part 2 is for use exclusively by C language programmers.

The appendixes provide supplementary reference material. There are summaries of the Libm functions for your reference.

The bibliography at the end of this book lists some of the major sources on numerics. Also at the end of this book are a glossary of terms and an index.

## Conventions Used in This Book

*Inside Macintosh* uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information appears in special formats so that you can scan it quickly.

### Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary at the end of this book.

When a word or character appears in italics, it represents a variable that is replaced with a literal value in an actual computation. For example,

`sqrt(`*x*`)`

means take the square root of any floating-point value *x*, such as 1.45 or 2.789.

When a character appears in italics in one of the tables for special cases in Chapter 5, it represents a nonzero, finite floating-point number.

### Types of Notes

There are several types of notes used in *Inside Macintosh*.

**Note**
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

**IMPORTANT**
A note like this contains information that is essential for an understanding of the main text. ▲

▲ **WARNING**
Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. ▲

## For More Information

See http://developer.apple.com and, in particular, http://developer.apple.com/samplecode/Sample_Code/Runtime_Architecture.htm for more information on Mac OS X and the implementation of its numerics environment.

# The IEEE Standard Numerics Environment

This part is a general description of IEEE standard numerics. Chapter 1 describes the standards for floating-point arithmetic that the IEEE standard specifies and discusses why the standard is important. If you are unfamiliar with how computers perform floating-point arithmetic, you should read Chapter 1. Chapters 2 through 5 describe how the standard is implemented. They describe the basic features shared by all IEEE standard numerics implementations, including

- the numeric data formats
- the special values NaN (Not-a-Number) and Infinity
- the methods by which floating-point expressions are evaluated
- environmental controls, such as setting the rounding direction and handling exceptions
- conversions between the different numeric formats
- operations supported by IEEE standard numerics

Although Part 1 uses the C programming language in its examples, many of the facilities of the the Mac OS X implementation of the IEEE standard numerics (the Libm math library) are accessible to users of virtually any high-level programming language, as well as to assembly-language programmers.

# IEEE Standard Arithmetic

## Contents

# IEEE Standard Arithmetic

This chapter describes why IEEE standard floating-point arithmetic is important and why you should use it when programming. The Libm math library is an implementation of the IEEE Standard 754 for binary floating-point arithmetic (augmented by the IEEE Standard 9899 for C language numerics). This chapter explains the benefits that the Libm math library provides by conforming to this standard. It provides an overview of the IEEE standard—describing the scope of the standard and explaining how following it improves the accuracy of your programs. It provides some examples to demonstrate how much easier programming is when the standard is followed.

You should read this chapter if you are unfamiliar with IEEE Standard 754 and you want to find out more about it. If you are already familiar with this standard but you would like to find out how the Libm math library implements them, you can skip to the next chapter.

## About the IEEE Standard

**The Libm math library** in Mac OS X is a floating-point environment that complies with IEEE Standard 754 (as well as the IEEE Standard 9899). There is another IEEE standard for floating-point arithmetic, the **IEEE Standard 854** for radix-independent floating-point arithmetic, which we do not deal with in this document.

The IEEE standard ensures that computers represent real numbers as accurately as possible and that computers perform arithmetic on real numbers as accurately as possible. Although there are infinitely many real numbers, a computer can represent only a finite number of them. Computers represent real numbers as **binary floating-point numbers**. Binary floating-point numbers can represent real numbers exactly in relatively few cases; in all other cases the representation is approximate. For example, 1/2 (0.5 in decimal) can be represented exactly in binary as 0.1. Other real numbers that can be represented exactly in decimal have repeating digits in binary and hence cannot be represented exactly, as shown in Table 1-1. For example, 1/10, or decimal 0.1 exactly, is 0.000110011 . . . in binary. Errors of this kind are unavoidable in any computer approximation of real numbers. Because of these errors, sums of fractions are often slightly incorrect. For example, 4/3 – 5/6 is not exactly equal to 1/2 on any computer, even on computers that use IEEE standard arithmetic.

The IEEE standard defines data formats for floating-point numbers, shows how to interpret these formats, and specifies how to perform operations (known as **floating-point operations**) on numbers in these formats. It requires the following types of floating-point operations:

- basic arithmetic operations (add, subtract, multiply, divide, square root, remainder, and round-to-integer)

**Table 1-1** Approximation of real numbers

| Fraction | Decimal approximation* | Binary approximation† |
|---|---|---|
| 1/10 | 0.1000000000‡ | 0.000110011001100110011001101 |
| 1/2 | 0.5000000000‡ | 0.100000000000000000000000‡ |
| 4/3 | 1.333333333 | 1.01010101010101010101011 |
| 5/6 | 0.8333333333 | 0.110101010101010101010101 |
| 4/3 – 5/6 | 0.4999999997 | 0.100000000000000000000001 |

* 10 significant decimal digits
† 23 significant binary digits
‡ Exact value

- conversion operations, which convert numbers to and from the floating-point data formats
- comparison operations, such as less than, greater than, and equal to
- environmental control operations, which manipulate the floating-point environment

The IEEE standard requires that the basic arithmetic operations have the following attributes:

- The result must be accurate in the precision in which the operation is performed. When a numerics environment is performing a floating-point operation, it calculates the result to a predetermined number of binary digits. This number of digits is called the **precision.** The result must be correct to the last binary digit.
- If the result cannot be represented exactly in the destination data format, it must be changed to the closest value that can be represented, using **rounding.** See the section "Careful Rounding" on page 1-4 for more information on why careful rounding is important.
- If an invalid input is provided or if the result cannot be represented exactly, a floating-point **exception** must be raised. See the section "Exception Handling" on page 1-6 for a description of why exception handling is important in floating-point arithmetic.

## Careful Rounding

If the result of an IEEE arithmetic operation cannot be represented exactly in binary format, the number is rounded. IEEE arithmetic normally rounds results to the nearest value that can be represented in the chosen data format. The difference between the exact result and the represented result is the **roundoff error.**

The IEEE standard requires that users be able to choose to round in directions other than to the nearest value. For example, sometimes you might want to know that rounding has not invalidated a computation. One way to do that would be to force the rounding direction so that you can be sure your results are higher (or lower) than the exact answer. Because it conforms to the IEEE standard, Mac OS X numerics gives you a means of doing that. Fully developed, this strategy is called *interval arithmetic* (Kahan 1980). For complete details on rounding directions, see Chapter 3, "Environmental Controls."

The following example is a simple demonstration of the advantages of careful rounding. Suppose your application performs operations that are mutually inverse; that is, operations $y = f(x)$, $x = g(y)$, such that $g(f(x)) = x$. There are many such operations, such as

$$y = x^2, \quad x = \sqrt{y}$$

$$y = 375x, \quad x = y/375$$

Suppose $F(x)$ is the computed value of $f(x)$, and $G(y)$ is the computed value of $g(y)$. Because many numbers cannot be represented exactly in binary, the computed values $F(x)$ and $G(y)$ will often differ from $f(x)$ and $g(y)$. Even so, if both functions are continuous and well behaved, and if you always round $F(x)$ and $G(y)$ to the nearest value, you might expect your computer arithmetic to return *x* when it performs the cycle of inverse operations, $G(F(x))$. It is difficult to predict when this relation will hold for computer numbers. Experience with other computers says it is too much to expect, but IEEE arithmetic very often returns the correct inverse value.

The reason for IEEE arithmetic's good behavior with respect to inverse operations is that it rounds so carefully. Even with all operations in, say, single precision, it evaluates the expression $3 \times 1/3$ to 1.0 exactly; some computers that do not follow the standards do not evaluate this expression exactly. If you find that surprising, you might enjoy running the code example in Listing 1-1 on a computer that does not use IEEE arithmetic and then on a computer using Mac OS X. The default rounding provided by the numerics environment gives good results; the Mac OS X computer prints "No failures." The program will fail on a computer that doesn't have IEEE arithmetic—in particular, that doesn't round halfway cases in the same way that the IEEE standard's default rounding direction mode does.

**Listing 1-1** Inverse operations

```
#include <stdio.h>
main()
{
   float x, y, a, b;
   int ix, iy,
   int nofail = 1;      /* Boolean, initialized to true */

   for (ix = 1; ix <= 12; ix++) {
      if ((ix != 7) && (ix != 11)) {      /* x is a sum of powers of two */
         for (iy = 1; iy <= 50; iy++) {
            x = ix;
            y = iy;
            a = y / x;
```

```
                b = x * a;          /* b == (x * y / x) == y */
                if (b != y) {
                   nofail = 0;      /* false */
                   printf("It failed for x = %d, y = %d\n", ix, iy);
                }
             }
          }
       }
       if (nofail) printf("No failures\n");
    }
```

## Exception Handling

The IEEE standard defines five exceptions that indicate when an exceptional event has occurred. They are

- invalid operation
- underflow
- overflow
- division by zero
- inexact result

There are three ways your application can deal with exceptions:

- Continue operation.
- Stop on exceptions if you think they will invalidate your results.
- Include code to do something special when exceptions happen.

The IEEE standard lets programs deal with the exceptions in reasonable ways. It defines the special values NaN (Not-a-Number) and Infinity, which allow a program to continue operation; see the section “Interpreting Floating-Point Values” on page 2-4 in “Floating-Point Data Formats”. The IEEE standard also defines exception flags, which a program can test to detect exceptional events.

IEEE arithmetic allows the option to stop computation when exceptional events arise; see “Trapping Floating-point Exceptions” on page 7-14 for details. But there are good reasons why you might prefer not to have to stop. The following examples illustrate some of those reasons.

### Example: Finding Zero Return Values

Suppose you want to find the first positive integer that causes a function to cross the x-axis. A simple version of the code might look like this:

```
for (i = 0; i < MAXVALUE; i++)
   if (func(i) == 0)
      printf("It crosses when x = %g\n", i);
```

Further, suppose that `func` was defined like this:

```
double func(double x)
{
   return(sqrt(x - 3));
}
```

The intent of the `for` loop is to find out where the function crosses the x-axis and print out that information; it does not really care about the value returned from `func` unless the value is 0. However, this loop will fail when `i` is less than 3 because you cannot take the square root of a negative number. With a C compiler that supports the IEEE standard, performing the square root operation on a negative number returns a NaN, allowing the loop to produce the desired result. To obtain the desired result on all computers, something more cumbersome would have to be written. By allowing the square root of a negative number, the IEEE standard allows more straightforward code.

This program fragment demonstrates the principal service performed by NaNs: they permit deferred judgments about variables whose values might be unavailable (that is, uninitialized) or the result of invalid operations. Instead of having the computer stop a computation as soon as a NaN appears, you might prefer to have it continue if whatever caused the NaN is irrelevant to the solution.

## Example: Searching Without Stopping

Suppose a program has to search through a database for a maximum value that has to be calculated. The search loop might call a subroutine to perform some calculation on the data in each record and return a value for the program to test or compare. The code might look like this:

```
max = –INFINITY;
for (i = 0; i < MAXRECORDS; i++)
   if((temp = computation(record[i].value)) > max)
      max = temp;
```

Suppose that the `value` field of the `record` structure is not a required field when the data is entered, so that for some records, data might be nonexistent or invalid. In many machines, that would cause the program to stop. To avoid having the program stop during the search, you would have to add tests for all the exceptional cases. With IEEE standard numerics, the subroutine `computation` does not stop for nonexistent or invalid data; it simply returns a NaN.

This is another example of the way arithmetic that includes NaNs allows the program to ignore irrelevancies, even when they cause invalid operations. Using arithmetic without NaNs, you would have to anticipate all exceptional cases and add code to the program to handle every one of them in advance. With NaNs, you can handle all exceptional cases after they have occurred, or you can simply ignore them, as in this example.

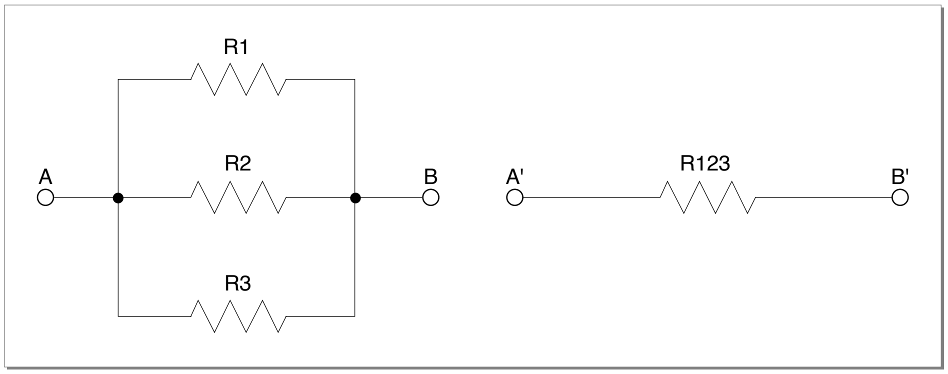
### Example: Parallel Resistances

Like NaNs, Infinities enable the program to handle cases that otherwise would require special programming to keep from stopping. Here is an example where arithmetic with Infinities is entirely reasonable.

When three electrical resistances R1, R2, and R3 are connected in parallel, as shown in Figure 1-1, their effective resistance is the same as a single resistance whose value R123 is given by this formula:

$$R123 = \frac{1}{\frac{1}{R1} + \frac{1}{R2} + \frac{1}{R3}}$$

**Figure 1-1** Parallel resistances



The formula gives correct results for positive resistance values between 0 (corresponding to a short circuit) and ∞ (corresponding to an open circuit) inclusive. On computers that do not allow division by zero, you would have to add tests designed to filter out the cases with resistance values of zero. (Negative values can cause trouble for this formula, regardless of the style of the arithmetic, but that reflects their troublesome nature in circuits, where they can cause instability.)

Arithmetic with Infinities usually gives reasonable results for expressions in which each independent variable appears only once.

## Using IEEE Arithmetic

This section provides some example computations and describes how using IEEE arithmetic with Mac OS X makes programming these computations easier.

## Evaluating Continued Fractions

Consider a typical continued fraction $\mathrm{cf}(x)$ .

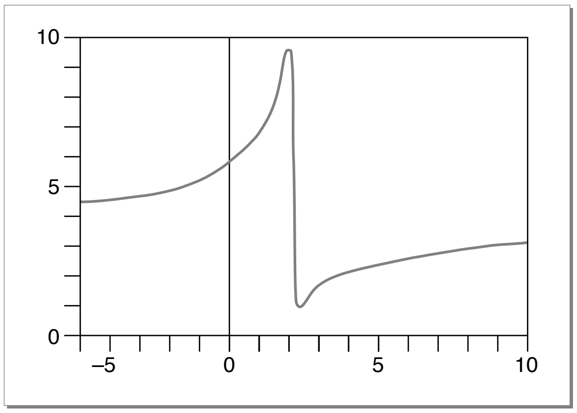
$$\mathrm{cf}(x) = 4 - \cfrac{3}{x - 2 - \cfrac{1}{x - 7 + \cfrac{10}{x - 2 - \cfrac{2}{x - 3}}}}$$

An algebraically equivalent expression is $\mathrm{rf}(x)$ :

$$\mathrm{rf}(x) = \frac{622 - x(751 - x(324 - x(59 - 4x)))}{112 - x(151 - x(72 - x(14 - x)))}$$

Both expressions represent the same rational function, one whose graph is smooth and unexceptional, as shown in Figure 1-2.

**Figure 1-2** Graph of continued fraction functions $\mathrm{cf}(x)$ and $\mathrm{rf}(x)$



Although the two functions $\mathrm{rf}(x)$ and $\mathrm{cf}(x)$ are equal, they are not computationally equivalent. For instance, consider $\mathrm{rf}(x)$ at the following values of $x$:

| | |
|---|---|
| $x = 1$ | $\mathrm{rf}(1) = 7$ |
| $x = 2$ | $\mathrm{rf}(2) = 4$ |
| $x = 3$ | $\mathrm{rf}(3) = 8/5$ |
| $x = 4$ | $\mathrm{rf}(4) = 5/2$ |

Whereas $\mathrm{rf}(x)$ is perfectly well behaved, those values of $x$ lead to division by zero when computing $\mathrm{cf}(x)$ and cause many computers to stop. In IEEE standard arithmetic, division by zero produces an Infinity. Therefore, Mac OS X has no difficulty in computing $\mathrm{cf}(x)$ for those values.

On the other hand, simply computing $\mathrm{rf}(x)$ instead of $\mathrm{cf}(x)$ can also cause problems. If the absolute value of $x$ is so big that $x^4$overflows the chosen data format, then $\mathrm{cf}(x)$ approaches $\mathrm{cf}(\infty) = 4$ but computing $\mathrm{rf}(x)$ encounters $(\text{overflow})/(\text{underflow})$, which yields something else. Mac OS X returns NaN for such cases; some other machines return $(\text{maximum value})/(\text{maximum value}) = 1$. Also, at arguments $x$ between 1.6 and 2.4, the formula $\mathrm{rf}(x)$ suffers from roundoff error much more than $\mathrm{cf}(x)$ does. For those reasons, computing $\mathrm{cf}(x)$ is preferable to computing $\mathrm{rf}(x)$ if division by zero works the way it does in Mac OS X, that is, if it produces Infinity instead of stopping computation.

In general, division by zero is an exceptional event not merely because it is rare but because different applications require different consequences. If you are not satisfied with the consequences supplied by the default Mac OS X environment, you can choose other consequences by making the program test for NaNs and Infinities (or for the flags that signal their creation).

Rather than sprinkle tests throughout the program in an attempt to keep exceptions from occurring, you might prefer to put one or two tests near the end of the code to detect the (rare) occurrence of an exception and modify the results appropriately. That is more economical than testing every divisor for zero (since zero divisors are rare).

## Computing the Area of a Triangle

Here is a familiar and straightforward task that fails when subtraction is aberrant: Compute the area $A(x, y, z)$ of a triangle given the lengths $x, y, z$ of its sides. The formula given here performs this calculation almost as accurately as its individual floating-point operations are performed by the computer it runs on, provided the computer does not drop digits prematurely during subtraction. The formula works correctly, and provably so, on a wide range of machines, including all implementations of the IEEE standards.

The classical formula, attributed to Heron of Alexandria, is

$$A(x, y, z) = \sqrt{s(s-x)(s-y)(s-z)}$$

where $s = (x+y+z)/2$.

For needle-shaped triangles, that formula gives incorrect results on computers *even when every arithmetic operation is correctly rounded*. For example, Table 1-2 shows an extreme case with results rounded to five decimal digits. With the values shown, rounded $(x+(y+z))/2$ must give either 100.01 or 100.02. Substituting those values for $s$ in Heron's formula yields either 0.0 or 1.5813 instead of the correct value 1.000025.

Evidently, Heron's formula would be a very bad way for computers to calculate ratios of areas of nearly congruent needle-shaped triangles.

**Table 1-2** Area using Heron's formula

| | Correct | Rounding downward | Roundin g upward |
|---|---|---|---|
| $x$ | 100.01 | 100.01 | 100.01 |
| $y$ | 99.995 | 99.995 | 99.995 |
| $z$ | 0.025 | 0.025 | 0.025 |
| $(x+(y+z))/2$ | 100.015 | 100.01 | 100.02 |
| $A$ | 1.000025 | 0.0000 | 1.5813 |

A good procedure, numerically stable on machines that do not truncate prematurely during subtraction (such as machines that use IEEE arithmetic), is the following:

1. Sort $x, y, z$ so that $x \geq y \geq z$.
2. Test for $z \geq x - y$ to see whether the triangle exists.
3. Compute $A$ by the formula

$$A = \sqrt{((x+(y+z))(z-(x-y))(x+(y-z)))/4}$$

▲ **WARNING**
This formula works correctly only if you do not remove any of the parentheses. ▲

The success of the formula depends upon the following easily proved theorem:

THEOREM *If p and q are represented exactly in the same conventional floating-point format, and if* $1/2 \leq p/q \leq 2$, *then* $p - q$ *too is representable exactly in the same format (unless* $p - q$ *suffers underflow, something that cannot happen in IEEE arithmetic).*

The theorem merely confirms that subtraction is exact when massive cancellation occurs. That is why each factor inside the square root expression is computed correctly to within a unit or two in its last digit kept, and $A$ is not much worse, on computers that subtract the way ta PowerPC microprocessor does. On machines that flush tiny results to zero, this formula for $A$ fails because $(p - q)$ can underflow.

## About the FPCE Technical Report

Even though many computers now conform to the IEEE standard, the standard has suffered from a lack of high-level portability. The reason is that the standard does not define bindings to high-level languages; it only defines a programming environment. For instance, the standard defines data formats that should be supported but does not tell how these data formats should map to variable types in high-level languages. It also specifies that the user must be able to control rounding direction but falls short of defining how the user is able to do so.

However, the definition of a binding is in progress for the C programming language. The Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group (NCEG), or **ANSI X3J11.1,** has proposed a general floating-point specification for the C programming language, called the **FPCE technical report,** that contains additional specifications for implementations that comply with IEEE floating-point standards 754 and 854.

The FPCE technical report not only specifies how to implement the requirements of the IEEE standards, but also requires some additional functions, called **transcendental functions** (sometimes called *elementary functions*). These functions are consistent with the IEEE standard and can be used as building blocks in numerical functions. The transcendental functions include the usual logarithmic and exponential functions, as well as $\ln(1 + x)$ and $e^x - 1$; financial functions for compound interest and annuity calculations; trigonometric functions; error and gamma functions; and a random number generator. The **PowerPC Numerics library,** contained in the file MathLib, implements the transcendental functions.

Part 2 of this book describes how PowerPC Numerics complies with the recommendations in the FPCE technical report.

# Floating-Point Data Formats

## Contents

# Floating-Point Data Formats

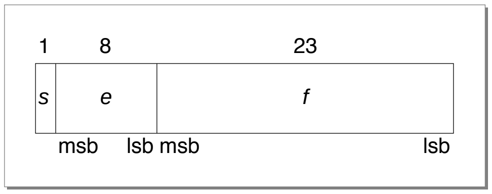
This chapter describes the data formats your Mac OS X application can use to represent floating-point numbers. It begins by discussing in general the methods Libm uses to store and interpret floating-point values and by explaining why those methods were chosen. The chapter introduces the special values zero, NaN (Not-a-Number), and Infinity and explains why these special values are necessary. Next is an in-depth description of the numeric data formats with a discussion of how these formats represent floating-point values. At the end of the chapter, you will find a table comparing the size, range, and precision of the numeric data formats. This table can help you choose which data format is best for your application.

You should read this chapter to learn about the floating-point data formats available on Macintosh computers using Mac OS X and to learn more about how your computer encodes and manipulates floating-point numbers.

## About Floating-Point Data Formats

The IEEE standard defines several floating-point data formats, one required and the others recommended. IEEE requires that each data format have a **sign bit** (*s*), an **exponent** field (*e*), and a **fraction** field (*f*). For each format, it lists requirements for the minimum lengths of these fields. For example, the standard describes a 32-bit single format whose exponent field must be 8 bits long and whose fraction field must be 23 bits long. Figure 2-1 shows the IEEE requirements for the single format. (In this figure, *msb* stands for *most significant bit* and *lsb* stands for *least significant bit.*)

**Figure 2-1** IEEE single format



The only required data format is the 32-bit single format. A 64-bit double format is strongly recommended. The IEEE standard also describes two data formats called *single-extended* and *double-extended* and recommends that floating-point environments provide the extended format corresponding to the widest basic format (single or double) they support.

To conform to the IEEE requirements on floating-point data formats, Mac OS X provides two data formats: single (32 bits) and double (64 bits). The single and double formats are implemented exactly as described in the standard.

Table 2-1 shows how the two numeric data formats correspond to C variable types. For more information about data types in C, refer to “Numeric Data Types in C” on page 6-3.

**Table 2-1** Names of data types

| **PowerPC Numerics data format** | **C type** |
|---|---|
| IEEE single | `float` |
| IEEE double | `double` |

The IEEE standard also makes requirements about how the values in these data formats are interpreted. Mac OS X follows these requirements exactly. They are described in the next section.

## Interpreting Floating-Point Values

Regardless of which data format (single or double) you use, the numerics environment uses the same basic method to interpret which floating-point value the data format represents. This section describes that method.

Every floating-point data format has a sign bit, an exponent field, and a fraction field. These three fields provide binary encodings of a sign (+ or –), an exponent, and a **significand,** respectively, of a floating-point value. The value is interpreted as

$$\pm\ significand \times 2^{exponent - bias}$$

where

| | |
|---|---|
| ± | is the sign stored in the sign bit (1 is negative, 0 is positive). |
| *significand* | has the form $b_0 . b_1 b_2 b_3 \ldots b_{precision-1}$ where $b_1 b_2 b_3 \ldots b_{precision-1}$ are the bits in the fraction field and $b_0$ is an implicit bit whose value is interpreted as described in the sections “Normalized Numbers” and “Denormalized Numbers.” The significand is sometimes called the *mantissa*. |
| *exponent* | is the value of the exponent field. |
| *bias* | is the bias of the exponent. The **bias** is a predefined value (127 for single format, 1023 for double formats) that is added to the exponent when it is stored in the exponent field. When the floating-point number is evaluated, the bias is subtracted to return the correct exponent. The minimum biased exponent field (all 0’s) and maximum biased exponent field (all 1’s) are assigned special floating-point values (described in the next several sections). |

In a numeric data format, each valid representation belongs to exactly one of these classes, which are described in the sections that follow:

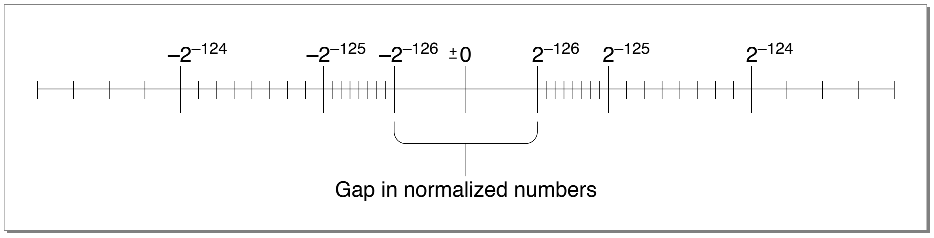
- normalized numbers
- denormalized numbers
- Infinities
- NaNs (signaling or quiet)
- zeros

## Normalized Numbers

The numeric data formats represent most floating-point numbers as **normalized numbers,** meaning that the implicit leading bit ($b_0$ on page 2-4) of the significand is 1. Normalization maximizes the resolution of the data type and ensures that representations are unique. Figure 2-2 shows the magnitudes of normalized numbers in single precision on the number line. The spacing of the vertical marks indicates the relative density of numbers in each binade. (A **binade** is a collection of numbers between two successive powers of 2.) Notice that the numbers get more dense as they approach 0.

**Note**
The figure shows only the relative density of the numbers; in reality, the density is immensely greater than it is possible to show in such a figure. For example, there are $2^{23}$ (8,388,608) single-precision numbers in the interval $2^{-126} \le x < 2^{-125}$. ◆

**Figure 2-2** Normalized single-precision numbers on the number line



Using only normalized representations creates a gap around the value 0, as shown in Figure 2-2. If a computer supports only the normalized numbers, it must round all tiny values to 0. For example, suppose such a computer must perform the operation $x - y$, where $x$ and $y$ are very close to, but not equal to, each other. If the difference between $x$ and $y$ is smaller than the smallest normalized number, the computer must deliver 0 as the result. Thus, for such **flush-to-zero systems,** the following statement is *not* true for all real numbers:

$x - y = 0$ if and only if $x = y$

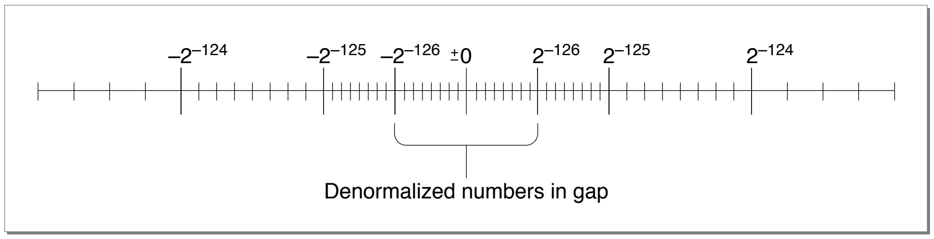
## Denormalized Numbers

Instead of using only normalized numbers and allowing this small gap around 0, Libm uses **denormalized numbers,** in which the leading implicit bit ($b_0$ on page 2-4) of the significand is 0 and the minimum exponent is used.

**Note**
Some references use the term **subnormal numbers** instead of *denormalized numbers.* ◆

Figure 2-3 illustrates the relative magnitudes of normalized and denormalized numbers in single precision. Notice that the denormalized numbers have the same density as the numbers in the smallest normalized binade. This means that the roundoff error is the same regardless of whether an operation produces a denormalized number or a very small normalized number. As stated previously, without denormalized numbers, operations would have to round tiny values to 0, which is a much greater roundoff error.

**Figure 2-3** Denormalized single-precision numbers on the number line



To put it another way, the use of denormalized numbers makes the following statement true for all real numbers:

$x - y = 0$ if and only if $x = y$

Another advantage of denormalized numbers is that error analysis involving small values is much easier without the gap around zero shown in Figure 2-2 (Demmel 1984).

The computer determines that a floating-point number is denormalized (and therefore that its implicit leading bit is interpreted as 0) when the biased exponent field is filled with 0's and the fraction field is nonzero.

Table 2-2 shows how a single-precision value $A_0$ becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called **gradual underflow.** In the table, values $A_2 \ldots A_{25}$ are denormalized; $A_{25}$ is the smallest positive denormalized number in single format. Notice that as soon as the values are too small to be normalized, the biased exponent value becomes 0.

**Table 2-2** Example of gradual underflow

| Variable or operation | Value | Biased exponent | Comment |
|---|---|---|---|
| $A_0$ | $1.100\ 1100\ 1100\ 1100\ 1100\ 1101 \times 2^{-125}$ | 2 | |
| $A_1 = A_0/2$ | $1.100\ 1100\ 1100\ 1100\ 1100\ 1101 \times 2^{-126}$ | 1 | |
| $A_2 = A_1/2$ | $0.110\ 0110\ 0110\ 0110\ 0110\ 0110 \times 2^{-126}$ | 0 | Inexact* |
| $A_3 = A_2/2$ | $0.011\ 0011\ 0011\ 0011\ 0011\ 0011 \times 2^{-126}$ | 0 | Exact result |
| $A_4 = A_3/2$ | $0.001\ 1001\ 1001\ 1001\ 1001\ 1010 \times 2^{-126}$ | 0 | Inexact* |
| | . | | |
| | . | | |
| | . | | |
| $A_{23} = A_{22}/2$ | $0.000\ 0000\ 0000\ 0000\ 0000\ 0011 \times 2^{-126}$ | 0 | Exact result |
| $A_{24} = A_{23}/2$ | $0.000\ 0000\ 0000\ 0000\ 0000\ 0010 \times 2^{-126}$ | 0 | Inexact* |
| $A_{25} = A_{24}/2$ | $0.000\ 0000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126}$ | 0 | Exact result |
| $A_{26} = A_{25}/2$ | 0.0 | 0 | Inexact* |

* Whenever division returns an inexact tiny value, the exception bit for underflow is set to indicate that a low-order bit has been lost.

## Infinities

An **Infinity** is a special bit pattern that can arise in one of two ways:

- When an operation (such as $1/0$) should produce a mathematical infinity, the result is an Infinity.
- When an operation attempts to produce a number with a magnitude too great for the number's intended floating-point data type, the result might be a value with the largest possible magnitude or it might be an Infinity (depending on the current rounding direction).

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The Infinities, one positive and one negative, generally behave as suggested by the theory of limits. For example:

- Adding 1 to $+\infty$ yields $+\infty$.
- Dividing $-1$ by $+0$ yields $-\infty$ .
- Dividing 1 by $-\infty$ yields $-0$ .

The computer determines that a floating-point number is an Infinity if its exponent field is filled with 1's and its fraction field is filled with 0's. So, for example, in single format, if the sign bit is 1, the exponent field is 255 (which is the maximum biased exponent for the single format), and the fraction field is 0, the floating-point number represented is $-\infty$ (see Figure 2-4).

**Figure 2-4** Infinities represented in single precision

| | Hexadecimal | Binary | | |
|---|---|---|---|---|
| $+\infty$ | 7F800000 | 0 | 1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| $-\infty$ | FF800000 | 1 | 1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

## NaNs

When a numeric operation cannot produce a meaningful result, the operation delivers a special bit pattern called a **NaN (Not-a-Number).** For example, zero divided by zero, $+\infty$ added to $-\infty$ , and $\sqrt{-1}$ yield NaNs. A NaN can occur in any of the numeric data formats (single or double), but generally, system-specific **integer types** (non-numeric types exclusively for integer values) have no representation for NaNs.

NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN. If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: **quiet NaNs,** the usual kind produced by floating-point operations, and **signaling NaNs.**

When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and a quiet NaN is the delivered result. Signaling NaNs are not created by any numeric operations, but you might find it useful to create signaling NaNs manually. For example, you might fill uninitialized memory with signaling NaNs so that if one is ever encountered in a program, you will know that uninitialized memory is accessed.

A NaN may have an associated code that indicates its origin. These codes are listed in Table 2-3. The NaN code is the 8th through 15th most significant bits of the fraction field.

**Table 2-3** NaN codes

| Decimal | Hexadecimal | Meaning |
|---|---|---|
| 1 | 0x00 | Invalid square root, such as $\sqrt{-1}$ |
| 2 | 0x00 | Invalid addition, such as $(+\infty) + (-\infty)$ |
| 4 | 0x00 | Invalid division, such as $0/0$ |
| 8 | 0x00 | Invalid multiplication, such as $0 \times \infty$ |
| 9 | 0x09 | Invalid remainder or modulo, such as $x$ rem 0 |
| 17 | 0x11 | Attempt to convert invalid ASCII string |
| 21 | 0x15 | Attempt to create a NaN with a zero code |
| 33 | 0x21 | Invalid argument to trigonometric function (such as cos, sin, tan) |
| 34 | 0x22 | Invalid argument to inverse trigonometric function (such as acos, asin, atan) |
| 36 | 0x24 | Invalid argument to logarithmic function (such as log, $\log 10$ ) |
| 37 | 0x25 | Invalid argument to exponential function (such as exp, expm1) |
| 38 | 0x26 | Invalid argument to financial function (compound or annuity) |
| 40 | 0x28 | Invalid argument to inverse hyperbolic function (such as acosh, asinh) |
| 42 | 0x2A | Invalid argument to gamma function (gamma or lgamma) |

**Note**
The first four cases are computed directly by the PowerPC processor, which always returns 0 for the NaN code. The remaining cases make use of Libm code, which sets the appropriate non-zero NaN code.

The computer determines that a floating-point number is a NaN if its exponent field is filled with 1's and its fraction field is nonzero. The most significant bit of the fraction field distinguishes quiet and signaling NaNs. It is set for quiet NaNs and clear for signaling NaNs. For example, in single format, if the sign field has the value 1, the exponent field has the value 255, and the fraction field has the value 65,280, then the number is a signaling NaN. If the sign is 1, the exponent is 255, and the fraction field has the value 4,259,584 (which means the fraction field has a leading 1 bit), the value is a quiet NaN. Figure 2-5 illustrates these examples.

**Figure 2-5** NaNs represented in single precision

| | Hexadecimal | Binary | | |
|---|---|---|---|---|
| Signaling NaN | FF80FF00 | 1 | 1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 |
| Quiet NaN | FFC0FF00 | 1 | 1 1 1 1 1 1 1 1 | 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 |

## Zeros

Each floating-point format has two representations for zero: +0 and $-0$. Although the two zeros compare as equal $(+0) = -0$, their behaviors in IEEE arithmetic are slightly different.

Ordinarily, the sign of zero does not matter except (possibly) for a function discontinuous at zero. Though the two forms are numerically equal, a program can distinguish +0 from $-0$ by operations such as division by zero or by performing the numeric copysign function.

The sign of zero obeys the usual sign laws for multiplication and division. For example, $(+0) \times (-1) = -0$ and $1/(-0) = -\infty$. Because extreme negative underflows yield $-0$, expressions like $1/x^3$ produce the correct sign for $\infty$ when $x$ is tiny and negative. Addition and subtraction produce $-0$ only in these cases:

- $(-0) - (+0)$ yields $-0$
- $(-0) + (-0)$ yields $-0$

When rounding downward, with $x$ finite,

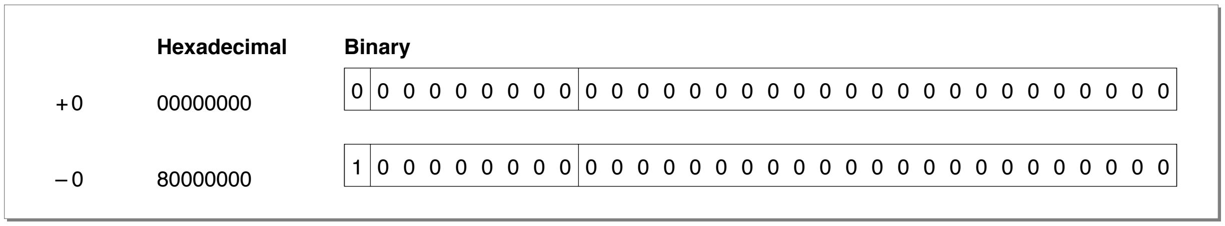
- $x - x$ yields $-0$
- $x + (-x)$ yields $-0$

The square root of $-0$ is $-0$.

The sign of zero is important in complex arithmetic (Kahan 1987).

The computer determines that a floating-point number is 0 if its exponent field and its fraction field are filled with 0's. For example, in single format, if the sign bit is 0, the exponent field is 0, and the fraction field is 0, the number is +0 (see Figure 2-6).

**Figure 2-6** Zeros represented in single precision



# Formats

This section shows the three numeric data formats: single, double. These are pictorial representations and might not reflect the actual byte order in any particular implementation.

Each of the diagrams on the following pages is followed by a table that gives the rules for evaluating the number. In each field of each diagram, the leftmost bit is the most significant bit (msb) and the rightmost is the least significant bit (lsb). Table 2-4 defines the symbols used in the diagrams.

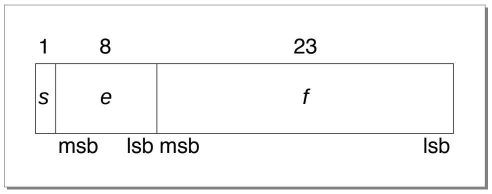
**Table 2-4** Symbols used in format diagrams

| Symbol | Description |
|---|---|
| *v* | Value of number |
| *s* | Sign bit |
| *e* | Biased exponent (*exponent* + *bias*) |
| *f* | Fraction (*significand* without leading bit) |

## Single Format

The 32-bit **single format** is divided into three fields having 1, 8, and 23 bits (see Figure 2-7).

**Figure 2-7** Single format



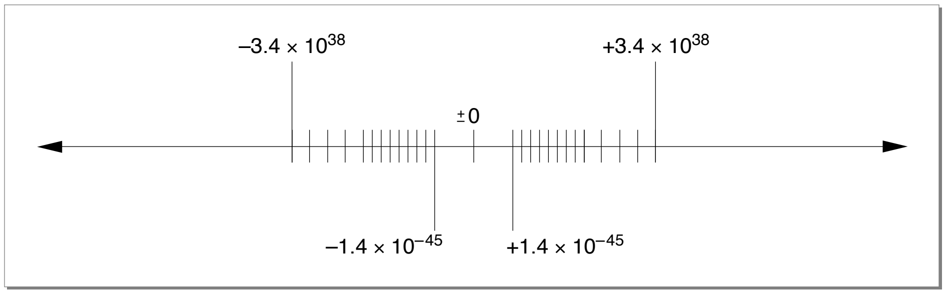
The interpretation of a single-format number depends on the values of the exponent field (*e*) and the fraction field (*f*), as shown in Table 2-5.

**Table 2-5** Values of single-format numbers (32 bits)

| If biased exponent *e* is: | And fraction *f* is: | Then value *v* is: | And the class of *v* is: |
|---|---|---|---|
| $0 < e < 255$ | (any) | $v = (-1)^s \times 2^{(e-127)} \times (1.f)$ | Normalized |
| $e = 0$ | $f \neq 0$ | $v = (-1)^s \times 2^{(-126)} \times (0.f)$ | Denormalized |
| $e = 0$ | $f = 0$ | $v = (-1)^s \times 0$ | Zero |
| $e = 255$ | $f = 0$ | $v = (-1)^s \times \infty$ | Infinity |
| $e = 255$ | $f \neq 0$ | $v$ is a NaN | NaN |

Figure 2-8 shows the range and density of the real numbers that can be represented as single-format floating-point numbers using normalized and denormalized values. The vertical marks indicate the relative density of the numbers that can be represented. As explained in the section "Normalized Numbers" on page 2-5, the number of representable values gets more dense closer to 0.

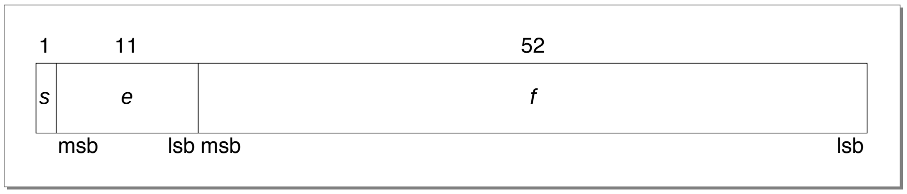
**Figure 2-8** Single-format floating-point numbers on the real number line



## Double Format

The 64-bit **double format** is divided into three fields having 1, 11, and 52 bits (see Figure 2-9).

**Figure 2-9** Double format



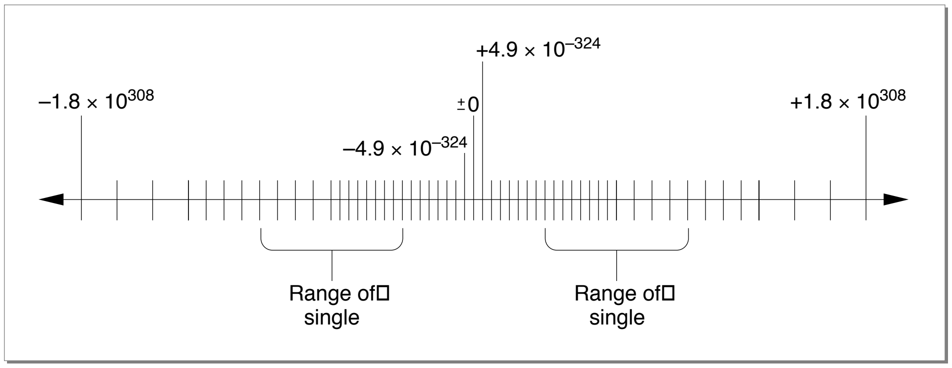
The interpretation of a double-format number depends on the values of the exponent field (*e*) and the fraction field (*f*), as shown in Table 2-6.

**Table 2-6** Values of double-format numbers (64 bits)

| If biased exponent *e* is: | And fraction *f* is: | Then value *v* is: | And the class of *v* is: |
|---|---|---|---|
| $0 < e < 2047$ | (any) | $v = (-1)^s \times 2^{(e-1023)} \times (1.f)$ | Normalized |
| $e = 0$ | $f \neq 0$ | $v = (-1)^s \times 2^{(-1022)} \times (0.f)$ | Denormalized |
| $e = 0$ | $f = 0$ | $v = (-1)^s \times 0$ | Zero |
| $e = 2047$ | $f = 0$ | $v = (-1)^s \times \infty$ | Infinity |
| $e = 2047$ | $f \neq 0$ | $v$ is a NaN | NaN |

Figure 2-10 shows the range and density of the real numbers that can be represented as double-format floating-point numbers using normalized and denormalized values. The vertical marks indicate the relative density of the numbers that can be represented. As explained in the section “Normalized Numbers” on page 2-5, the number of representable values gets more dense closer to 0.

**Figure 2-10** Double-format floating-point values on the real number line



## Range and Precision of Data Formats

Table 2-7 shows the precision, range, and memory usage for each numeric data format. You can use this table to compare the data formats and choose which one is needed for your application. Typically, choosing a data format requires that you determine the tradeoffs between

- fixed-point or floating-point form
- precision
- range
- memory usage
- speed

In the table, decimal ranges are expressed as rounded, two-digit decimal representations of the exact binary values. The speed of a given data format varies depending on the particular implementation of IEEE standard numerics. (See Chapter 4, "Conversions," for information on aspects of conversion relating to precision.)

**Table 2-7** Summary of PowerPC Numerics data formats

| | Single | Double |
|---|---|---|
| Size (bytes:bits) | 4:32 | 8:64 |
| Range of binary exponents | | |
| Minimum | –126 | –1022 |
| Maximum | 127 | 1023 |
| Significand precision | | |

**Table 2-7** Summary of PowerPC Numerics data formats

| | **Single** | **Double** |
|---|---|---|
| Bits | 24 | 53 |
| Decimal digits | 7–8 | 15–16 |
| Decimal range (approximate) | | |
| Maximum positive | $3.4 \times 10^{+38}$ | $1.8 \times 10^{+308}$ |
| Minimum positive norm | $1.2 \times 10^{-38}$ | $2.2 \times 10^{-308}$ |
| Minimum positive denorm | $1.4 \times 10^{-45}$ | $4.9 \times 10^{-324}$ |
| Maximum negative denorm | $-1.4 \times 10^{-45}$ | $-4.9 \times 10^{-324}$ |
| Maximum negative norm | $-1.2 \times 10^{-38}$ | $-2.2 \times 10^{-308}$ |
| Minimum negative | $-3.4 \times 10^{+38}$ | $-1.8 \times 10^{+308}$ |

For example, in single format, the largest representable number is composed as follows:

| | |
|---|---|
| *significand* | $= (2 - 2^{-23})$ |
| | $= 1.11111111111111111111111_2$ |
| exponent | $= 127$ |
| value | $= (2 - 2^{-23}) \times 2^{127}$ |
| | $\approx 3.403 \times 10^{38}$ |

The smallest positive normalized number representable in single format is made up as follows:

| | |
|---|---|
| *significand* | $= 1$ |
| | $= 1.00000000000000000000000_2$ |
| exponent | $= -126$ |
| value | $= 1 \times 2^{-126}$ |
| | $\approx 1.175 \times 10^{-38}$ |

For denormalized numbers, the smallest positive value representable in the single format is made up as follows:

| | |
|---|---|
| *significand* | $= 2^{-23}$ |
| | $= 0.00000000000000000000001_2$ |
| exponent | $= -126$ |
| value | $= 2^{-23} \times 2^{-126}$ |
| | $\approx 1.401 \times 10^{-45}$ |

# Environmental Controls

## Contents

# Environmental Controls

This chapter describes the parts of the floating-point environment that you can control. The IEEE standard specifies that users should be able to control the rounding direction, floating-point exceptions, and in some instances the rounding precision. Libm provides utilities (called **environmental controls**) with which you can set, clear, and test the rounding direction and floating-point exception flags. (See Part 2 for the exact names of functions and instructions that control the floating-point environment.) This chapter describes the four rounding direction modes and the five floating-point exception flags that you can set, clear, and test with Libm. You should read it to learn more about the floating-point environment.

## Rounding Direction Modes

The available **rounding direction modes** are

- to nearest
- upward (toward $+\infty$)
- downward (toward $-\infty$ )
- toward zero

The rounding direction affects all conversions and all arithmetic operations except remainder. All operations are calculated without regard to the range and precision of the data type in which the result is to be stored. That is, conceptually, an operation first produces a result that is infinitely precise, or exact. If the destination data type cannot represent this number exactly, the result is rounded in the direction specified by the rounding mode.

The default rounding direction is to nearest. In this mode, floating-point expressions deliver the value nearest to the exact result that the destination data type can represent. If two representable values are equally close to the exact result, the expression delivers the one whose least significant bit is zero. Hence, halfway cases (for example, 1.5) round to even when the destination is an integer type or when the round-to-integer operation is used. If the magnitude of the exact result is greater than the data type's largest value (by at least one half unit in the last place), then the Infinity with the corresponding sign is delivered.

The other rounding directions are upward, downward, and toward zero. When rounding upward, the result is the representable value (possibly $+\infty$) closest to, and not less than, the exact result. When rounding downward, the result is the representable value (possibly $-\infty$ ) closest to, and not greater than, the exact result. When rounding toward zero, the result is the representable value closest to, and not greater in magnitude than, the exact result. Toward-zero rounding truncates a number to an integer (when the destination is an integer type). Table 3-1 shows some values rounded to integers using different rounding modes.

**Table 3-1** Examples of rounding to integer in different directions

| **Floating-point number** | **Rounded to nearest** | **Rounded toward 0** | **Rounded downward** | **Rounded upward** |
|---|---|---|---|---|
| 1.5 | 2 | 1 | 1 | 2 |
| 2.5 | 2 | 2 | 2 | 3 |
| –2.2 | –2 | –2 | –3 | –2 |

## Exception Flags

Floating-point exceptions are signaled with **exception flags.** When an application begins, all floating-point exception flags are cleared and the default rounding direction (round to nearest) is in effect. This is the **default environment.** When an exception occurs, the appropriate exception flag is set, but the application continues normal operation. Floating-point exception flags merely indicate that a particular event has occurred; they do not change the flow of control for the application. An application can examine or set individual exception flags and can save and retrieve the entire environment (rounding direction and exception flags).

The numerics environment supports five exception flags:

- invalid operation (often called simply *invalid*)
- underflow
- overflow
- divide-by-zero
- inexact

These are discussed in the paragraphs that follow.

## Invalid Operation

The **invalid exception** (or invalid-operation exception) occurs if an operand is invalid for the operation being performed. The result is a quiet NaN for all destination formats (single, double, or double-double). The invalid conditions for the different operations are

| Operation | Invalid condition |
|---|---|
| Addition or subtraction | Magnitude subtraction of Infinities, for example, $(+\infty) + (-\infty)$ |
| Multiplication | $0 \times \infty$ |
| Division | $0/0$ or $\infty/\infty$ |
| Remainder | $x$ rem $y$, where $y$ is 0 or $x$ is infinite |
| Square root | A negative operand |
| Conversion | See Chapter 4, “Conversions” |
| Comparison | With predicates involving less than or greater than, but not unordered, when at least one operand is a NaN |

In addition, any operation on a signaling NaN except the class and sign inquiries produce an invalid exception.

## Underflow

The **underflow exception** occurs when a floating-point result is both tiny and inexact (and therefore is perhaps significantly less accurate than if there were no limit to the exponent range). A result is considered **tiny** if it must be represented as a denormalized number.

## Overflow

The **overflow exception** occurs when the magnitude of a rounded floating-point result is greater than the largest finite number that the floating-point destination data format can represent. (Invalid exceptions, rather than overflow exceptions, flag the production of an out-of-range value for an integer destination type.)

## Divide-by-Zero

The **divide-by-zero exception** occurs when a finite, nonzero number is divided by zero. It also occurs, in the more general case, when an operation on finite operands produces an exact infinite result; for example, $\mathrm{logb}(0)$ returns $-\infty$ and signals divide-by-zero. (Overflow exceptions, rather than divide-by-zero exceptions, flag the production of an inexact infinite result.)

## Inexact

The **inexact exception** occurs if the rounded result of an operation is not identical to the exact (infinitely precise) result. Thus, an inexact exception always occurs when an overflow or underflow occurs. Valid operations on Infinities are always exact and therefore signal no exceptions. Invalid operations on Infinities are described at the beginning of this section.

# Conversions

## Contents

# Conversions

This chapter describes how floating-point numbers are converted to different formats. Most conversions take place implicitly due to assignment statements or evaluations written in a high-level language (such as C). For example, when a floating-point expression is evaluated, one or more of its operands might automatically be converted to a different data format. When a floating-point value is assigned to a variable, another automatic conversion might be necessary. These conversions are handled by code generated by the compiler.

A program can also make explicit Libm calls (such as rint) to perform conversions.

This chapter lists the supported numeric conversions and describes how each of these conversions is performed. You should read it to find out exactly how a floating-point value is converted to a different format. Parts 2 and 3 describe the conversion utilities available to the users of different implementations.

## About Conversions

The IEEE standard requires the following types of conversions:

- from floating-point formats to integer formats
- from integer formats to floating-point formats
- from floating-point values to integer values, with the result in a floating-point format
- between all supported floating-point formats

## Converting Floating-Point to Integer Formats

In IEEE standard arithmetic, the following three types of floating-point to integer conversions are supported either directly by the programming languages or by library implementations:

- round to integer in current rounding direction (the required conversion, discussed in detail in Chapter 3, “Environmental Controls”)
- chop to integer (or round toward zero)
- add half to magnitude and chop

Although the IEEE standard specifies that conversions from floating-point to integer formats be rounded in the current rounding direction, high-level languages usually define their own methods. For example, the default method of converting from floating-point to integer formats in C is simply to discard the fractional part (**truncate**).

Conversions from floating-point to integer formats raise the invalid floating-point exception flag in any of the following cases:

- The floating-point value is out of range for the integer type (for example, an attempt to convert a 64-bit integer value stored in the double data type to a 32-bit integer type).
- The floating-point value is a NaN.
- The floating-point value is an Infinity.

All floating-point to integer conversions that are in range but inexact (that is, the floating-point value was not an integer) raise the inexact floating-point exception flag, although this is not required by the IEEE standard.

Table 4-1 shows some examples of how floating-point values might be converted to a 32-bit integer format by rounding in the current rounding direction. Note that IEEE rounding in the default direction (to nearest) differs from most common rounding functions on halfway cases.

**Table 4-1** Examples of floating-point to integer conversion

| **Floating-point number** | **Rounded to nearest** | **Rounded toward 0** | **Rounded downward** | **Rounded upward** |
|---|---|---|---|---|
| 1.5 | 2 | 1 | 1 | 2 |
| 2.5 | 2 | 2 | 2 | 3 |
| –2.2 | –2 | –2 | –3 | –2 |
| 2,147,483,648.5 | NaN | NaN | NaN | NaN |

## Rounding Floating-Point Numbers to Integers

Programming languages can also round floating-point numbers to integers and leave them stored in the same floating-point data format. These conversions may round in the current rounding direction, or they may explicitly round upward, downward, to the nearest value, or toward zero. These operations do not affect zeros, NaNs, or Infinities, because these three types of special values are already considered integers.

## Converting Integers to Floating-Point Formats

When an integer is converted to a floating-point format whose precision is greater than or equal to the size of the integer format, the conversion is exact. When an integer is converted to a floating-point format whose precision is less than the size of the integer format, the integer is rounded in the current rounding direction. For example, because the single format has 24 bits in the significand, any integer requiring more than 24 bits of precision will not be converted to its exact value.

## Converting Between Floating-Point Formats

Programming languages support conversions between both of the IEEE floating-point data formats supported by Mac OS X. This section describes these conversions.

### Converting Between Single and Double Formats

Libm directly supports the single and double formats and conversions between them. When a single format number is converted to a double format number, the conversion is exact.

When a double format number is converted to a single format number, it is rounded to the closest single value in the current rounding direction. The conversion might raise the exceptions shown in Table 4-2**.**

**Table 4-2** Double to single conversion: Possible exceptions

| **Exception** | **Raised when** |
|---|---|
| Inexact | Significand requires > 24 bits of precision |
| Overflow | Exponent > 127 |
| Underflow | Exponent < –126 |

# Numeric Operations and Functions

## Contents

# Numeric Operations and Functions

This chapter describes the operations (comparisons, arithmetic operations, and auxiliary and transcendental functions) that programming languages and Libm allow you to perform on floating-point numbers. Numeric operations are evaluated as floating-point expressions; as such they are affected by, and might affect, the floating-point environment.

Read this chapter to find out what numeric operations are supported and how they work. For a description of the floating-point environment, see Chapter 3, "Environmental Controls."

## Comparisons

Programming languages for Mac OS X support the usual numeric comparisons: less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to (see Table 5-1 for a complete listing). For real numbers, these comparisons behave according to the familiar ordering of real numbers.

### Comparisons With NaNs and Infinities

Numeric comparisons handle NaNs and Infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any numeric values *a* and *b*, exactly one of the following statements is true:

- $a < b$
- $a > b$
- $a = b$
- *a* and *b* are unordered

The following rule determines which statement is true: If *a* or *b* is a NaN, then *a* and *b* are unordered; otherwise, *a* is less than, equal to, or greater than *b* according to the ordering of the real numbers, with the understanding that

$+0 = -0 \quad \text{and} \quad -\infty < \text{every real number} < +\infty$

### Comparison Operators

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the C expression $x <= y$ is true if *x* is less than *y* or if *x* equals *y*, and is false if *x* is greater than *y* or if *x* and *y* are unordered. Note that the numeric not-equal relation means less than, greater than, or unordered. The

IEEE Standard 9899 extends the usual set of C relational operators to a set of 14 comparisons, shown in Table 5-1.

**Table 5-1** Comparison symbols

| **Symbol** | **Relation** | **Invalid if unordered?** |
|---|---|---|
| < | Less than | Yes |
| > | Greater than | Yes |
| <= | Less than or equal to | Yes |
| >= | Greater than or equal to | Yes |
| == | Equal to | No |
| != | Not equal to (unordered, less than, or greater than) | No |
| !<>= | Unordered | No |
| <> | Less than or greater than | Yes |
| <>= | Not unordered (less than, equal to, or greater than) | Yes |
| !<= | Not less than or equal to (unordered or greater than) | No |
| !< | Not less than (unordered, greater than, or equal to) | No |
| !>= | Not greater than or equal to (unordered or less than) | No |
| !> | Not greater than (unordered, less than, or equal to) | No |
| !<> | Unordered or equal | No |

Some relational operators in high-level language comparisons contain the predicate less than or greater than, but not unordered. In C, those relational operators are <, <=, >, and >= (but not == and !=). For those relations, comparisons signal invalid if the operands are unordered, that is, if either operand is a NaN. For the operators equal and nonequal, comparisons with NaN are not misleading; thus, when $x$ or $y$ is a NaN, the relation $x == y$ is false, which is not misleading. Likewise, when $x$ or $y$ is a NaN, $x != y$ returns true, again not misleading. On the other hand, when $x$ or $y$ is a NaN, $x < y$ being false might tempt you to conclude that $x \geq y$, so the floating-point routines signal invalid to help you avoid the pitfall. Table 5-1 shows the results of such comparisons in C.

The full 26 distinct comparison predicates of the IEEE standard may be obtained by logical negation of all of the operators except for == and !=, which never signal invalid. For example, ($x < y$) and !($x$ !< $y$) are logically equivalent for all possible values of $a$ and $b$, but the former raises the invalid exception flag when $x$ and $y$ compare unordered while the latter does not.

In addition to the comparison operators, there are also library functions that perform comparisons. See “Comparison Functions” on page 9-3.

## Arithmetic Operations

Mac OS X programming languages and Libm provide the seven arithmetic operations required by the IEEE standard for its two data types, as shown for the C language in Table 5-2 and described in the sections that follow.

**Table 5-2** Arithmetic operations in C

| Operation | C symbol |
|---|---|
| Add | `+` |
| Subtract | `–` |
| Multiply | `*` |
| Divide | `/` |
| Square root | `sqrt` |
| Remainder | `remainder` |
| Round-to-integer | `rint` |

The language processors for Mac OS X automatically use their chosen expression evaluation methods for the normal inline operators (`+`, –, `*`, `/`). All the arithmetic operations produce the best possible result: the mathematically exact result, coerced to the precision and range of the evaluation format. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE standard (see Chapter 3, “Environmental Controls”).

Some of the arithmetic operations are implemented in software. These operations are declared to be type `double_t`, which is defined to be type `double`.

### +

You can use the + symbol to add two real numbers.

```
x + y
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The + operator performs the standard addition of two floating-point numbers.

**EXCEPTIONS**

When $x$ and $y$ are both finite and nonzero, either the result of $x + y$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 5-3 shows the results when one of the operands of the addition operation is a zero, a NaN, or an Infinity. In this table, $x$ is any floating-point number.

**Table 5-3** Special cases for floating-point addition

| Operation | Result | Exceptions raised |
|---|---|---|
| $x + (+0)$ | $x$ | None |
| $x + (-0)$ | $x$ | None |
| $(-0) + (+0)$ | +0 | None |
| $(-0) + (-0)$ | –0 | None |
| $x$ + NaN | NaN | None* |
| $x + (+\infty)$ | $+\infty$ | None |
| $x + (-\infty)$ | $-\infty$ | None |
| $+\infty + (-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

## –

You can use the – symbol to subtract one real number from another.

```
x – y
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The – operator performs the standard subtraction of two floating-point numbers.

### EXCEPTIONS

When $x$ and $y$ are both finite and nonzero, either the result of $x - y$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

### SPECIAL CASES

Table 5-4 shows the results when one of the operands of the subtraction operation is a zero, a NaN, or an Infinity. In this table, $x$ is any floating-point number.

**Table 5-4** Special cases for floating-point subtraction

| Operation | Result | Exceptions raised |
|---|---|---|
| $x - (+0)$ | $x$ | None |
| $(+0) - x$ | $-x$ | None |
| $(+0) - (-0)$ | $+0$ | None |
| $x - (-0)$ | $x$ | None |
| $(-0) - x$ | $-x$ | None |
| $(-0) - (+0)$ | $-0$ | None |
| $(-0) - (-0)$ | $+0$ | None |
| $x$ – NaN | NaN | None* |
| NaN – $x$ | NaN | None* |
| $x - (+\infty)$ | $-\infty$ | None |
| $(+\infty) - x$ | $+\infty$ | None |
| $(+\infty) - (+\infty)$ | NaN | Invalid |
| $x - (-\infty)$ | $+\infty$ | None |
| $(-\infty) - x$ | $-\infty$ | None |
| $(-\infty) - (-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

## *

You can use the `*` symbol to multiply two real numbers.

```
x * y
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The `*` operator performs the standard multiplication of two floating-point numbers $(x \times y)$ .

**EXCEPTIONS**

When $x$ and $y$ are both finite and nonzero, either the result of $x * y$ is exact or it raises one of the following exceptions:

- inexact (if the result of $x * y$ must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 5-5 shows the results when one of the operands of the multiplication operation is a zero, a NaN, or an Infinity. In this table, $x$ is a nonzero floating-point number.

**Table 5-5** Special cases for floating-point multiplication

| Operation | Result | Exceptions raised |
|---|---|---|
| $x$ * +0 | ±0 | None |
| $x$ * –0 | ±0 | None |
| ±∞ * ±0 | NaN | Invalid |
| $x$ * NaN | NaN | None* |
| $x$ * +∞ | ±∞ | None |
| $x$ * –∞ | ±∞ | None |
| ±0 * ±∞ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

## /

You can use the `/` symbol to divide one real number by another.

```
x / y
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The `/` operator performs the standard division of two floating-point numbers.

**EXCEPTIONS**

When $x$ and $y$ are both finite and nonzero, either the result of $x/y$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 5-6 shows the results when one of the operands of the division operation is a zero, a NaN, or an Infinity. In this table, $x$ is any floating-point number.

**Table 5-6** Special cases for floating-point division

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $(+0)/x$ | ±0 | None |
| $x/(+0)$ | ±∞ | Divide-by-zero |
| $(–0)/x$ | ±0 | None |
| $x/(–0)$ | ±∞ | Divide-by-zero |
| ±0/±0 | NaN | Invalid |
| $x$/NaN | NaN | None* |
| NaN/$x$ | NaN | None* |
| $x/(+\infty)$ | ±0 | None |

*continued*

**Table 5-6** Special cases for floating-point division (continued)

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $(+\infty)/x$ | ±∞ | None |
| $x\ /\ (-\infty)$ | ±0 | None |
| $(-\infty)/x$ | ±∞ | None |
| $(\pm\infty)/(\pm\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

## sqrt

You can use the square root (`sqrt`) function to compute the square root of a real number.

```
double_t sqrt(double_t x);
```

| | |
|---|---|
| `x` | Any positive floating-point number. |

### DESCRIPTION

$$\text{sqrt}(x) = \sqrt{x}$$

### EXCEPTIONS

When $x$ is finite and nonzero, either the result of $\text{sqrt}(x)$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded)
- invalid (if $x$ is negative)

### SPECIAL CASES

Table 5-7 shows the results when the argument to the square root function is a zero, a NaN, or an Infinity, plus other special cases for the square root function. In this table, $x$ is a finite, nonzero floating-point number.

**Table 5-7** Special cases for floating-point square root

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $\text{sqrt}(x)$ for $x < 0$ | NaN | Invalid |
| $\text{sqrt}(+0)$ | +0 | None |
| $\text{sqrt}(-0)$ | –0 | None |
| $\text{sqrt}(\text{NaN})$ | NaN | None* |
| $\text{sqrt}(+\infty)$ | $+\infty$ | None |
| $\text{sqrt}(-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

## remainder, remquo, and fmod

You can use the `remainder`, `remquo`, and `fmod` functions to perform the remainder operation recommended in the IEEE standard.

```
double_t remainder (double_t x, double_t y);
double_t remquo (double_t x, double_t y, int *quo);
double_t fmod (double_t x, double_t y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |
| `quo` | On return, the signed lowest seven bits (in the range of –127 to +127, inclusive) of the integer value closest to the quotient $x/y$. This partial quotient might be of use in certain argument reduction algorithms. |

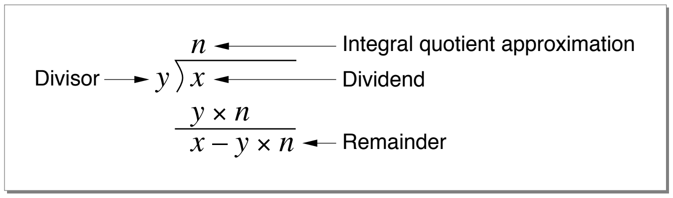
**DESCRIPTION**

The IEEE remainder (rem) operation returns the result of the following computation.

$r = x \text{ rem } y = x - y \times n$

where $n$ is the integer nearest the exact value of the quotient $x/y$. This expression can be found even in the conventional integer-division algorithm, shown in Figure 5-1.

**Figure 5-1** Integer-division algorithm



Whenever $|n - x/y| = 1/2$, $n$ is even.

If the value of $r$ is 0, the sign of $r$ is that of $x$.

The rem operation is always exact.

The IEEE rem operation differs from other commonly used remainder and modulo operations. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. Other remainder functions can be constructed from the IEEE remainder function by appropriately adding or subtracting $y$.

**EXCEPTIONS**

When $x$ and $y$ are finite, nonzero floating-point numbers in single or double format, the result of $x$ rem $y$ is exact.

**SPECIAL CASES**

Table 5-8 shows the results when one of the arguments to the rem operation is a zero, a NaN, or an Infinity. In this table, $x$ is a finite, nonzero floating-point number.

**Table 5-8** Special cases for floating-point remainder

| Operation | Resul t | Exceptions raised |
|---|---|---|
| +0 rem $x$ | +0 | None |
| $x$ rem (+0) | NaN | Invalid |
| –0 rem $x$ | –0 | None |
| $x$ rem (–0) | NaN | Invalid |
| $x$ rem NaN | NaN | None* |
| NaN rem $x$ | NaN | None* |
| $x$ rem +∞ | $x$ | None |
| +∞ rem $x$ | NaN | Invalid |
| $x$ rem –∞ | $x$ | None |
| –∞ rem $x$ | NaN | Invalid |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = remainder(5, 3);    /* z = –1. */
/* 5 rem 3 = 5 – 3 × 2 = –1 because 1 < 5/3 < 2 and because
   5/3 = 1.66666... is closer to 2 than to 1, quo is taken to
   be 2. */

z = remainder(43.75, 2.5); /* z = –1.25. */
/* 43.75 rem 2.5 = 43.75 – 2.5 × 18 = –1.25 because
   17 < 43.75/2.5 < 18 and because 43.75/2.5 = 17.5 is
   equally close to both 17 and 18, quo is taken to be the
   even quotient, 18. */

z = remainder(43.75, +INFINITY); /* z = 43.75 */
/* 43.75 rem ∞ = 43.75 – 0 × ∞ = 43.75 because 43.75 / ∞ = 0,
   quo is taken to be 0. */
```

## rint

You can use the round-to-integer operation (`rint` function) to round a number to the nearest integer in the current rounding direction.

```
double_t rint(double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `rint` function rounds its argument to an integer in the current rounding direction. The available rounding directions are upward, downward, to nearest (default), and toward zero. With the default rounding direction, if the argument is equally near two integers, the even integer is used as the result.

In each floating-point data type, all values of sufficiently great magnitude are integers. For example, in single format, all numbers whose magnitudes are at least $2^{23}$ are integers. This means that $+\infty$ and $-\infty$ are already integers and return exact results.

The `rint` function performs the round-to-integer arithmetic operation described in the IEEE standard. For other C functions that perform rounding to integer, see Chapter 8, "Conversion Functions."

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of rint($x$) is exact or it raises the following exception

- inexact (if $x$ is not an integer)

**SPECIAL CASES**

Table 5-9 shows the results when the argument to the round-to-integer operation is a zero, a NaN, or an Infinity.

**Table 5-9** Special cases for floating-point round-to-integer

| Operation | Result | Exceptions raised |
|---|---|---|
| rint(+0) | +0 | None |
| rint(–0) | –0 | None |
| rint(NaN) | NaN | None[*] |
| rint(+∞) | +∞ | None |
| rint(–∞) | –∞ | None |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

Table 5-10 shows some example results of `rint`, given different rounding directions.

**Table 5-10** Examples of `rint`

| | Current rounding direction | | | |
|---|---|---|---|---|
| **Example** | To nearest | Toward 0 | Downward | Upward |
| `rint(1.5)` | 2 | 1 | 1 | 2 |
| `rint(2.5)` | 2 | 2 | 2 | 3 |
| `rint(–2.2)` | –2 | –2 | –3 | –2 |

## Auxiliary Functions

The IEEE standard defines a number of recommended functions (called *auxiliary functions*) that are generally useful in numerical programming. The recommended functions supported by Libm are

- copysign$(x, y)$ : copy the sign of $y$ onto the magnitude of $x$
- fabs$(x)$ : absolute value
- logb$(x)$ : binary exponent
- nan functions: NaN generators
- nextafter functions
- scalb$(x)$ : binary scaling

The auxiliary functions are provided in the C library MathLib. For more information about these functions, see Part 2.

## Transcendental Functions

Libm provides several basic mathematical functions in addition to the auxiliary functions recommended in the IEEE standard. These functions include

- logarithms
- exponentials
- trigonometric functions
- error and gamma functions

For information about the transcendental functions supported, see Part 2.

# The Mac OS X Numerics C Implementation

This part describes the Macintosh OS X implementation for the C programming language. The numeric implementation for the C language conforms to both IEEE Standard 754, referred to in this book as *the IEEE standard*, and the IEEE Standard 9899. As stated in Part 1, the IEEE Standard 9899 describes a standard way of doing floating-point arithmetic for the C programming language. The IEEE Standard 754 specifies a standard for floating-point arithmetic for all computers regardless of the architecture or of any high-level language. The IEEE Standard 9899 conforms to the IEEE standard and standardizes its implementation for the C programming language, so that if you write a program that uses IEEE Standard 9899 features, it will compile with any 9899-compliant compiler.

Macintosh OS X numerics in C is supported largely through a library called Libm. This library contains macros, functions, and type definitions that provide conformance to the IEEE Standard 754 and the IEEE Standard 9899.

This part describes the Libm library, its adherence to each piece of the IEEE numerics environment, and its additional features that conform to the IEEE Standard 9899. For more information about the semantics of Macintosh OS X numerics, see Part 1. Read Part 2 if you are a programmer and you want to find out how to access the features described in Part 1 using the C language.

# Numeric Data Types in C

## Contents

# Numeric Data Types in C

This chapter describes the numeric data types available in C and shows how to determine the class and sign of values represented in numeric data types. As stated in Chapter 2, "Floating-Point Data Formats," the Mac OS X numerics environment provides two numeric data formats: single (32 bits long) and double (64 bits long). Each can represent normalized numbers, denormalized numbers, zeros, NaNs, and Infinities. See Chapter 2 for information about the numeric data formats and about how they represent values. Read this chapter to find out about the mapping of numeric formats to floating-point types in C, about the floating-point type declarations made in the Mac OS X numerics library (Libm), and about the library utilities available that can determine the class of a floating-point value.

## C Data Types

Table 6-1 shows how the PowerPC Numerics data formats map to the C floating-point variable types. This mapping follows the recommendations in the FPCE technical report.

**Table 6-1** Names of data types

| PowerPC Numerics format | C type |
|---|---|
| IEEE single | `float` |
| IEEE double | `double` |

## Efficient Type Declarations

Libm contains two floating-point type definitions, `float_t` and `double_t` in the header `Types.h`. If you define a variable to be `float_t` or `double_t`, it means "use the most efficient floating-point format for this architecture." Table 6-2 shows the definitions for `float_t` and `double_t` for the PowerPC architecture.

**Table 6-2** `float_t` and `double_t` types

| Architecture | `float_t` type | `double_t` type |
|---|---|---|
| PowerPC | `float` | `double` |

For the PowerPC microprocessor, the most natural format for computations is double, but the architecture allows computations in single precision as well. Therefore, for the PowerPC microprocessor, `float_t` is defined to be `float` (single precision) and `double_t` is defined to be `double`.

## Inquiries: Class and Sign

Libm provides macros you can use to determine the class and sign of a floating-point value. All of these macros return type `long int`. They are listed in Table 6-3.

**Table 6-3** Class and sign inquiry macros

| Macro | Value returned | Condition |
|---|---|---|
| `fpclassify(x)` | `FP_NAN` | `x` is a NaN |
| | `FP_INFINITE` | `x` is –∞ or +∞ |
| | `FP_ZERO` | `x` is +0 or –0 |
| | `FP_NORMAL` | `x` is a normalized number |
| | `FP_SUBNORMAL` | `x` is a denormalized (subnormal) number |
| `isnormal(x)` | 1 | `x` is a normalized number |
| | 0 | `x` is not a normalized number |
| `isfinite(x)` | 1 | `x` is not –∞ , +∞, or NaN |
| | 0 | `x` is –∞ , +∞, or NaN |
| `isnan(x)` | 1 | `x` is a NaN (quiet or signaling) |
| | 0 | `x` is not a NaN (quiet or signaling) |
| `signbit(x)` | 1 | The sign bit of `x` is 1 (`x` is negative) |
| | 0 | The sign bit of `x` is 0 (`x` is positive) |

## Creating Infinities and NaNs

Lib defines the constants `INFINITY` and `NAN`, so that you can assign these values to variables in your program, and provides the following function that returns NaNs:

```
double nan  (const char *tagp);
```

The `nan` function returns a quiet NaN with a fraction field that is equal to the argument `tagp`. The argument `tagp` is a pointer to a string that will be copied into bits 8 through 15 of the NaN's fraction field. The string should specify a decimal number between 0 and 255. For example:

```
nan("32")
```

creates a NaN with code 32. If you supply a negative string, or if `tagp` is empty, it is the same as supplying the string "0". If you supply a string greater than 255, it is the same as supplying the string "255". For a list of predefined NaN codes, see Chapter 2, "Floating-Point Data Formats."

## Numeric Data Types Summary

This section summarizes the C constants, macros, functions, and type definitions associated with creating floating-point values or determining the class and sign of a floating-point value.

### C Summary

#### Constants

```
#define           HUGE_VAL            1e500
#define           HUGE_VALF           1e50f
#define           INFINITY            HUGE_VALF

#if defined(__APPLE_CC__) && (__APPLE_CC__ >= 1345)
#define NAN__builtin_nanf("0x7fc00000") /* Constant expression, can be used
as initializer. */
#else
#define NAN       __nan( )
#endif
```

### Class and Sign Inquiry Macros

```
#define  fpclassify( x )   ( (   sizeof ( x ) == sizeof(double) ) ? \
                                 __fpclassifyd  ( x )             : \
                           (     sizeof ( x ) == sizeof( float) ) ? \
                                 __fpclassifyf ( x )              : \
                                 __fpclassify  ( x ) )

#define  isnormal( x )     ( (   sizeof ( x ) == sizeof(double) ) ? \
                                 __isnormald ( x )                : \
                           (     sizeof ( x ) == sizeof( float) ) ? \
                                 __isnormalf ( x )                : \
                                 __isnormal  ( x ) )

#define  isfinite( x )      ( (  sizeof ( x ) == sizeof(double) ) ? \
                                 __isfinited ( x )                : \
                           (     sizeof ( x ) == sizeof( float) ) ? \
                                 __isfinitef ( x )                : \
                                 __isfinite  ( x ) )

#define  isnan( x )        ( (   sizeof ( x ) == sizeof(double) ) ? \
                                 __isnand ( x )                   : \
                           (     sizeof ( x ) == sizeof( float) ) ? \
                                 __isnanf ( x )                   : \
                                 __isnan  ( x ) )

#define  signbit( x )      ( (   sizeof ( x ) == sizeof(double) ) ? \
                                 __signbitd ( x )                 : \
                           (     sizeof ( x ) == sizeof( float) ) ? \
                                 __signbitf ( x )                 : \
                                 __signbitl ( x ) )
```

## Data Types

```
enum {

         FP_NAN          = 1,     /* NaN */
         FP_INFINITE     = 2,     /* + or - infinity */
         FP_ZERO         = 3,     /* + or - zero */
         FP_NORMAL       = 4,     /* all normal numbers */
         FP_SUBNORMAL    = 5      /* denormal numbers */
};
```

```
typedef float  float_t;
typedef double double_t;
```

## Special Value Routines

### Creating NaNs

```
double nan                  (const char *tagp);
```

# Environmental Control Functions

## Contents

# Environmental Control Functions

This chapter describes how to control the floating-point environment using functions defined in Libm.

As described in Chapter 3, "Environmental Controls," the rounding direction and the exception flags are the parts of the environment that you can access. You can test and change the rounding direction, and you can test, set, and clear the exceptions flags. You may also save and restore both the rounding direction and exception flags together as a single entity. This chapter describes the functions that perform these tasks. For the definitions of rounding direction and exception flags, see Chapter 3.

Read this chapter to learn how to access and manipulate the floating-point environment in the C language. All of the environmental control function declarations appear in the file `fenv.h`.

## Controlling the Rounding Direction

In Libm, the following functions control the rounding direction:

| | |
|---|---|
| `fegetround` | Returns the current rounding direction. |
| `fesetround` | Sets the rounding direction. |

The four rounding direction modes are defined as the constants shown in Table 7-1.

**Table 7-1** Rounding direction modes in Libm

| **Rounding direction** | **Constant** |
|---|---|
| To nearest | `FE_TONEAREST` |
| Toward zero | `FE_TOWARDZERO` |
| Upward | `FE_UPWARD` |
| Downward | `FE_DOWNWARD` |

### fegetround

You can use the `fegetround` function to save the current rounding direction.

```
int fegetround (void);
```

**DESCRIPTION**

The `fegetround` function returns an integer that specifies which rounding direction is currently being used. The integer it returns will be equal to one of the constants shown in Table 7-1. You can save the returned value in an integer variable to save the current rounding direction.

**EXAMPLES**

```
int rounddir;
double_t x, y, result;

rounddir = fegetround();      /* save rounding direction */

result = x + y;
if (rounddir == FE_TONEAREST)
   printf("The result was rounded to the nearest value.\n");
else if (rounddir == FE_UPWARD)
   printf("The result was rounded upward.\n");
else if (rounddir == FE_DOWNWARD)
   printf("The result was rounded downward.\n");
else if (rounddir == FE_TOWARDZERO)
   printf("The result was rounded toward zero.\n");
```

## fesetround

You can use the `fesetround` function to change the rounding direction.

```
int fesetround (int round);
```

| | |
|---|---|
| `round` | One of the four rounding direction constants (see Table 7-1). |

**DESCRIPTION**

The `fesetround` function sets the rounding direction to the mode specified by its argument. If the value of `round` does not match any of the rounding direction constants, the function returns 1 and does not change the rounding direction. (Otherwise it returns 0.)

By convention, if you change the rounding direction inside a function, first save the rounding direction of the calling function using `fegetround` and restore the saved direction at the end of the function. This way, the function does not affect the rounding direction of its caller. If the function is to be reentrant, then storage for the caller's rounding direction must be local.

One reason to change the rounding direction would be to put bounds on errors (at least for the basic arithmetic operations and square root). Suppose you want to evaluate an expression such as

$$x = (a \times b + c \times d)/(f + g)$$

where *a, b, c, d, f,* and *g* are positive.

To make sure that the result is always larger than the exact value, you can change the expression such that all roundings cause errors in the same direction. The example that follows changes the rounding direction to compute an upper bound for the expression, and then restores the previous rounding.

**EXAMPLES**

```
double_t big_divide(void)
{
   double_t x_up, a, b, c, d, f, g;
   int r;              /* specifies rounding direction */

   r = fegetround();    /* save caller’s rounding direction */
   fesetround(FE_DOWNWARD);
                     /* downward rounding for denominator */
   x_up = f + g;
   fesetround(FE_UPWARD);
                     /* upward rounding for expression */
   x_up = (a * b + c * d) / x_up;
   fesetround(r);
                     /* restore caller’s rounding direction */
   return(x_up);
}
```

## Controlling the Exception Flags

In Libm, the following functions control the floating-point exception flags:

| | |
|---|---|
| `feclearexcept` | Clears one or more exceptions. |
| `fegetexceptflag` | Saves one or more exception flags. |
| `feraiseexcept` | Raises one or more exceptions. |
| `fesetexceptflag` | Restores the state of one or more exception flags. |
| `fetestexcept` | Returns the value of one or more exception flags. |

The five floating-point exception flags are defined as the constants shown in Table 7-2.

**Table 7-2** Floating-point exception flags in Libm

| Exception | Constant |
|---|---|
| Inexact | `FE_INEXACT` |
| Divide-by-zero | `FE_DIVBYZERO` |
| Underflow | `FE_UNDERFLOW` |
| Overflow | `FE_OVERFLOW` |
| Invalid | `FE_INVALID` |

Libm also defines another constant, `FE_ALL_EXCEPT`, which is the logical OR of all five exceptions. Using `FE_ALL_EXCEPT`, you can manipulate all five floating-point exception flags as a single entity. The type `fexcept_t` also exists so that all the exception flags may be accessed at once.

## feclearexcept

You can use the `feclearexcept` function to clear one or more floating-point exceptions.

```
void feclearexcept (int excepts);
```

| | |
|---|---|
| `excepts` | A mask indicating which floating-point exception flags should be cleared. |

**DESCRIPTION**

The `feclearexcept` function clears the floating-point exceptions specified by its argument. The argument may be one of the constants in Table 7-2, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

**EXAMPLES**

```
feclearexcept(FE_INEXACT);          /* clears the inexact flag */
feclearexcept(FE_INEXACT|FE_UNDERFLOW);
                     /* clears the inexact and underflow flags */
feclearexcept(FE_ALL_EXCEPT);             /* clears all flags */
```

## fegetexceptflag

You can use the `fegetexcept` function to save the current value of one or more floating-point exception flags.

```
void fegetexceptflag (fexcept_t *flagp, int excepts);
```

| | |
|---|---|
| `flagp` | A pointer to where the exception flag values are to be stored. |
| `excepts` | A mask indicating which exception flags to save. |

**DESCRIPTION**

The `fegetexceptflag` function saves the values of the floating-point exception flags specified by the argument `excepts` to the area pointed to by the argument `flagp`. The `excepts` argument may be one of the constants in Table 7-2 on page 7-6, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

**EXAMPLES**

```
fegetexceptflag(flagp, FE_INVALID); /* saves the invalid flag */
fegetexceptflag(flagp, FE_INVALID|FE_OVERFLOW|FE_DIVBYZERO);
      /* saves the invalid, overflow, and divide-by-zero flags */
fegetexceptflag(flagp, FE_ALL_EXCEPT);    /* saves all flags */
```

## feraiseexcept

You can use the `feraiseexcept` function to raise one or more floating-point exceptions.

```
void feraiseexcept (int excepts);
```

| | |
|---|---|
| `excepts` | A mask indicating which floating-point exception flags should be set. |

**DESCRIPTION**

The `feraiseexcept` function sets the floating-point exception flags specified by its argument. The argument may be one of the constants in Table 7-2 on page 7-6, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

**EXAMPLES**

```
feraiseexcept(FE_OVERFLOW);         /* sets the overflow flag */
feraiseexcept(FE_INEXACT|FE_UNDERFLOW);
                        /* sets the inexact and underflow flags */
feraiseexcept(FE_ALL_EXCEPT);                /* sets all flags */
```

## fesetexcept

You can use the `fesetexcept` function to restore the values of the floating-point exception flags previously saved by a call to `fegetexcept`.

```
void fesetexcept (const fexcept_t *flagp, int excepts);
```

| | |
|---|---|
| `flagp` | A pointer to the values the floating-point exception flags should have. |
| `excepts` | A mask indicating which exception flags should have their values changed. |

**DESCRIPTION**

The `fesetexceptflag` function sets the floating-point exception flags indicated by the argument `excepts` to the values indicated by the argument `flagp`. The `excepts` argument may be one of the constants in Table 7-2 on page 7-6, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

You must call `fegetexcept` before this function to set the `flagp` argument. This argument cannot be set in any other way.

**EXAMPLES**

```
fesetexceptflag(flagp, FE_INVALID);/* restores the invalid flag */
fesetexceptflag(flagp, FE_INVALID|FE_OVERFLOW|FE_DIVBYZERO);
   /* restores the invalid, overflow, and divide-by-zero flags */
fesetexceptflag(flagp, FE_ALL_EXCEPT);    /* restores all flags */
```

## fetestexcept

You can use the `fetestexcept` function to find out if one or more floating-point exceptions has occurred.

```
int fetestexcept (int excepts);
```

| | |
|---|---|
| `excepts` | A mask indicating which floating-point exception flags should be tested. |

**DESCRIPTION**

The `fetestexcept` function tests the floating-point exception flags specified by its argument. The argument may be one of the constants in Table 7-2 on page 7-6, two or more of these constants ORed together, or the constant `FE_ALL_EXCEPT`.

If all exception flags being tested are clear, `fetestexcept` returns a 0. If one of the flags being tested is set, `fetestexcept` returns the constant associated with that flag. If more than one flag is set, `fetestexcept` returns the result of ORing their constants together. For example, if the inexact exception is set, `fetestexcept` returns `FE_INEXACT`. If both the inexact and overflow exceptions flags are set, `fetestexcept` returns `FE_INEXACT | FE_OVERFLOW`.

**EXAMPLES**

```
feraiseexcept(FE_DIVBYZERO|FE_OVERFLOW);
feclearexcept(FE_INEXACT|FE_UNDERFLOW|FE_INVALID);

/* Now the divide-by-zero and overflow flags are 1, and the
   rest of the flags are 0. */

i = fetestexcept(FE_INEXACT);
                              /* i = 0 because inexact is clear */
i = fetestexcept(FE_DIVBYZERO);
                              /* i = FE_DIVBYZERO */
i = fetestexcept(FE_UNDERFLOW);
                              /* i = 0 */
i = fetestexcept(FE_OVERFLOW);
                              /* i = FE_OVERFLOW */
i = fetestexcept(FE_ALL_EXCEPT);
                              /* i = FE_DIVBYZERO | FE_OVERFLOW */
i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
                              /* i = FE_DIVBYZERO */
```

## Accessing the Floating-Point Environment

Libm defines four functions that access the entire floating-point environment:

| | |
|---|---|
| `fegetenv` | Returns the current environment. |
| `feholdexcept` | Saves the previous environment and clears all exception flags. |
| `fesetenv` | Sets new environmental values. |
| `feupdateenv` | Restores a previously saved environment. |

These functions take parameters of type `fenv_t`. Type `fenv_t` is the environment word type. In general, the environmental access functions either take a pointer to a variable of type `fenv_t` or accept the macro `FE_DFL_ENV`, which defines the default environment (default rounding direction and all exceptions cleared).

## fegetenv

You can use the `fegetenv` function to save the current state of the floating-point environment.

```
void fegetenv (fenv_t *envp);
```

| | |
|---|---|
| `envp` | A pointer to an environment word that will store the current state of the environment upon the function's return. |

**DESCRIPTION**

The `fegetenv` function saves the current state of the rounding direction modes and the floating-point exception flags in the object pointed to by its `envp` argument.

**EXAMPLES**

```
double_t func (double_t x, double_t y)
{
   fenv_t *env;

   x = x + y;        /* floating-point op; may raise exceptions */
   fegetenv(env);    /* save state of env after add */

   y = y * x;        /* floating-point op; may raise exceptions */
   .
   .
   .
}
```

## feholdexcept

You can use the `feholdexcept` function to save the current floating-point environment and then clear all exception flags.

```
int feholdexcept (fenv_t *envp);
```

| | |
|---|---|
| `envp` | A pointer to an environment word where the environment should be saved. |

**DESCRIPTION**

The `feholdexcept` function stores the current environment in the argument `envp` and clears the floating-point exception flags. Note that this function does not affect the rounding direction. It is the same as performing the following two calls:

```
fegetenv(envp);
feclearexcept(FE_ALL_EXCEPT);
```

Call `feholdexcept` at the beginning of a function so that the function can start with all exceptions cleared but not change the caller's environment. Use `feupdateenv` to restore the caller's environment at the end of the function. The `feupdateenv` function keeps any exceptions raised by the current function set while restoring the rest of the caller's environment. Thus, using `feholdexcept` and `feupdateenv` together preserves all raised floating-point exceptions while allowing new ones to be raised as well.

**EXAMPLES**

```
void subroutine(void)
{
   fenv_t *e;        /* local storage for environment */

   feholdexcept(e);  /* save caller's environment and
                        clear exceptions */

   /* subroutine's operations here */

  feupdateenv(e);    /* restore caller's environment */
}
```

## fesetenv

You can use the `fesetenv` function to restore the floating-point environment.

```
void fesetenv (const fenv_t *envp);
```

| | |
|---|---|
| `envp` | A pointer to a word containing the value to which the environment should be set. |

**DESCRIPTION**

The `fesetenv` function sets the floating-point environment to the value pointed to by its argument `envp`. The value of `envp` must come from a call to either `fegetenv` or `feholdexcept`, or it may be the constant `FE_DFL_ENV`, which specifies the default environment. In the default environment, all exception flags are clear and the rounding direction is set to the default.

**EXAMPLES**

```
double_t func (double_t x, double_t y)
{
   fenv_t *env;

   fesetenv(FE_DFL_ENV);      /* clear environment */

   x = x + y;         /* floating-point op; may raise exceptions */
   fegetenv(env);     /* save state of env after add */

   y = y * x;         /* floating-point op; may raise exceptions */
   fesetenv(env);     /* ignore environmental changes by
                         multiplication operator */
   .
   .
   .
}
```

## feupdateenv

You can use the `feupdateenv` function to restore the floating-point environment previously saved with `feholdexcept`.

```
void feupdateenv (const fenv_t *envp);
```

| | |
|---|---|
| `envp` | A pointer to the word containing the environment to be restored. |

**DESCRIPTION**

The `feupdateenv` function, which takes a saved environment as argument, does the following:

1. It temporarily saves the exception flags (raised by the current function).
2. It restores the environment received as an argument.
3. It signals the temporarily saved exceptions.

The feupdateenv function facilitates writing subroutines that appear to their callers to be **atomic operations** (such as addition, square root, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which might be irrelevant or misleading. Thus, exceptions signaled between the feholdexcept and feupdateenv functions are hidden from the calling function unless the exceptions remain raised when the feupdateenv procedure is called.

**EXAMPLES**

```
/* NumFcn signals underflow if its result is denormalized,
overflow if its result is INFINITY, and inexact always, but hides
spurious exceptions occurring from internal computations. */

long double NumFcn(void)
{
   fenv_t e;                     /* local environment storage */
   enum NumKind c;               /* for class inquiry */
   fexcept_t * flagp;
   long double result;

   feholdexcept(&e);             /* save caller’s environment and
                                    clear exceptions */

      /* internal computation */

   c = fpclassify(result);       /* class inquiry */

   feclearexcept(FE_ALL_EXCEPT); /* clear all exceptions */
   feraiseexcept(FE_INEXACT);    /* signal inexact */

   if (c == FP_INFINITE)
      feraiseexcept(FE_OVERFLOW);
   else if (c == FP_SUBNORMAL)
      feraiseexcept(FE_UNDERFLOW);

   feupdateenv(&e);
   /* restore caller’s environment, and then signal
      exceptions raised by NumFcn */

   return(result);
}
```

## Trapping Floating-point Exceptions

When a floating-point exception occurs, the Macinstosh OS floating-point library returns a result value (usually a NaN, infinity, underflow, or overflow) and sets the appropriate flaoting-point exception flags.

For many programs the result value delivered is appropriate, and enables the application to continue, complete without error, and return a reasonable result.

In other programs, where more direct control is required, the program may test the exception flags explicity after a critical operation, using the floating point environment inquiry functions described in this chapter (such as `fegetenv`).

There is a third way of dealing with exceptions, used when the application requires an unusually fine degree of control. By enabling (setting) bits in the Floating Point Status and Control Register (FPSCR), the application can cause the PowerPC microprocessor to generate hardware interrupt signals when a floating-point exception occurs. These bits are set through the use of the `fesetenvd` call. The Mac OS X will deliver that interrupt to your application by raising the UNIX signal `SIGFPE`.

Your application can specify how this interrupt is to be handled by calling the sigaction function with a first parameter of `SIGFPE`. When a floating-point operation causes and exception, the associated interrupt will trigger a special handler (installed with the same sigaction call) which is executed before the operation returns a result. The handler (which may be a default system handler or a user-defined handler) has access to a great deal of information about the system (including the floating-point registers) at the time the exception occurs, and may make adjustments as desired.

sigaction is part of the standard UNIX signal-handling facilities, as defined by IEEE Std1003.1-1988. For more information see `http://developer.apple.com/documentation/Darwin/Reference/ManPages/html/sigaction.2.html`. When an exception handler is invoked, the parameters delivered to it include machine-specific (usually PPC-specific) data about the state of the floating-point environment. For details about the data structures, see the header files `ucontext.h` and `thread_status.h`.

Here is sample exception handler (`myHandler`), and a program (`main`) which installs it and generates some floating-point exceptions to invoke it. In this case the sample exception handler simply prints out (using `printf`) the address at which it was invoked and the value of the PowerPC FPSCR before reurning control to the main program.

**Listing 7-1** Sample Exception Handler

```
#include <math.h>
#include <fenv.h>
#include <signal.h>
#include <ucontext.h>
#include <mach/thread_status.h>


#define fegetenvd(x) asm volatile("mffs %0" : "=f" (x));
#define fesetenvd(x) asm volatile("mtfsf 255,%0" : : "f" (x));
enum {
   FE_ENABLE_INEXACT    = 0x00000008,
   FE_ENABLE_DIVBYZERO  = 0x00000010,
   FE_ENABLE_UNDERFLOW  = 0x00000020,
   FE_ENABLE_OVERFLOW   = 0x00000040,
   FE_ENABLE_INVALID    = 0x00000080,
   FE_ENABLE_ALL_EXCEPT = 0x000000F8
};

typedef union {
   struct {
      unsigned long hi;
      unsigned long lo;
   } i;
   double d;
} hexdouble;

void myHandler(sig, sip, scp)
   int sig;
   siginfo_t *sip;
   struct ucontext *scp;
   {
      hexdouble t;
      ppc_float_state_t *fs;
      ppc_thread_state_t *ss;

      fs = &scp->uc_mcontext->fs;
      ss = &scp->uc_mcontext->ss;

      printf("SIGFPE taken at 0x%x invokes myHandler, fpscr = %08X\n",
               sip->si_addr, fs->fpscr);

         /* Re-arms interrupts when this state is restored */
      fs->fpscr &= FE_ENABLE_ALL_EXCEPT;

            /* Advances the PC when this state is restored */
      ss->srr0 += 4;

      printf("fpscr = %08X\n", fs->fpscr);
}

static struct sigaction act = { myHandler, (sigset_t)0, SA_SIGINFO };

main ()
{
   float s;
   hexdouble t;

   fegetenvd(t.d);

            /* Enable hardware trapping for all exceptions */
   t.i.lo |= FE_ENABLE_ALL_EXCEPT;
   fesetenvd(t.d);

            /* Set handler */

   if (sigaction(SIGFPE, &act, (struct sigaction *)0) != 0) {
      perror("Yikes");
      exit(-1);
   }
   fegetenvd(t.d);

            /* Overflow folded out by compiler */
   s = HUGE_VALF * HUGE_VALF;
            /* Inf/Inf raises invalid operation */
   s = s / (1.0 + s);
   fegetenvd(t.d);

            /* verify that the compiler is doing the right thing */
   printf("In main(1), s computed as: %e , fpscr = %08X\n", s, t.i.lo);

            /* Overflow folded out by compiler */
   s = HUGE_VALF * HUGE_VALF;

            /* Inf/Inf raises invalid operation */
   s = s / (2.0 + s);
   fegetenvd(t.d);

            /* verify that the compiler is doing the right thing */
   printf("In main(2), s computed as: %e , fpscr = %08X\n", s, t.i.lo);
}
```

# Environmental Controls Summary

This section summarizes the C constants, macros, functions, and type definitions associated with controlling the floating-point environment.

## C Summary

### Constants

#### Rounding Direction Modes

```
#define  FE_TONEAREST      0x00000000
#define  FE_TOWARDZERO     0x00000001
#define  FE_UPWARD         0x00000002
#define  FE_DOWNWARD       0x00000003
```

#### Floating-Point Exception Flags

```
#define  FE_INEXACT        0x02000000     /* inexact */
#define  FE_DIVBYZERO      0x04000000     /* divide-by-zero */
#define  FE_UNDERFLOW      0x08000000     /* underflow */
#define  FE_OVERFLOW       0x10000000     /* overflow */
#define  FE_INVALID        0x20000000     /* invalid */

#define  FE_ALL_EXCEPT     (  FE_INEXACT | FE_DIVBYZERO | FE_UNDERFLOW | \
                              FE_OVERFLOW | FE_INVALID )

#define  FE_DFL_ENV        &_FE_DFL_ENV   /* pointer to default environment*/
```

### Data Types

```
typedef     long        fenv_t;
typedef     long        fexcept_t;
```

## Environment Access Routines

### Controlling the Rounding Direction

```
int fegetround              (void);
int fesetround              (int round);
```

### Controlling the Exception Flags

```
void feclearexcept          (int excepts);
void fegetexceptflag        (fexcept_t *flagp, int excepts);
void feraiseexcept          (int excepts);
void fesetexceptflag        (const fexcept_t *flagp, int excepts);
int fetestexcept            (int excepts);
```

### Accessing the Floating-Point Environment

```
void fegetenv               (fenv_t *envp);
int feholdexcept            (fenv_t *envp);
void fesetenv               (const fenv_t *envp);
void feupdateenv            (const fenv_t *envp);
```

# Conversion Functions

## Contents

CHAPTER 8

# Conversion Functions

This chapter describes how you can perform the conversions required by the IEEE standard using Libm C functions. For each type of conversion, this chapter lists the functions you can use to perform that conversion. It shows the declarations of these functions, describes what they do, describes when they raise floating-point exceptions, and gives examples of how to use them. For a description of the conversions required by the IEEE standard and the details of how each conversion is performed in Mac OS X numerics, see Chapter 4, "Conversions." All of the conversion function declarations appear in the file `math.h`.

## Converting Floating-Point to Integer Formats

In C, the default method of converting floating-point numbers to integers is to simply discard the fractional part (truncate). Libm provides two functions that convert floating-point numbers to integers using methods other than the default C method and that return the integers in integer types.

| | |
|---|---|
| llrint($x$) | Returns the nearest integer to $x$ in the current rounding direction as type `long long int`. |
| lrint($x$) | Adds 1/2 to the magnitude of $x$, chops to an integer, and returns the value as type `long long int`. |
| lrint($x$) | Returns the nearest integer to $x$ in the current rounding direction as type `long int`. |
| lround($x$) | Adds 1/2 to the magnitude of $x$, chops to an integer, and returns the value as type `long int`. |

### llrint

You can use the `llrint` function to round a real number to the nearest integer (of type `long long int`) in the current rounding direction.

```
long long int llrint (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `llrint` function rounds its argument to the nearest integer (of type `long long int`) in the current rounding direction and places the result in a `long long int` type. The available rounding directions are upward, downward, to nearest, and toward zero.

`llrint` differs from `rint` (described on page 5-13) in that it returns the value in type `long int`; `rint` returns the value in a floating-point type. It differs from `lrint` in its return type as well (`long long int` rather than `long int`).

**EXCEPTIONS**

When *x* is finite and nonzero, either the result of `llrint`(*x*) is exact or it raises one of the following exceptions:

- inexact (if *x* is not an integer)
- invalid (if the integer result is outside the range of the `long long int` type)

**SPECIAL CASES**

Table 8-2 shows the results when the argument to the `llrint` function is a zero, a NaN, or an Infinity.

**Table 8-1** Special cases for the `llrint` function

| Operation | Result | Exceptions raised |
|---|---|---|
| llrint(+0) | +0 | None |
| llrint(–0) | –0 | None |
| llrint(NaN) | Undefined | Invalid |
| llrint(+∞) | Undefined | Invalid |
| llrint(–∞) | Undefined | Invalid |

**EXAMPLES**

```
z = llrint(+INFINITY);/* z = unspecified value for all rounding
                        directions because +INFINITY exceeds the
                        range of long long int. The invalid
                        exception is raised. */
z = lrint(300.1);      /* z = 301 if rounding direction is upward
                        else z = 300. The inexact exception is
                        raised.*/
z = lrint(–300.1);     /* z = –301 if rounding direction is
                        downward else z = –300. The inexact
                        exception is raised. */
```

## lrint

You can use the `lrint` function to round a real number to the nearest integer (of type `long int`) in the current rounding direction.

```
long int lrint (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `lrint` function rounds its argument to the nearest integer (of type `long int`) in the current rounding direction and places the result in a `long int` type. The available rounding directions are upward, downward, to nearest, and toward zero.

The `lrint` function provides the floating-point to integer conversion as described in the IEEE standard. It differs from `rint` (described on page 5-13) in that it returns the value in type `long int`; `rint` returns the value in a floating-point type. It differs from `llrint` in its return type as well (`long int` rather than `long long int`).

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\mathrm{lrint}(x)$ is exact or it raises one of the following exceptions:

- inexact (if $x$ is not an integer)
- invalid (if the integer result is outside the range of the `long int` type)

**SPECIAL CASES**

Table 8-2 shows the results when the argument to the `lrint` function is a zero, a NaN, or an Infinity.

**Table 8-2** Special cases for the `lrint` function

| Operation | Result | Exceptions raised |
|---|---|---|
| lrint(+0) | +0 | None |
| lrint(–0) | –0 | None |
| lrint(NaN) | Undefined | Invalid |
| lrint(+∞) | Undefined | Invalid |
| lrint(–∞) | Undefined | Invalid |

**EXAMPLES**

```
z = lrint(+INFINITY);/* z = unspecified value for all rounding
                        directions because +INFINITY exceeds the
                        range of long int. The invalid exception
                        is raised. */
z = lrint(300.1);      /* z = 301 if rounding direction is upward
                        else z = 300. The inexact exception is
                        raised.*/
z = lrint(–300.1);     /* z = –301 if rounding direction is
                        downward else z = –300. The inexact
                        exception is raised. */
```

## llround

You can use the `llround` function to round a real number to the nearest integer value (of type `long long int`) by adding 1/2 to the magnitude and truncating.

```
long long int llround (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The llround function adds 1/2 to the magnitude of its argument and chops to integer, returning the answer in `long long int` type.

`llround` differs from `round` (described on page 8-12) in that it returns the value in type `long long int`; `round` returns the value in a floating-point type. It differs from `lround` in its return type as well (`long long int` rather than `long int`).

This function is not affected by the current rounding direction. Notice that the `llround` function rounds halfway cases (1.5, 2.5, and so on) away from 0. With the default rounding direction, `llrint` (described on page 8-3) rounds halfway cases to the even integer.

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\text{llround}(x)$ is exact or it raises one of the following exceptions:

- inexact (if $x$ is not an integer)
- invalid (if the integer result is outside the range of the `long long int` type)

**SPECIAL CASES**

Table 8-4 shows the results when the argument to the `llround` function is a zero, a NaN, or an Infinity.

**Table 8-3** Special cases for the `llround` function

| Operation | Result | Exceptions raised |
|---|---|---|
| llround(+0) | +0 | None |
| llround(–0) | –0 | None |
| llround(NaN) | Undefined | Invalid |
| llround(+∞) | Undefined | Invalid |
| llround(–∞) | Undefined | Invalid |

**EXAMPLES**

```
z = llround(+INFINITY);     /* z = an unspecified value because
                               +∞ is outside of the range of long
                               long int. */
z = llround(0.5);           /* z = 1 because |0.5| + 0.5 = 1.0. The
                               inexact exception is raised. */
z = llround(–0.9);          /* z = –1 because |–0.9| + 0.5 = 1.4.
                               The inexact exception is raised. */
```

## lround

You can use the `lround` function to round a real number to the nearest integer value (of type `long int`) by adding 1/2 to the magnitude and truncating.

```
long int lround (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `lround` function adds 1/2 to the magnitude of its argument and chops to integer, returning the answer in `long int` type.

`lround` differs from `round` (described on page 8-12) in that it returns the value in type `long int`; `round` returns the value in a floating-point type. It differs from `llround` in its return type as well (`long int` rather than `long long int`).

This function is not affected by the current rounding direction. Notice that the `lround` function rounds halfway cases (1.5, 2.5, and so on) away from 0. With the default rounding direction, `lrint` (described on page 8-5) rounds halfway cases to the even integer.

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\text{lround}(x)$ is exact or it raises one of the following exceptions:

- inexact (if $x$ is not an integer)
- invalid (if the integer result is outside the range of the `long int` type)

**SPECIAL CASES**

Table 8-4 shows the results when the argument to the `roundtol` function is a zero, a NaN, or an Infinity.

**Table 8-4** Special cases for the `lround` function

| Operation | Result | Exceptions raised |
|---|---|---|
| lround(+0) | +0 | None |
| lround(–0) | –0 | None |
| lround(NaN) | Undefined | Invalid |
| lround(+∞) | Undefined | Invalid |
| lround(–∞) | Undefined | Invalid |

**EXAMPLES**

```
z = lround(+INFINITY);     /* z = an unspecified value because
                              +∞ is outside of the range of long
                              int. */
z = lround(0.5);           /* z = 1 because |0.5| + 0.5 = 1.0. The
                              inexact exception is raised. */
z = lround(–0.9);          /* z = –1 because |–0.9| + 0.5 = 1.4.
                              The inexact exception is raised. */
```

## Rounding Floating-Point Numbers to Integers

Libm provides six functions that convert floating-point numbers to integers and return the integer in a floating-point type. The first is the `rint` function, which performs the round-to-integer operation as described in Chapter 5, "Numeric Operations and Functions." The other functions either round in a specific direction or perform a variation of the `rint` operation.

| | |
|---|---|
| ceil($x$) | Returns the nearest integer not less than $x$. |
| floor($x$) | Returns the nearest integer not greater than $x$. |
| nearbyint($x$) | Returns the nearest integer to $x$ in the current rounding direction. |
| round($x$) | Adds 1/2 to the magnitude of $x$ and chops to an integer. |
| trunc($x$) | Truncates the fractional part of $x$. |

## ceil

You can use the `ceil` function to round a real number upward to the nearest integer value.

```
double_t ceil (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `ceil` function rounds its argument upward. This is an ANSI standard C library function. The result is returned in a floating-point data type.

This function is the same as performing the following code sequence:

```
r = fegetround();        /* save current rounding direction */
fesetround(FE_UPWARD);   /* round upward */
rint(x);                 /* round to integer */
fesetround(r);           /* restore rounding direction */
```

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of ceil($x$) is exact.

**SPECIAL CASES**

Table 8-5 shows the results when the argument to the `ceil` function is a zero, a NaN, or an Infinity.

**Table 8-5** Special cases for the `ceil` function

| Operation | Result | Exceptions raised |
|---|---|---|
| ceil(+0) | +0 | None |
| ceil(–0) | –0 | None |
| ceil(NaN) | NaN | None[*] |
| ceil(+∞) | +∞ | None |
| ceil(–∞) | –∞ | None |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = ceil(+INFINITY); /* z = +INFINITY because +INFINITY is already
                        an integer value by definition. */
z = ceil(300.1);     /* z = 301.0 */
z = ceil(–300.1);    /* z = –300.0 */
```

## floor

You can use the `floor` function to round a real number downward to the next integer value.

```
double_t floor (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `floor` function rounds its argument downward. This is an ANSI standard C library function. The result is returned in a floating-point data type.

This function is the same as performing the following code sequence:

```
r = fegetround();          /* save current rounding direction */
fesetround(FE_DOWNWARD);   /* round downward */
rint(x);                   /* round to integer */
fesetround(r);             /* restore rounding direction */
```

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\text{floor}(x)$ is exact.

**SPECIAL CASES**

Table 8-6 shows the results when the argument to the `floor` function is a zero, a NaN, or an Infinity.

**Table 8-6** Special cases for the `floor` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\text{floor}(+0)$ | +0 | None |
| $\text{floor}(-0)$ | –0 | None |
| $\text{floor}(\text{NaN})$ | NaN | None* |
| $\text{floor}(+\infty)$ | $+\infty$ | None |
| $\text{floor}(-\infty)$ | $-\infty$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = floor(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = floor(300.1);       /* z = 300.0 */
z = floor(–300.1);      /* z = –301.0 */
```

## nearbyint

You can use the `nearbyint` function to round a real number to the nearest integer in the current rounding direction.

```
double_t nearbyint (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `nearbyint` function rounds its argument to the nearest integer in the current rounding direction. The available rounding directions are upward, downward, to nearest, and toward zero.

The `nearbyint` function provides the floating-point to integer conversion described in the IEEE Standard 854. It differs from `rint` (described on page 5-13) only in that it does not raise the inexact flag when the argument is not already an integer.

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of nearbyint($x$) is exact.

**SPECIAL CASES**

Table 8-7 shows the results when the argument to the `nearbyint` function is a zero, a NaN, or an Infinity.

**Table 8-7** Special cases for the `nearbyint` function

| Operation | Result | Exceptions raised |
|---|---|---|
| nearbyint(+0) | +0 | None |
| nearbyint(–0) | –0 | None |
| nearbyint(NaN) | NaN | None* |
| nearbyint(+∞) | +∞ | None |
| nearbyint(–∞) | –∞ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = nearbyint(+INFINITY);  /* z = +INFINITY for all rounding
                              directions. */
z = nearbyint(300.1);      /* z = 301.0 if rounding direction is
                              upward, else z = 300.0. */
z = nearbyint(–300.1);     /* z = –301.0 if rounding direction is
                              downward, else z = –300.0. */
```

## round

You can use the `round` function to round a real number to the integer value obtained by adding 1/2 to the magnitude and truncating.

```
double_t round (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `round` function adds 1/2 to the magnitude of its argument and chops to integer. The result is returned in a floating-point data type.

This function is not affected by the current rounding direction. Notice that the `round` function rounds halfway cases (1.5, 2.5, and so on) away from 0. With the default rounding direction, `rint` (described on page 5-13) rounds halfway cases to the even integer.

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\text{round}(x)$ is exact or it raises the following exception:

- inexact (if $x$ is not an integer value)

**SPECIAL CASES**

Table 8-8 shows the results when the argument to the `round` function is a zero, a NaN, or an Infinity.

**Table 8-8** Special cases for the `round` function

| Operation | Result | Exceptions raised |
|---|---|---|
| round(+0) | +0 | None |
| round(–0) | –0 | None |
| round(NaN) | NaN | None* |
| round(+∞) | +∞ | None |
| round(–∞) | –∞ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = round(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = round(0.5);         /* z = 1.0 because |0.5| + 0.5 = 1.0. The
                           inexact exception is raised. */
z = round(–0.9);        /* z = –1.0 because |–0.9| + 0.5 = 1.4.
                           The inexact exception is raised. */
```

## trunc

You can use the `trunc` function to truncate the fractional part of a real number so that just the integer part remains.

```
double_t trunc (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `trunc` function chops off the fractional part of its argument. This is an ANSI standard C library function.

This function is the same as performing the following code sequence:

```
r = fegetround();            /* save current rounding direction */
fesetround(FE_TOWARDZERO); /* round toward zero */
rint(x);                     /* round to integer */
fesetround(r);               /* restore rounding direction */
```

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\mathrm{trunc}(x)$ is exact.

**SPECIAL CASES**

Table 8-9 shows the results when the argument to the `trunc` function is a zero, a NaN, or an Infinity.

**Table 8-9** Special cases for the `trunc` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\mathrm{trunc}(+0)$ | +0 | None |
| $\mathrm{trunc}(-0)$ | –0 | None |
| $\mathrm{trunc}(\mathrm{NaN})$ | NaN | None* |
| $\mathrm{trunc}(+\infty)$ | $+\infty$ | None |
| $\mathrm{trunc}(-\infty)$ | $-\infty$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = trunc(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = trunc(300.1);       /* z = 300.0 */
z = trunc(–300.1);      /* z = –300.0 */
```

## Converting Integers to Floating-Point Formats

In the C programming language, conversions from integers stored in an integer format to floating-point formats are automatic when you assign an integer to a floating-point variable.

```
double d;
int x = 1;
d = x;      /* value 1 automatically converted to double format */
```

## Converting Between Floating-Point Formats

In the C programming language, conversions between floating-point formats are automatic when you assign a floating-point number of one type to a variable of another type.

```
float f = 0.0f;         /* single format */
double d = 1.1;
long double ld;         /* double-double format */

f = d;      /* double 1.1 converted to single format */
ld = f;     /* single 1.1 converted to double-double format */
d = ld;     /* double-double 1.1 converted to double format */
```

## Conversions Summary

This section summarizes the C functions associated with converting floating-point values.

## C Summary

### Conversion Routines

#### Converting Floating-Point Formats to Integer Formats

```
long int lrint              (double_t x);
long int lround             (double_t x);
```

#### Rounding Floating-Point Numbers to Integers

```
double_t ceil               (double_t x);
double_t floor              (double_t x);
double_t nearbyint          (double_t x);
double_t round              (double_t x);
double_t trunc              (double_t x);
```

# Transcendental Functions

## Contents

# Transcendental Functions

This chapter describes how to use the transcendental and auxiliary functions declared in Libm. This chapter describes the following types of functions:

- comparison
- sign manipulation
- exponential
- logarithmic
- trigonometric
- hyperbolic
- error and gamma

It shows the declarations of these functions, describes what they do, describes when they raise floating-point exceptions, and gives examples of how to use them. For functions that manipulate the floating-point environment, see Chapter 7, "Environmental Control Functions." For functions that perform conversions, see Chapter 8, "Conversion Functions." For basic arithmetic and comparison operations, see Chapter 5, "Numeric Operations and Functions."

## Comparison Functions

Libm provides four functions that perform comparisons between two floating-point arguments:

| | |
|---|---|
| fdim($x$, $y$) | Returns the positive difference $x - y$ or 0. |
| fmax($x$, $y$) | Returns the maximum of $x$ or $y$. |
| fmin($x$, $y$) | Returns the minimum of $x$ or $y$. |

These functions take advantage of the rule from the IEEE standard that all values except NaNs have an order:

$$-\infty < \text{all negative real numbers} < -0 = +0 < \text{all positive real numbers} < +\infty$$

These functions also make special cases of NaNs so that they raise no floating-point exceptions.

## fdim

You can use the `fdim` function to determine the positive difference between two real numbers.

```
double_t fdim (double_t x, double_t y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The `fdim` function returns the positive difference between its two arguments.

$$\begin{array}{ll} \mathrm{fdim}(x, y) = x - y & \text{if } x > y \\ \mathrm{fdim}(x, y) = +0 & \text{if } x \le y \end{array}$$

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero and $x > y$, either the result of $\mathrm{fdim}(x, y)$ is exact or it raises one of the following exceptions:

- inexact (if the result of $x - y$ must be rounded)
- overflow (if the result of $x - y$ is outside the range of the data type)
- underflow (if the result of $x - y$ is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-1 shows the results when one of the arguments to the `fdim` function is a zero, a NaN, or an Infinity. In this table, $x$ and $y$ are finite, nonzero floating-point numbers.

**Table 9-1** Special cases for the `fdim` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\mathrm{fdim}(+0, y)$ | +0 | None |
| $\mathrm{fdim}(x, +0)$ | $x$ | None |
| $\mathrm{fdim}(-0, y)$ | +0 | None |
| $\mathrm{fdim}(x, -0)$ | $x$ | None |
| $\mathrm{fdim}(\mathrm{NaN}, y)$ | NaN[*] | None[†] |
| $\mathrm{fdim}(x, \mathrm{NaN})$ | NaN | None[†] |

**Table 9-1** Special cases for the `fdim` function (continued)

| Operation | Result | Exceptions raised |
|---|---|---|
| $\mathrm{fdim}(+\infty, y)$ | $+\infty$ | None |
| $\mathrm{fdim}(x, +\infty)$ | $+0$ | None |
| $\mathrm{fdim}(-\infty, y)$ | $+0$ | None |
| $\mathrm{fdim}(x, -\infty)$ | $+\infty$ | None |

* If both arguments are NaN, the first NaN is returned.
† If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = fdim(+INFINITY, 300);  /* z = +∞ – 300 = +INFINITY because
                              +∞ > 300 */
z = fdim(300, +INFINITY);  /* z = +0 because 300 ≤ +∞ */
```

## fmax

You can use the `fmax` function to find out which is the larger of two real numbers.

```
double_t fmax (double_t x, double_t y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The `fmax` function determines the larger of its two arguments.

$$\mathrm{fmax}(x, y) = x \quad \text{if } x \geq y$$
$$\mathrm{fmax}(x, y) = y \quad \text{if } x < y$$

If one of the arguments is a NaN, the other argument is returned.

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero, the result of $\mathrm{fmax}(x, y)$ is exact.

**SPECIAL CASES**

Table 9-2 shows the results when one of the arguments to the `fmax` function is a zero, a NaN, or an Infinity. In this table, *x* is a finite, nonzero floating-point number. (Note that the order of operands for this function does not matter.)

**Table 9-2** Special cases for the `fmax` function

| Operation | Result | Exceptions raised |
|---|---|---|
| fmax(+0, $x$) | $x$ if $x > 0$ | None |
| | +0 if $x < 0$ | None |
| fmax(–0, $x$) | $x$ if $x > 0$ | None |
| | –0 if $x < 0$ | None |
| fmax(–0, ±0) | -0 | None |
| fmax(+0, ±0) | +0 | None |
| fmax(NaN, $x$) | $x$* | None† |
| fmax(+∞, $x$) | +∞ | None |
| fmax(–∞, $x$) | $x$ | None |

* If both arguments are NaNs, the first NaN is returned.
† If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = fmax(–INFINITY, –300,000);    /* z = –300,000 because any
                                     integer is greater than –∞ */
z = fmax(NAN, –300,000);    /* z = –300,000 by definition of the
                               function fmax. */
```

## fmin

You can use the `fmin` function to determine which is the smaller of two real numbers.

```
double_t fmin (double_t x, double_t y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The `fmin` function determines the lesser of its two arguments.

$$\begin{array}{ll} \text{fmin}(x, y) = x & \text{if } x \le y \\ \text{fmin}(x, y) = y & \text{if } y < x \end{array}$$

If one of the arguments is a NaN, the other argument is returned.

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero, the result of $\text{fmin}(x, y)$ is exact.

**SPECIAL CASES**

Table 9-3 shows the results when one of the arguments to the `fmin` function is a zero, a NaN, or an Infinity. In this table, $x$ is a finite, nonzero floating-point number. (Note that the order of operands for this function does not matter.)

**Table 9-3** Special cases for the `fmin` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\text{fmin}(+0, x)$ | $x$ if $x < 0$ | None |
| | $+0$ if $x > 0$ | None |
| $\text{fmin}(-0, x)$ | $x$ if $x < 0$ | None |
| | $-0$ if $x > 0$ | None |
| $\text{fmin}(-0, \pm 0)$ | $-0$ | None |
| $\text{fmin}(+0, \pm 0)$ | $+0$ | None |
| $\text{fmin}(\text{NaN}, x)$ | $x$* | None† |
| $\text{fmin}(+\infty, x)$ | $x$ | None |
| $\text{fmin}(-\infty, x)$ | $-\infty$ | None |

* If both arguments are NaNs, the first NaN is returned.
† If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = fmin(–INFINITY, –300,000);   /* z = –INFINITY because –∞ is
                                       smaller than any integer. */
z = fmin(NAN, –300,000);   /* z = –300,000 by definition of the
                               function fmin. */
```

## Sign Manipulation Functions

Libm provides two functions that manipulate the sign bit of the floating-point value:

| | |
|---|---|
| copysign$(x, y)$ | Copies the sign of $y$ to $x$. |
| fabs$(x)$ | Returns the absolute value (positive form) of $x$. |

Because these functions only manipulate the sign bit of the value and do not try to compute the value at all, they raise no floating-point exceptions.

### copysign

You can use the `copysign` function to assign to some real number the sign of a second value.

```
double_t copysign (double_t x, double_t y);
long double copysignl (long double x, long double y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The `copysign` function copies the sign of the `y` parameter onto the magnitude of the `x` parameter and returns the resulting number.

`copysign(x, 1.0)` is always the absolute value of `x`. The `copysign` function simply manipulates sign bits and hence raises no exception flags.

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero, the result of copysign$(x, y)$ is exact.

**SPECIAL CASES**

Table 9-4 shows the results when one of the arguments to the `copysign` function is a zero, a NaN, or an Infinity. In this table, *x* and *y* are finite, nonzero floating-point numbers.

**Table 9-4** Special cases for the `copysign` function

| Operation | Result | Exceptions raised |
|---|---|---|
| copysign$(+0, y)$ | 0 with sign of $y$ | None |
| copysign$(x, +0)$ | $\lvert x \rvert$ | None |
| copysign$(-0, y)$ | 0 with sign of $y$ | None |
| copysign$(x, -0)$ | $-\lvert x \rvert$ | None |
| copysign(NaN, $y$) | NaN with sign of $y$ | None* |
| copysign($x$, NaN) | $x$ with sign of NaN | None* |
| copysign$(+\infty, y)$ | $\infty$ with sign of $y$ | None |
| copysign$(x, +\infty)$ | $\lvert x \rvert$ | None |
| copysign$(-\infty, y)$ | $\infty$ with sign of $y$ | None |
| copysign$(x, -\infty)$ | $-\lvert x \rvert$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = copysign(–1234.567, 1.0);/* z = 1234.567 */
z = copysign(1.0, –1234.567);/* z = –1.0 */
```

## fabs

You can use the `fabs` function to determine the absolute value of a real number.

```
double_t fabs (double_t x);
long double fabsl (long double x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `fabs` function returns the absolute value (positive value) of its argument.

$$\text{fabs}(x) = |x|$$

This function looks only at the sign bit, not the value, of its argument.

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of fabs($x$) is exact.

**SPECIAL CASES**

Table 9-5 shows the results when the argument to the `fabs` function is a zero, a NaN, or an Infinity.

**Table 9-5** Special cases for the `fabs` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| fabs(+0) | +0 | None |
| fabs(–0) | +0 | None |
| fabs(NaN) | NaN | None[*] |
| fabs(+∞) | +∞ | None |
| fabs(–∞) | +∞ | None |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = fabs(–1.0);   /* z = 1 */
z = fabs(245.0);  /* z = 245 */
```

## Exponential Functions

Libm provides six exponential functions:

| | |
|---|---|
| exp($x$) | The base $e$ or natural exponential $e^x$. |
| exp2($x$) | The base 2 exponential $2^x$. |
| expm1($x$) | The base $e$ exponential minus 1. |
| ldexp($x$, $n$) | Returns $x \times 2^n$ (equivalent to scalb). |
| pow($x$, $y$) | Returns $x^y$. |
| scalbn($x$, $n$) | Returns $x \times 2^n$. |

## exp

You can use the `exp` function to raise $e$ to some power.

```
double_t exp (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `exp` function performs the exponential function on its argument.

$$\exp(x) = e^x$$

The `log` function performs the inverse operation $(\ln x)$.

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\exp(x)$ might raise the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-6 shows the results when the argument to the `exp` function is a zero, a NaN, or an Infinity.

**Table 9-6** Special cases for the `exp` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\exp(+0)$ | +1 | None |
| $\exp(-0)$ | +1 | None |
| $\exp(\text{NaN})$ | NaN | None* |
| $\exp(+\infty)$ | $+\infty$ | None |
| $\exp(-\infty)$ | +0 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = exp(0.0);  /* z = e^0 = 1. */
z = exp(1.0);  /* z = e^1 ≈ 2.71828128...The inexact exception is
                  raised. */
```

## exp2

You can use the `exp2` function to raise 2 to some power.

```
double_t exp2 (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `exp2` function returns the base 2 exponential of its argument.

$$\exp2(x) = 2^x$$

The `log2` function performs the inverse operation $(\log_2 x)$ .

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of exp2($x$) might raise the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-7 shows the results when the argument to the `exp2` function is a zero, a NaN, or an Infinity.

**Table 9-7** Special cases for the `exp2` function

| Operation | Result | Exceptions raised |
|---|---|---|
| exp2(+0) | +1 | None |
| exp2(–0) | +1 | None |
| exp2(NaN) | NaN | None* |
| exp2(+∞) | +∞ | None |
| exp2(–∞) | +0 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = exp2(2.0); /* z = 2^2 = 4. The inexact exception is raised. */
z = exp2(1.5); /* z = 2^1.5 ≈ 2.82843. The inexact exception is
                  raised. */
```

## expm1

You can use the `expm1` function to raise $e$ to some power and subtract 1.

```
double_t expm1 (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

## DESCRIPTION

The `expm1` function returns the natural exponential decreased by 1.

$$\text{expm1}(x) = e^x - 1$$

For small numbers, use the function call `expm1(x)` instead of the expression

```
exp(x) – 1
```

The call `expm1(x)` produces a more exact result because it avoids the roundoff error that might occur when the expression is computed.

## EXCEPTIONS

When $x$ is finite and nonzero, the result of $\text{expm1}(x)$ might raise the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

## SPECIAL CASES

Table 9-8 shows the results when the argument to the `expm1` function is a zero, a NaN, or an Infinity.

**Table 9-8** Special cases for the `expm1` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\text{expm1}(+0)$ | +0 | None |
| $\text{expm1}(-0)$ | –0 | None |
| $\text{expm1}(\text{NaN})$ | NaN | None* |
| $\text{expm1}(+\infty)$ | $+\infty$ | None |
| $\text{expm1}(-\infty)$ | –1 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

## EXAMPLES

```
z = expm1(–2.1);     /* z = e^–2.1 – 1 = –0.877544. The inexact
                        exception is raised. */
z = expm1(6);        /* z = e^6 – 1 = 402.429. The inexact
                        exception is raised. */
```

## ldexp

You can use the `ldexp` function to perform efficient scaling by a power of 2.

```
double_t ldexp (double_t x, int n);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `n` | An integer representing a power of 2 by which `x` should be multiplied. |

**DESCRIPTION**

The `ldexp` function computes the value $x \times 2^n$ without computing $2^n$. This is an ANSI standard C library function.

$$\text{ldexp}(x, n) = x \times 2^n$$

The `scalb` function (described on page 9-18) performs the same operation as this function. The `frexp` function performs the inverse operation; that is, it splits `x` into its fraction field and exponent field.

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\text{ldexp}(x, n)$ is exact or it raises one of the following exceptions:

- inexact (if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-9 shows the results when the floating-point argument to the `ldexp` function is a zero, a NaN, or an Infinity. In this table, $n$ is any integer.

**Table 9-9** Special cases for the `ldexp` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $\text{ldexp}(+0, n)$ | +0 | None |
| $\text{ldexp}(-0, n)$ | –0 | None |
| $\text{ldexp}(\text{NaN}, n)$ | NaN | None* |
| $\text{ldexp}(+\infty, n)$ | +∞ | None |
| $\text{ldexp}(-\infty, n)$ | –∞ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = ldexp(3.0, 3);   /* z = 3 × 2³ = 24 */
z = ldexp(0.0, 3);   /* z = 0 × 2³ = 0 */
```

## pow

You can use the pow function to raise a real number to the power of some other real number.

```
double_t pow (double_t x, double_t y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The pow function computes `x` to the `y` power. This is an ANSI standard C library function.

$$\text{pow}(x, y) = x^y$$

Use the function call `pow(x,y)` instead of the expression

```
exp(y * log(x))
```

The call `pow(x,y)` produces a more exact result.

There are some differences between this implementation and the behavior of the pow function in a SANE implementation. For example, in SANE `pow(NAN,0)` returns a NaN, whereas in PowerPC Numerics, `pow(NAN,0)` returns a 1.

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero, either the result of $\text{pow}(x, y)$ is exact or it raises one of the following exceptions:

- inexact (if $y$ is not an integer or an underflow or overflow occurs)
- invalid (if $x$ is negative and $y$ is not an integer)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-10 shows the results when one of the arguments to the pow function is a zero, a NaN, or an Infinity, plus other special cases for the pow function. In this table, $x$ and $y$ are finite, nonzero floating-point numbers.

**Table 9-10** Special cases for the pow function

| Operation | Result | Exceptions raised |
|---|---|---|
| pow($x$, $y$) for $x < 0$ | NaN if $y$ is not integer | Invalid |
| | $x^y$ if $y$ is integer | None |
| pow(+0, $y$) | +0 if $y$ is > 0 | None |
| | +∞ if $y < 0$ | Divide-by-zero |
| pow($x$, +0) | +1 | None |
| pow(–0, $y$) | -0 if $y$ is odd integer > 0 | None |
| | +0 if $y > 0$ but not odd integer | None |
| | -∞ if $y$ is odd integer < 0 | Divide-by-zero |
| | +∞ if $y < 0$ but not odd integer | Divide-by-zero |
| pow($x$, –0) | +1 | None |
| pow(NaN, $y$) | NaN if $y \neq 0$ | None* |
| | +1 if $y = 0$ | None* |
| pow($x$, NaN) | NaN | None* |
| pow(+∞, $y$) | +∞ if $y > 0$ | None |
| | +0 if $y < 0$ | None |
| | +1 if $y = 0$ | None |
| pow($x$, +∞) | +∞ if $\lvert x \rvert > 1$ | None |
| | +0 if $\lvert x \rvert < 1$ | None |
| | 1 if $\lvert x \rvert = 1$ | Invalid |
| pow(–∞, $y$) | –∞ if $y$ is odd integer > 0 | None |
| | +∞ if $y > 0$ but not odd integer | None |
| | –0 if $y$ is odd integer < 0 | None |
| | +0 if $y < 0$ but not odd integer | None |
| | +1 if $y = 0$ | None |
| pow($x$, –∞) | +0 if $\lvert x \rvert > 1$ | None |
| | +∞ if $\lvert x \rvert < 1$ | None |
| | 1 if $\lvert x \rvert = 1$ | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = pow(NAN, 0);      /* z = 1 */
```

## scalbn

You can use the `scalbn` function to perform efficient scaling by a power of 2.

```
double_t scalbn (double_t x, long int n);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `n` | An integer representing a power of 2 by which `x` should be multiplied. |

**DESCRIPTION**

The `scalbn` function performs efficient scaling of its floating-point argument by a power of 2.

$$\text{scalbn}(x, n) = x \times 2^n$$

Using the `scalbn` function is more efficient than performing the actual arithmetic.

This function performs the same operation as the `ldexp` transcendental function described on page 9-15.

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\text{scalbn}(x, n)$ is exact or it raises one of the following exceptions:

- inexact (if the result causes an overflow or underflow exception)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-11 shows the results when the floating-point argument to the `scalbn` function is a zero, a NaN, or an Infinity. In this table, $n$ is any integer.

**EXAMPLES**

```
z = scalbn(1, 3); /* z = 1 × 2^3 = 8 */
```

**Table 9-11** Special cases for the `scalb` function

| Operation | Result | Exceptions raised |
|---|---|---|
| scalbn(+0, $n$) | +0 | None |
| scalbn(–0, $n$) | –0 | None |
| scalbn(NaN, $n$) | NaN | None* |
| scalbn(+∞, $n$) | +∞ | None |
| scalbn(–∞, $n$) | –∞ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

## Logarithmic Functions

MathLib provides seven logarithmic functions:

| | |
|---|---|
| frexp($x$, $exp$) | Splits $x$ into fraction and exponent fields. |
| log($x$) | Base $e$ or natural logarithm. |
| log10($x$) | Base 10 logarithm. |
| log1p($x$) | Computes $\log(1+x)$. |
| log2($x$) | Base 2 logarithm. |
| logb($x$) | Returns exponent part of $x$. |
| modf($x$, $iptr$) | Splits $x$ into an integer and a fraction. |

### frexp

You can use the `frexp` function to find out the values of a floating-point number's fraction field and exponent field.

```
double_t frexp (double_t x, int *exponent);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `exponent` | A pointer to an integer in which the value of the exponent can be returned. |

**DESCRIPTION**

The `frexp` function splits its first argument into a fraction part and a base 2 exponent part. This is an ANSI standard C library function.

$$\text{frexp}(x, n) = f \quad \text{such that } x = f \times 2^n$$

or

$$\text{frexp}(x, n) = f \quad \text{such that } \; n = (1 + \text{logb}(x)) \; \text{ and } \; f = \text{scalb}(x, -n)$$

The return value of `frexp` is the value of the fraction field of the argument `x`. The exponent field of `x` is stored in the address pointed to by the `exponent` argument.

For finite nonzero inputs, `frexp` returns either 0.0 or a value whose magnitude is between 0.5 and 1.0.

The `ldexp` and `scalb` functions perform the inverse operation (compute $f \times 2^n$).

**EXCEPTIONS**

If $x$ is finite and nonzero, the result of $\text{frexp}(x, n)$ is exact.

**SPECIAL CASES**

Table 9-12 shows the results when the input argument to the `frexp` function is a zero, a NaN, or an Infinity.

**Table 9-12** Special cases for the `frexp` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\text{frexp}(+0, n)$ | $+0\ (n = 0)$ | None |
| $\text{frexp}(-0, n)$ | $-0\ (n = 0)$ | None |
| $\text{frexp}(\text{NaN}, n)$ | NaN ($n$ is undefined) | None* |
| $\text{frexp}(+\infty, n)$ | $+\infty$ ($n$ is undefined) | None |
| $\text{frexp}(-\infty, n)$ | $-\infty$ ($n$ is undefined) | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = frexp(2E300, n);    /* z ≈ 0.746611 and n = 998. In other
                           words, 2 × 10^300 ≈ 0.746611 × 2^998. */
```

## log

You can use the `log` function to compute the natural logarithm of a real number.

```
double_t log (double_t x);
```

| | |
|---|---|
| `x` | Any positive floating-point number. |

**DESCRIPTION**

The `log` function returns the natural (base *e*) logarithm of its argument.

$$\log(x) = \log_e x = \ln x = y \quad \text{such that } x = e^y$$

The `exp` function performs the inverse (exponential) operation.

**EXCEPTIONS**

When *x* is finite and nonzero, the result of $\log(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x* other than +1)
- invalid (if *x* is negative)

**SPECIAL CASES**

Table 9-13 shows the results when the argument to the `log` function is a zero, a NaN, or an Infinity, plus other special cases for the `log` function.

**Table 9-13** Special cases for the `log` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\log(x)$ for $x < 0$ | NaN | Invalid |
| $\log(+1)$ | +0 | None |
| $\log(+0)$ | $-\infty$ | Divide-by-zero |
| $\log(-0)$ | $-\infty$ | Divide-by-zero |
| $\log(\text{NaN})$ | NaN | None[*] |
| $\log(+\infty)$ | $+\infty$ | None |
| $\log(-\infty)$ | NaN | Invalid |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = log(+1.0);     /* z = +0.0 because e^0 = 1 */
z = log(–1.0);     /* z = NAN because negative arguments are not
                      allowed. The invalid exception is raised. */
```

## log10

You can use the `log10` function to compute the common logarithm of a real number.

```
double_t log10 (double_t x);
```

| | |
|---|---|
| `x` | Any positive floating-point number. |

**DESCRIPTION**

The `log10` function returns the common (base 10) logarithm of its argument.

$$\log 10(x) = \log_{10} x = y \text{ such that } x = 10^y$$

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\log 10(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$ other than +1)
- invalid (when $x$ is negative)

**SPECIAL CASES**

Table 9-14 shows the results when the argument to the `log10` function is a zero, a NaN, or an Infinity, plus other special cases for the `log10` function.

**Table 9-14** Special cases for the `log10` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $\log 10(x)$ for $x < 0$ | NaN | Invalid |
| $\log 10(+1)$ | +0 | None |
| $\log 10(+0)$ | –∞ | Divide-by-zero |
| $\log 10(-0)$ | –∞ | Divide-by-zero |
| $\log 10(\text{NaN})$ | NaN | None* |
| $\log 10(+\infty)$ | +∞ | None |
| $\log 10(-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = log10(+1.0);  /* z = 0.0 because 10^0 = 1 */
z = log10(10.0);  /* z = 1.0 because 10^1 = 10. The inexact
                     exception is raised. */
z = log10(–1.0);  /* z = NAN because negative arguments are not
                     allowed. The invalid exception is raised. */
```

## log1p

You can use the `log1p` function to compute the natural logarithm of 1 plus a real number.

```
double_t log1p (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number greater than –1. |

**DESCRIPTION**

The `log1p` function computes the natural logarithm of 1 plus its argument.

$$\log 1\mathrm{p}(x) = \log_e(x+1) = \ln(x+1) = y \quad \text{such that } 1+x = 10^y$$

For small numbers, use the function call `log1p(x)` instead of the function call `log(1 + x)`. The call `log1p(x)` produces a more exact result because it avoids the roundoff error that might occur when the expression `1 + x` is computed.

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\log 1\mathrm{p}(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x > –1$)
- invalid (when $x$ is less than –1)
- divide-by-zero (when $x$ is –1)

## SPECIAL CASES

Table 9-15 shows the results when the argument to the `log1p` function is a zero, a NaN, or an Infinity, plus other special cases for the `log1p` function.

**Table 9-15** Special cases for the `log1p` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\log 1p(x)$ for $x < -1$ | NaN | Invalid |
| $\log 1p(-1)$ | –∞ | Divide-by-zero |
| $\log 1p(+0)$ | +0 | None |
| $\log 1p(-0)$ | –0 | None |
| $\log 1p(\text{NaN})$ | NaN | None* |
| $\log 1p(+\infty)$ | +∞ | None |
| $\log 1p(-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

## EXAMPLES

```
z = log1p(–1.0);  /* z = log(0) = –INFINITY. The divide-by-zero
                     and inexact exceptions are raised. */
z = log1p(0.0);   /* z = log(1) = 0.0 because e^0 = 1. */
z = log1p(–2.0);  /* z = log(–1) = NAN because logarithms of
                     negative numbers are not allowed. The
                     invalid exception is raised. */
```

## log2

You can use the `log2` function to compute the binary logarithm of a real number.

```
double_t log2 (double_t x);
```

| | |
|---|---|
| `x` | Any positive floating-point number. |

### DESCRIPTION

The `log2` function returns the binary (base 2) logarithm of its argument.

$\log 2(x) = \log_2 x = y$ such that $x = 2^y$

The `exp2` function performs the inverse operation.

### EXCEPTIONS

When $x$ is finite and nonzero, the result of $\log 2(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$ other than +1)
- invalid (when $x$ is negative)

### SPECIAL CASES

Table 9-16 shows the results when the argument to the `log2` function is a zero, a NaN, or an Infinity, plus other special cases for the `log2` function.

**Table 9-16** Special cases for the `log2` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $\log 2(x)$ for $x < 0$ | NaN | Invalid |
| $\log 2(+1)$ | +0 | None |
| $\log 2(+0)$ | –∞ | Divide-by-zero |
| $\log 2(-0)$ | –∞ | Divide-by-zero |
| $\log 2(\text{NaN})$ | NaN | None[*] |
| $\log 2(+\infty)$ | +∞ | None |
| $\log 2(-\infty)$ | NaN | Invalid |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = log2(+1.0);     /* z = +0 because 2^0 = 1 */
z = log2(2.0);      /* z = 1 because 2^1 = 2. The inexact exception
                       is raised. */
z = log2(–1.0);     /* z = NAN because negative arguments are not
                       allowed. The invalid exception is raised. */
```

## logb

You can use the `logb` function to determine the value in the exponent field of a floating-point number.

```
double_t logb (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `logb` function returns the signed exponent of its argument *x* as a floating-point value.

$\log b(x) = y$ such that $x = f \times 2^y$

When the argument is a denormalized number, the exponent is determined as if the input argument had first been normalized.

Note that for a nonzero finite *x*, $1 \le \text{fabs}(\text{scalb}(x, -\log b(x))) < 2$ .

That is, for a nonzero finite *x*, the magnitude of *x* taken to the power of its inverse exponent is between 1 and 2.

This function conforms to IEEE Standard 854, which differs from IEEE Standard 754 on the treatment of a denormalized argument *x*.

**EXCEPTIONS**

If *x* is finite and nonzero, the result of $\log b(x)$ is exact.

#### SPECIAL CASES

Table 9-17 shows the results when the argument to the `logb` function is a zero, a NaN, or an Infinity.

**Table 9-17** Special cases for the `logb` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\operatorname{logb}(+0)$ | $-\infty$ | Divide-by-zero |
| $\operatorname{logb}(-0)$ | $-\infty$ | Divide-by-zero |
| $\operatorname{logb}(\mathrm{NaN})$ | NaN | None* |
| $\operatorname{logb}(+\infty)$ | $+\infty$ | None |
| $\operatorname{logb}(-\infty)$ | $+\infty$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

#### EXAMPLES

```
z = logb(789.9);  /* z = 9.0 because 789.9 ≈ 1.54 × 2^9 */
z = logb(21456789);/* z = 24.0 because 21456789 ≈ 1.28 × 2^24 */
```

## ilogb

`ilogb` returns the value in the exponent field of a floating-point number. `ilogb` is similar to `logb`, but it returns the value as a signed integer (rather than as a floating-point value.)

int logb (double_t x);

| | |
|---|---|
| `x` | Any floating-point number. |

#### DESCRIPTION

The `ilogb` function returns the signed exponent of its argument $x$ as an integer value.

$$\operatorname{logb}(x) = y \text{ such that } x = f \times 2^y$$

When the argument is a denormalized number, the exponent is determined as if the input argument had first been normalized.

Note that for a nonzero finite $x$, $1 \le \operatorname{fabs}(\operatorname{scalb}(x, -\log(x))) < 2$ .

That is, for a nonzero finite $x$, the magnitude of $x$ taken to the power of its inverse exponent is between 1 and 2.

This function conforms to IEEE Standard 9899.

**EXCEPTIONS**

If $x$ is finite and nonzero, the result of ilogb($x$) is exact.

**SPECIAL CASES**

Table 9-17 shows the results when the argument to the `ilogb` function is a zero, a NaN, or an Infinity.

**Table 9-18** Special cases for the `logb` function

| Operation | Result | Exceptions raised |
|---|---|---|
| ilogb(+0) | -MAX_INT | None |
| ilogb(-0) | -MAX_INT | None |
| ilogb(NaN) | MAX_INT | None* |
| ilogb(+∞) | MAX_INT | None |
| ilogb(-∞) | MAX_INT | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = logb(789.9);  /* z = 9 because 789.9 ≈ 1.54 × 2^9 */
z = logb(21456789);/* z = 24 because 21456789 ≈ 1.28 × 2^24 */
```

## modf

You can use the `modf` function to split a real number into a fractional part and an integer part.

```
float modff (float x, float *iptrf);
double modf (double x, double *iptr);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `iptr` | A pointer to a floating-point variable in which the integer part can be stored upon return. |

**DESCRIPTION**

The `modf` function splits its first argument into a fractional part and an integer part. This is an ANSI standard C function.

$$\text{modf}(x, n) = f \text{ such that } |f| < 1.0 \text{ and } f + n = x$$

The fractional part is returned as the value of the function, and the integer part is stored as a floating-point number in the area pointed to by `iptr`. The fractional part and the integer part both have the same sign as the argument `x`.

**EXCEPTIONS**

If $x$ is finite and nonzero, the result of $\text{modf}(x, n)$ is exact.

**SPECIAL CASES**

Table 9-19 shows the results when the floating-point argument to the `modf` function is a zero, a NaN, or an Infinity.

**Table 9-19** Special cases for the `modf` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\text{modf}(+0, n)$ | $+0\ (n = 0)$ | None |
| $\text{modf}(-0, n)$ | $-0\ (n = 0)$ | None |
| $\text{modf}(\text{NaN}, n)$ | $\text{NaN}\ (n = \text{NaN})$ | None* |
| $\text{modf}(+\infty, n)$ | $+0\ (n = +\infty)$ | None |
| $\text{modf}(-\infty, n)$ | $-0\ (n = -\infty)$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = modf(1.0, n); /* z = 0.0 and n = 1.0 */
z = modf(+INFINITY, n); /* z = 0.0 and n = +INFINITY because the
                           value +∞ is an integer. */
```

# Trigonometric Functions

Libm provides the following **trigonometric functions:**

| | |
|---|---|
| $\cos(x)$ | Computes the cosine of x. |
| $\sin(x)$ | Computes the sine of $x$. |
| $\tan(x)$ | Computes the tangent of $x$. |
| $\operatorname{acos}(x)$ | Computes the arc cosine of $x$. |
| $\operatorname{asin}(x)$ | Computes the arc sine of $x$. |
| $\operatorname{atan}(x)$ | Computes the arc tangent of $x$. |
| $\operatorname{atan2}(y, x)$ | Computes the arc tangent of $y/x$. |

The remaining trigonometric functions can be computed easily and efficiently from the transcendental functions provided.

The arguments for trigonometric functions (`cos`, `sin`, and `tan`) and return values for inverse trigonometric functions (`acos`, `asin`, `atan`, and `atan2`) are expressed in radians. The cosine, sine, and tangent functions use an argument reduction based on the `remainder` function (see page 5-11) and the constant `pi`, where `pi` is the nearest approximation of π with 53 bits of precision. The cosine, sine, and tangent functions are periodic with respect to the constant `pi`, so their periods are different from their mathematical counterparts and diverge from their counterparts when their arguments become very large.

## cos

You can use the `cos` function to compute the cosine of a real number.

```
double_t cos (double_t x);
```

| | |
|---|---|
| `x` | Any finite floating-point number. |

### DESCRIPTION

The `cos` function returns the cosine of its argument. The argument is the measure of an angle expressed in radians. This function is **symmetric** with respect to the y-axis ($\cos x = \cos -x$).

The `acos` function performs the inverse operation $(\arccos(y))$ .

### EXCEPTIONS

When $x$ is finite and nonzero, $\cos(x)$ raises the inexact exception.

SPECIAL CASES

Table 9-20 shows the results when the argument to the `cos` function is a zero, a NaN, or an Infinity, plus other special cases for the `cos` function.

**Table 9-20** Special cases for the `cos` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\cos(\pi)$ | –1 | Inexact |
| $\cos(+0)$ | 1 | None |
| $\cos(-0)$ | 1 | None |
| $\cos(\mathrm{NaN})$ | NaN | None* |
| $\cos(+\infty)$ | NaN | Invalid |
| $\cos(-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = cos(0);     /* z = 1.0. */
z = cos(pi/2); /* z = –0.0. The inexact exception is raised. */
z = cos(pi);   /* z = –1.0. The inexact exception is raised. */
z = cos(–pi/2);/* z = 0.0. The inexact exception is raised. */
z = cos(–pi);  /* z = –1.0. The inexact exception is raised. */
```

## sin

You can use the `sin` function to compute the sine of a real number.

```
double_t sin (double_t x);
```

| | |
|---|---|
| `x` | Any finite floating-point number. |

DESCRIPTION

The `sin` function returns the sine of its argument. The argument is the measure of an angle expressed in radians. This function is **antisymmetric** with respect to the y-axis ($\sin x = -\sin -x$).

The `asin` function performs the inverse operation $(\arcsin(y))$.

#### EXCEPTIONS

When $x$ is finite and nonzero, the result of $\sin(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

#### SPECIAL CASES

Table 9-21 shows the results when the argument to the `sin` function is a zero, a NaN, or an Infinity, plus other special cases for the `sin` function.

**Table 9-21** Special cases for the `sin` function

| **Operation** | **Result** | **Exceptions raised** |
|---|---|---|
| $\sin(\pi)$ | 0 | Inexact |
| $\sin(+0)$ | +0 | None |
| $\sin(-0)$ | –0 | None |
| $\sin(\text{NaN})$ | NaN | None* |
| $\sin(+\infty)$ | NaN | Invalid |
| $\sin(-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

#### EXAMPLES

```
z = sin(pi/2);     /* z = 1. The inexact exception is raised. */
z = sin(pi);       /* z = 0. The inexact exception is raised. */
z = sin(–pi/2);    /* z = –1. The inexact exception is raised. */
z = sin(–pi);      /* z = 0. The inexact exception is raised. */
```

## tan

You can use the `tan` function to compute the tangent of a real number.

```
double_t tan (double_t x);
```

| | |
|---|---|
| `x` | Any finite floating-point number. |

**DESCRIPTION**

The `tan` function returns the tangent of its argument. The argument is the measure of an angle expressed in radians. This function is antisymmetric.

The `atan` function performs the inverse operation $(\arctan(y))$ .

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\tan(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-22 shows the results when the argument to the `tan` function is a zero, a NaN, or an Infinity, plus other special cases for the `tan` function.

**Table 9-22** Special cases for the `tan` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\tan(\pi)$ | 0 | Inexact |
| $\tan(\pi/2)$ | +∞ | Inexact |
| $\tan(+0)$ | +0 | None |
| $\tan(-0)$ | –0 | None |
| $\tan(\mathrm{NaN})$ | NaN | None* |
| $\tan(+\infty)$ | NaN | Invalid |
| $\tan(-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = tan(pi);   /* z = 0. The inexact exception is raised. */
z = tan(pi/2); /* z = +INFINITY. The inexact exception is
                  raised. */
z = tan(pi/4); /* z = 1. The inexact exception is raised. */
```

## acos

You can use the `acos` function to compute the arc cosine of a real number between –1 and +1.

```
double_t acos (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number in the range $-1 \le x \le 1$. |

**DESCRIPTION**

The `acos` function returns the arc cosine of its argument $x$. The return value is expressed in radians in the range [0, π].

$$\operatorname{acos}(x) = \arccos(x) = y \quad \text{such that} \quad \cos(y) = x \ \text{ for } -1 \le x \le 1$$

The `cos` function performs the inverse operation $(\cos(y))$ .

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\operatorname{acos}(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$ other than 1)
- invalid (if $|x|>1$)

**SPECIAL CASES**

Table 9-23 shows the results when the argument to the `acos` function is a zero, a NaN, or an Infinity, plus other special cases for the `acos` function.

**Table 9-23** Special cases for the `acos` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $\operatorname{acos}(x)$ for $\lvert x\rvert > 1$ | NaN | Invalid |
| $\operatorname{acos}(-1)$ | π | Inexact |
| $\operatorname{acos}(+1)$ | +0 | None |
| $\operatorname{acos}(+0)$ | π/2 | Inexact |
| $\operatorname{acos}(-0)$ | π/2 | Inexact |
| $\operatorname{acos}(\text{NaN})$ | NaN | None* |
| $\operatorname{acos}(+\infty)$ | NaN | Invalid |
| $\operatorname{acos}(-\infty)$ | NaN | Invalid |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = acos(1.0);     /* z = arccos (1) = 0.0 */
z = acos(–1.0);    /* z = arccos (–1) = π. The inexact exception is
                      raised. */
```

## asin

You can use the `asin` function to compute the arc sine of a real number between –1 and 1.

```
double_t asin (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number in the range $-1 \le x \le 1$. |

**DESCRIPTION**

The `asin` function returns the arc sine of its argument. The return value is expressed in radians in the range $[-\pi/2, +\pi/2]$. This function is antisymmetric.

$$\operatorname{asin}(x) = \arcsin(x) = y \quad \text{such that} \quad \sin(y) = x \quad \text{for } -1 \le x \le 1$$

The `sin` function performs the inverse operation $(\sin(y))$.

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\operatorname{asin}(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- invalid (if $|x| > 1$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-24 shows the results when the argument to the `asin` function is a zero, a NaN, or an Infinity, plus other special cases for the `asin` function.

**Table 9-24** Special cases for the `asin` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\operatorname{asin}(x)$ for $\lvert x \rvert > 1$ | NaN | Invalid |
| $\operatorname{asin}(-1)$ | $-\pi/2$ | Inexact |
| $\operatorname{asin}(+1)$ | $\pi/2$ | Inexact |
| $\operatorname{asin}(+0)$ | +0 | None |
| $\operatorname{asin}(-0)$ | –0 | None |
| asin(NaN) | NaN | None[*] |
| $\operatorname{asin}(+\infty)$ | NaN | Invalid |
| $\operatorname{asin}(-\infty)$ | NaN | Invalid |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = asin(1.0);      /* z = arcsin 1 = π/2. The inexact exception is
                       raised. */
z = asin(–1.0);     /* z = arcsin –1 = –π/2. The inexact exception
                       is raised. */
```

## atan

You can use the `atan` function to compute the arc tangent of a real number.

```
double_t atan (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `atan` function returns the arc tangent of its argument. The return value is expressed in radians in the range $[-\pi/2, +\pi/2]$. This function is antisymmetric.

$$\operatorname{atan}(x) = \arctan(x) = y \quad \text{such that } \tan(y) = x \text{ for all } x$$

The tan function performs the inverse operation $(\tan(y))$ .

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\text{atan}(x)$ might raise one of the following exceptions:

- inexact (for all nonzero values of $x$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-25 shows the results when the argument to the atan function is a zero, a NaN, or an Infinity.

**Table 9-25** Special cases for the atan function

| Operation | Result | Exceptions raised |
|---|---|---|
| atan(+0) | +0 | None |
| atan(–0) | –0 | None |
| atan(NaN) | NaN | None[*] |
| atan(+∞) | +π/2 | Inexact |
| atan(–∞) | –π/2 | Inexact |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = atan(1.0);     /* z = arctan 1 = π/4 */
z = atan(–1.0);    /* z = arctan –1 = –π/4. The inexact exception
                      is raised. */
```

## atan2

You can use the atan2 function to compute the arc tangent of a real number divided by another real number.

```
double_t atan2 (double_t y, double_t x);
```

| | |
|---|---|
| y | Any floating-point number. |
| x | Any floating-point number. |

**DESCRIPTION**

The `atan2` function returns the arc tangent of its first argument divided by its second argument. The return value is expressed in radians in the range [–π, +π], using the signs of its operands to determine the quadrant.

$$\operatorname{atan2}(y, x) = \arctan(y/x) = z \quad \text{such that} \quad \tan(z) = y/x$$

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero, the result of $\operatorname{atan2}(y, x)$ might raise one of the following exceptions:

- inexact (if either $x$ or $y$ is any finite, nonzero value)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-26 shows the results when one of the arguments to the `atan2` function is a zero, a NaN, or an Infinity. In this table, $x$ and $y$ are finite, nonzero floating-point numbers.

**Table 9-26** Special cases for the `atan2` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\operatorname{atan2}(+0, x)$ | $+0 \quad x > 0$ | None |
| | $+\pi \quad x < 0$ | None |
| $\operatorname{atan2}(y, +0)$ | $+\pi/2 \quad y > 0$ | None |
| | $-\pi/2 \quad y < 0$ | None |
| $\operatorname{atan2}(\pm 0, +0)$ | $\pm 0$ | None |
| $\operatorname{atan2}(-0, x)$ | $-0 \quad x > 0$ | Inexact |
| | $-\pi \quad x < 0$ | Inexact |
| $\operatorname{atan2}(y, -0)$ | $+\pi/2 \quad y > 0$ | None |
| | $-\pi/2 \quad y < 0$ | None |
| $\operatorname{atan2}(\pm 0, -0)$ | $\pm\pi$ | Inexact |
| $\operatorname{atan2}(\text{NaN}, x)$ | NaN* | None† |
| $\operatorname{atan2}(y, \text{NaN})$ | NaN | None† |
| $\operatorname{atan2}(+\infty, x)$ | $\pi/2$ | Inexact |
| $\operatorname{atan2}(\pm y, +\infty)$ | $\pm 0 \quad y > 0$ | None |
| $\operatorname{atan2}(\pm\infty, +\infty)$ | $\pm 3\pi/4$ | Inexact |

**Table 9-26** Special cases for the `atan2` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\text{atan2}(-\infty, x)$ | $-\pi/2$ | Inexact |
| $\text{atan2}(\pm y, -\infty)$ | $\pm\pi \quad y > 0$ | None |
| $\text{atan2}(\pm\infty, -\infty)$ | $\pm 3\pi/4$ | Inexact |

* If both arguments are NaNs, it is undefined which one `atan2` returns.
† If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = atan2(1.0, 1.0); /* z = arctan 1/1 = arctan 1 = π/4. The
                        inexact exception is raised. */
z = atan2(3.5, 0.0); /* z = arctan 3.5/0 = arctan ∞ = π/2 */
```

## Hyperbolic Functions

Libm provides hyperbolic and inverse hyperbolic functions.

| | |
|---|---|
| $\cosh(x)$ | Hyperbolic cosine of $x$. |
| $\sinh(x)$ | Hyperbolic sine of $x$. |
| $\tanh(x)$ | Hyperbolic tangent of $x$. |
| $\text{acosh}(x)$ | Inverse hyperbolic cosine of $x$. |
| $\text{asinh}(x)$ | Inverse hyperbolic sine of $x$. |
| $\text{atanh}(x)$ | Inverse hyperbolic tangent of $x$. |

These functions are based on other transcendental functions and defer most exception generation to the core functions they use.

### cosh

You can use the `cosh` function to compute the hyperbolic cosine of a real number.

```
double_t cosh (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `cosh` function returns the hyperbolic cosine of its argument. This function is symmetric.

The `acosh` function performs the inverse operation $(\text{arccosh}(y))$ .

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\cosh(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- overflow (if the result is outside the range of the data type)

**SPECIAL CASES**

Table 9-27 shows the results when the argument to the `cosh` function is a zero, a NaN, or an Infinity.

**Table 9-27** Special cases for the `cosh` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\cosh(+0)$ | +1 | None |
| $\cosh(-0)$ | +1 | None |
| $\cosh(\text{NaN})$ | NaN | None* |
| $\cosh(+\infty)$ | $+\infty$ | None |
| $\cosh(-\infty)$ | $+\infty$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = cosh(1.0);    /* z ≈ 1.54308. The inexact exception is
                     raised. */
z = cosh(–1.0);   /* z ≈ 1.54308. The inexact exception is
                     raised. */
```

## sinh

You can use the `sinh` function to compute the hyperbolic sine of a real number.

```
double_t sinh (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `sinh` function returns the hyperbolic sine of its argument. This function is antisymmetric.

The `asinh` function performs the inverse operation $(\text{arcsinh}(y))$.

**EXCEPTIONS**

When *x* is finite and nonzero, the result of $\sinh(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x*)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-28 shows the results when the argument to the `sinh` function is a zero, a NaN, or an Infinity.

**Table 9-28** Special cases for the `sinh` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\sinh(+0)$ | +0 | None |
| $\sinh(-0)$ | –0 | None |
| $\sinh(\mathrm{NaN})$ | NaN | None* |
| $\sinh(+\infty)$ | $+\infty$ | None |
| $\sinh(-\infty)$ | $-\infty$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
sinh(1.0);  /* z ≈ 1.175201. The inexact exception is raised. */
sinh(–1.0); /* z ≈ –1.175201. The inexact exception is raised. */
```

## tanh

You can use the `tanh` function to compute the hyperbolic tangent of a real number.

```
double_t tanh (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `tanh` function returns the hyperbolic tangent of its argument. The return value is in the range [–1, +1]. This function is antisymmetric.

The `atanh` function performs the inverse operation $(\operatorname{arctanh}(y))$ .

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\tanh(x)$ raises the following exception:

- inexact (for all finite, nonzero values of $x$)

**SPECIAL CASES**

Table 9-29 shows the results when the argument to the `tanh` function is a zero, a NaN, or an Infinity.

**Table 9-29** Special cases for the `tanh` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\tanh(+0)$ | +0 | None |
| $\tanh(-0)$ | –0 | None |
| $\tanh(\text{NaN})$ | NaN | None[*] |
| $\tanh(+\infty)$ | +1 | None |
| $\tanh(-\infty)$ | –1 | None |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = tanh(1.0);     /* z ≈ 0.761594. The inexact exception is
                      raised. */
z = tanh(–1.0);    /* z ≈ 0.761594. The inexact exception is
                      raised. */
```

## acosh

You can use the `acosh` function to compute the inverse hyperbolic cosine of a real number.

```
double_t acosh (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number in the range $1 \le x \le +\infty$. |

**DESCRIPTION**

The `acosh` function returns the inverse hyperbolic cosine of its argument. This function is antisymmetric.

$$\text{acosh}(x) = \text{arccosh x} = y \quad \text{such that cosh } y = x$$

The `cosh` function performs the inverse operation $(\cosh(y))$ .

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\text{acosh}(x)$ might raise one of the following exceptions:

- inexact (for all finite values of $x > 1$)
- invalid (if $x < 1$)

**SPECIAL CASES**

Table 9-30 shows the results when the argument to the `acosh` function is a zero, a NaN, or an Infinity, plus other special cases for the `acosh` function.

**Table 9-30** Special cases for the `acosh` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| $\text{acosh}(x)$ for $x < 1$ | NaN | Invalid |
| $\text{acosh}(1)$ | +0 | None |
| $\text{acosh}(+0)$ | NaN | Invalid |
| $\text{acosh}(-0)$ | NaN | Invalid |
| $\text{acosh}(\text{NaN})$ | NaN | None* |
| $\text{acosh}(+\infty)$ | +∞ | None |
| $\text{acosh}(-\infty)$ | NaN | Invalid |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = acosh(1.0);    /* z = +0 */
z = acosh(0.0);    /* z = NAN. The invalid exception is raised. */
```

## asinh

You can use the `asinh` function to compute the inverse hyperbolic sine of a real number.

```
double_t asinh (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `asinh` function returns the inverse hyperbolic sine of its argument. This function is antisymmetric.

$$\operatorname{asinh}(x) = \operatorname{arcsinh} x = y \quad \text{such that } \sinh y = x$$

The `sinh` function performs the inverse operation $(\sinh(y))$.

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\operatorname{asinh}(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-31 shows the results when the argument to the `asinh` function is a zero, a NaN, or an Infinity.

**Table 9-31** Special cases for the `asinh` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\operatorname{asinh}(+0)$ | +0 | None |
| $\operatorname{asinh}(-0)$ | –0 | None |
| $\operatorname{asinh}(\text{NaN})$ | NaN | None* |
| $\operatorname{asinh}(+\infty)$ | +∞ | None |
| $\operatorname{asinh}(-\infty)$ | –∞ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = asinh(1.0);   /* z ≈ 0.881374. The inexact exception is
                     raised. */
z = asinh(–1.0);  /* z ≈ 0.881374. The inexact exception is
                     raised. */
```

## atanh

You can use the `atanh` function to perform the inverse hyperbolic tangent of a real number.

```
double_t atanh (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number in the range $-1 \le x \le 1$. |

**DESCRIPTION**

The `atanh` function returns the inverse hyperbolic tangent of its argument. This function is antisymmetric.

$$\operatorname{atanh}(x) = \operatorname{arctanh} x = y \quad \text{such that } \tanh y = x$$

The `tanh` function performs the inverse operation $(\tanh(y))$.

**EXCEPTIONS**

When $x$ is finite and nonzero, the result of $\operatorname{atanh}(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x$ other than +1 and –1)
- invalid (if $|x| > 1$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-32 shows the results when the argument to the `atanh` function is a zero, a NaN, or an Infinity, plus other special cases for the `atanh` function.

**Table 9-32** Special cases for the `atanh` function

| Operation | Result | Exceptions raised |
|---|---|---|
| atanh($x$) for $\lvert x \rvert > 1$ | NaN | Invalid |
| atanh(–1) | –∞ | Divide-by-zero |
| atanh(+1) | +∞ | Divide-by-zero |
| atanh(+0) | +0 | None |
| atanh(–0) | –0 | None |
| atanh(NaN) | NaN | None[*] |
| atanh(+∞) | NaN | Invalid |
| atanh(–∞) | NaN | Invalid |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = atanh(1.0);   /* z = +INFINITY */
z = atanh(–1.0);  /* z = –INFINITY */
```

## Error and Gamma Functions

`Libm` provides four error and gamma functions:

| | |
|---|---|
| erf($x$) | Error function |
| erfc($x$) | Complementary error function |
| gamma($x$) | Computes $\Gamma(x)$ |
| lgamma($x$) | Computes the natural logarithm of the absolute value of gamma($x$) |

### erf

You can use the `erf` function to perform the error function.

```
double_t erf (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `erf` function computes the error function of its argument. This function is antisymmetric.

$$\mathrm{erf}(x) = \frac{2}{\pi}\int_0^x e^{(-t)^2}dt$$

**EXCEPTIONS**

When *x* is finite and nonzero, either the result of erf(*x*) is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an underflow occurs)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-33 shows the results when the argument to the `erf` function is a zero, a NaN, or an Infinity.

**Table 9-33** Special cases for the `erf` function

| Operation | Result | Exceptions raised |
|---|---|---|
| erf(+0) | +0 | None |
| erf(–0) | –0 | None |
| erf(NaN) | NaN | None* |
| erf(+∞) | +1 | None |
| erf(–∞) | –1 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = erf(1.0);      /* z ≈ 0.842701. The inexact exception is
                      raised. */
z = erf(–1.0);     /* z ≈ –0.842701. The inexact exception is
                      raised. */
```

## erfc

You can use the `erfc` function to perform the complementary error function.

```
double_t erfc (double_t x);
```

| | |
|---|---|
| `x` | Any floating-point number. |

**DESCRIPTION**

The `erfc` function computes the complementary error of its argument. This function is antisymmetric.

$$\operatorname{erfc}(x) = 1.0 - \operatorname{erf}(x)$$

For large positive numbers (around 10), use the function call `erfc(x)` instead of the expression `1.0 – erf(x)`. The call `erfc(x)` produces a more exact result.

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\operatorname{erfc}(x)$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an underflow occurs)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-34 shows the results when the argument to the `erfc` function is a zero, a NaN, or an Infinity.

**Table 9-34** Special cases for the `erfc` function

| Operation | Result | Exceptions raised |
|---|---|---|
| $\operatorname{erfc}(+0)$ | +1 | None |
| $\operatorname{erfc}(-0)$ | +1 | None |
| $\operatorname{erfc}(\mathrm{NaN})$ | NaN | None[*] |
| $\operatorname{erfc}(+\infty)$ | +0 | None |
| $\operatorname{erfc}(-\infty)$ | +2 | None |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = erfc(–INFINITY); /* z = 1 – erf(–∞ ) = 1 – –1 = +2.0 */
z = erfc(0.0);       /* z = 1 – erf(0) = 1 – 0 = 1.0 */
```

## gamma

You can use the `gamma` function to perform $\Gamma(x)$ .

```
double_t gamma (double_t x);
```

| | |
|---|---|
| `x` | Any positive floating-point number. |

**DESCRIPTION**

The `gamma` function performs $\Gamma(x)$ .

$$\text{gamma}(x) = \Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

The `gamma` function reaches overflow very fast as *x* approaches +∞. For large values, use the `lgamma` function (described in the next section) instead.

**EXCEPTIONS**

When *x* is finite and nonzero, either the result of $\text{gamma}(x)$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an overflow occurs)
- invalid (if *x* is a negative integer)
- overflow (if the result is outside the range of the data type)

**SPECIAL CASES**

Table 9-35 shows the results when the argument to the `gamma` function is a zero, a NaN, or an Infinity, plus other special cases for the `gamma` function.

**Table 9-35** Special cases for the `gamma` function

| Operation | Result | Exceptions raised |
|---|---|---|
| gamma($x$) for negative integer $x$ | NaN | Invalid |
| gamma(+0) | +∞ | Divide-by-zero |
| gamma(–0) | -∞ | Divide-by-zero |
| gamma(NaN) | NaN | None[*] |
| gamma(+∞) | +∞ | None |
| gamma(–∞) | NaN | Invalid |

[*] If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = gamma(–1.0);  /* z = NAN. The invalid exception is raised. */
z = gamma(6);     /* z = 120 */
```

## lgamma

You can use the `lgamma` function to compute the natural logarithm of the absolute value of $\Gamma(x)$ .

```
double_t lgamma (double_t x);
```

| | |
|---|---|
| `x` | Any positive floating-point number. |

**DESCRIPTION**

The `lgamma` function computes the natural logarithm of the absolute value of $\Gamma(x)$ .

$$\mathrm{lgamma}(x) = \log_e(|\Gamma(x)|) = \ln(|\Gamma(x)|)$$

**EXCEPTIONS**

When $x$ is finite and nonzero, either the result of $\mathrm{lgamma}(x)$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an overflow occurs)
- overflow (if the result is outside the range of the data type)
- invalid (if $x \leq 0$)

**SPECIAL CASES**

Table 9-36 shows the results when the argument to the `lgamma` function is a zero, a NaN, or an Infinity, plus other special cases for the `lgamma` function.

**Table 9-36** Special cases for the `lgamma` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| lgamma($x$) for x a negative integer | $+\infty$ | Divide-by-zero |
| lgamma(+0) | $+\infty$ | Divide-by-zero |
| lgamma(–0) | $+\infty$ | Divide-by-zero |
| lgamma(NaN) | NaN | None* |
| lgamma($+\infty$) | $+\infty$ | None |
| lgamma($-\infty$) | $+\infty$ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = lgamma(–1.0);    /* z = NAN. The invalid exception is
                        raised. */
z = lgamma(3.41);    /* z = 1.10304. The inexact exception is
                        raised. */
```

## Bessel Functions

`Libm` provides six Bessel functions:

| | |
|---|---|
| j0($x$) | Bessel function of the first kind of order 0 |
| j1($x$) | Bessel function of the first kind of order 1 |
| jn($n$,$x$) | Bessel function of the first kind of integer order $n$ |

| | |
|---|---|
| y0(*x*) | Linearly independent Bessel function of the second kind of order 0 |
| y1(*x*) | Linearly independent Bessel function of the second kind of order 1 |
| yn(*n*,*x*) | Linearly independent Bessel function of the second kind of integer order *n* |

## j0

You can use the `j0` function to calculate the Bessel function of the first kind of order 0 for argument *x*.

```
double j0 (double x);
x  Any floating-point number.
```

### DESCRIPTION

The `j0` function calculates the Bessel function of the first kind of order 0 for argument *x*.

### EXCEPTIONS

When *x* and *y* are finite and nonzero, j0(*x*) raises the inexact flag. It may also cause the following exceptions:

- overflow (if *x* is finite and the result is infinite)
- underflow (if the result is inexact, must be represented as a denormalized number or 0)

### SPECIAL CASES

Table 9-41 shows the results when the argument to the j0 function is a zero, a NaN, or an infinity.

**Table 9-37** Special cases for the j0 function

| Operation | Result | Exceptions raised |
|---|---|---|
| j0(-0) | 1 | None |
| j0(+0) | 1 | None |
| j0(NaN) | NaN | None* |
| j0(-∞) | +0 | None |
| j0(+∞) | +0 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

#### EXAMPLES

```
z = j0(0.0);            /* z = 1.0
z = j0(1.0);            /* z = 7.651976865579666e-01
```

## j1

You can use the `j1` function to calculate the Bessel function of the first kind of order 1 for argument *x*.

```
double j1 (double x);
x  Any floating-point number.
```

#### DESCRIPTION

The `j1` function calculates the Bessel function of the first kind of order 1 for argument *x*.

#### EXCEPTIONS

When *x* and *y* are finite and nonzero, j1(*x*) raises the inexact flag. It may also cause the following exceptions:

- overflow (if *x* is finite and the result is infinite)
- underflow (if the result is inexact, must be represented as a denormalized number or 0)

#### SPECIAL CASES

Table 9-41 shows the results when the argument to the j1 function is a zero, a NaN, or an infinity.

**Table 9-38** Special cases for the j1 function

| **Operation** | **Result** | **Exceptions raised** |
|---|---|---|
| j1(-0) | -0 | None |
| j1(+0) | +0 | None |
| j1(NaN) | NaN | None* |
| j1(-∞) | -0 | None |
| j1(+∞) | +0 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = j1(0.0);            /* z = 0.0
z = j1(1.0);            /* z = 4.400505857449335e-01
```

## jn

You can use the `jn` function to calculate the Bessel function of the first kind of integer order n for argument *x*.

```
double jn (double x);
x  Any floating-point number.

n  Any integer.
```

DESCRIPTION

The `jn` function calculates the Bessel function of the first kind of integer order n for argument *x*.

**EXAMPLES**

```
z = yn(2, 0.0);         /* z = 0.0
z = yn(2, 1.0);         /* z = 1.149034849319005e-01
```

## y0

You can use the `y0` function to calculate the linearly independent Bessel function of the second kind of order 0 for argument *x*.

```
double y0 (double x);
x  Any floating-point number.
```

**DESCRIPTION**

The `y0` function calculates the linearly independent Bessel function of the second kind of order 0 for argument *x*.

**EXCEPTIONS**

When *x* and *y* are finite and nonzero, y0(*x*) raises the inexact flag. It may also cause the following exceptions:

- overflow (if *x* is finite and the result is infinite)
- underflow (if the result is inexact, must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-41 shows the results when the argument to the y0 function is a zero, a NaN, or an infinity.

**Table 9-39** Special cases for the j0 function

| **Operation** | **Result** | **Exceptions raised** |
|---|---|---|
| y0(-0) | -∞ | None |
| y0(+0) | -∞ | None |
| y0(NaN) | NaN | None* |
| y0(-∞) | NaN | Invalid |
| y0(+∞) | +0 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = y0(0.0);            /* z = -∞
z = y0(1.0);            /* z = 8.825696421567697e-02
```

## y1

You can use the `y1` function to calculate the linearly independent Bessel function of the second kind of order 1 for argument *x*.

```
double y1 (double x);
x  Any floating-point number.
```

**DESCRIPTION**

The `y1` function calculates the linearly independent Bessel function of the second kind of order 1 for argument *x*.

#### EXCEPTIONS

When *x* and *y* are finite and nonzero, y1(*x*) raises the inexact flag. It may also cause the following exceptions:

- overflow (if *x* is finite and the result is infinite)
- underflow (if the result is inexact, must be represented as a denormalized number or 0)

#### SPECIAL CASES

Table 9-41 shows the results when the argument to the y1 function is a zero, a NaN, or an infinity.

**Table 9-40** Special cases for the j0 function

| **Operation** | **Result** | **Exceptions raised** |
|---|---|---|
| y1(-0) | -∞ | None |
| y1(+0) | -∞ | None |
| y1(NaN) | NaN | None* |
| y1(-∞) | NaN | Invalid |
| y1(+∞) | +0 | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

#### EXAMPLES

```
z = y1(0.0);            /* z = -∞
z = y1(1.0);            /* -7.812128213002887e-01
```

## yn

You can use the yn function to calculate the Bessel function of the second kind of integer order n for argument *x*.

```
double yn (double x);
x  Any floating-point number.

n  Any integer.

DESCRIPTION
```

The `jn` function calculates the Bessel function of the second kind of integer order n for argument *x*.

**EXAMPLES**

```
z = yn(2, 0.0);         /* z = -∞
z = yn(2, 1.0);         /* z = –1.650682606816254e+00
```

## Miscellaneous Functions

There are four remaining Libm transcendental functions:

| | |
|---|---|
| nextafter(*x*, *y*) | Returns next representable number after *x* in direction of *y*. |
| hypot(*x*) | Computes hypotenuse of a right triangle. |
| cbrt(*x*) | Computes cube root of *x*. |
| fma(*x*,*y*,*z*) | Computes (*x***y*)+*z* as a single operation. |

### nextafter

You can use the **nextafter functions** to find out the next value that can be represented after a given value in a particular floating-point type.

```
float       nextafterf (float x, float y);
double      nextafterd (double x, double y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The nextafter functions (one for each data type) generate the next representable neighbor of `x` in the direction of `y` in the proper format.

The floating-point values representable in single and double formats constitute a finite set of real numbers. The nextafter functions illustrate this fact by returning the next representable value.

If $x = y$, nextafter(*x*, *y*) returns *x* if *x* and *y* are not signed zeros.

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero, either the result of nextafter($x$, $y$) is exact or it raises one of the following exceptions:

- inexact (if an overflow or underflow exception occurs)
- overflow (if $x$ is finite and the result is infinite)
- underflow (if the result is inexact, must be represented as a denormalized number or 0, and $x \neq y$)

**SPECIAL CASES**

Table 9-41 shows the results when one of the arguments to a nextafter function is a zero, a NaN, or an Infinity. In this table, $x$ and $y$ are finite, nonzero floating-point numbers.

**Table 9-41** Special cases for the nextafter functions

| Operation | Result | Exceptions raised |
|---|---|---|
| nextafter(+0, $y$) | Next representable number in direction of $y$ | Underflow |
| nextafter($x$, +0) | Next representable number in direction of 0 | None |
| nextafter(–0, $y$) | Next representable number in direction of $y$ | Underflow |
| nextafter(–0, +0) | +0 | None |
| nextafter($x$, –0) | Next representable number in direction of 0 | None |
| nextafter(+0, –0) | –0 | None |
| nextafter(NaN, $y$) | NaN* | None† |
| nextafter($x$, NaN) | NaN | None† |
| nextafter(+∞, $y$) | Largest respresentable number | None |
| nextafter($x$, +∞) | Next representable number greater than $x$ | None |
| nextafter(–∞, $y$) | Smallest representable number | None |
| nextafter($x$, –∞) | Next representable number smaller than $x$ | None |

* If both arguments are NaNs, the value of the first NaN is returned.
† If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = nextafterf(1.0, +∞);/* z = 1.00000000000000000000001₂
                             ≈ 1.000000119209289551 */
z = nextafterd(1.0, +∞);/* z = 1.00000000...000000000000000001₂
                             ≈ 1.000000000000000222 */
```

## hypot

You can use the `hypot` function to compute the length of the hypotenuse of a right triangle.

```
double_t hypot(double_t x, double_t y);
```

| | |
|---|---|
| `x` | Any floating-point number. |
| `y` | Any floating-point number. |

**DESCRIPTION**

The `hypot` function computes the square root of the sum of the squares of its arguments. This is an ANSI standard C library function.

$$\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$$

The function `hypot` performs its computation without undeserved overflow or underflow. For example, if $x^2 + y^2$ is greater than the maximum representable value of the data type but the square root of $x^2 + y^2$ is not, then no overflow occurs.

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero, either the result of $\mathrm{hypot}(x, y)$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-42 shows the results when one of the arguments to the `hypot` function is a zero, a NaN, or an Infinity. In this table, *x* and *y* are finite, nonzero floating-point numbers.

**Table 9-42** Special cases for the `hypot` function

| Operation | Result | Exceptions raised |
|---|---|---|
| hypot(+0, *y*) | \|*y*\| | None |
| hypot(*x*, +0) | \|*x*\| | None |
| hypot(–0, *y*) | \|*y*\| | None |
| hypot(*x*, –0) | \|*x*\| | None |
| hypot(NaN, *y*) | NaN | None* |
| hypot(*x*, NaN) | NaN | None* |
| hypot(NaN, ±∞) | ∞ | None |
| hypot(±∞, NaN) | ∞ | None |
| hypot(+∞, *y*) | +∞ | None |
| hypot(*x*, +∞) | +∞ | None |
| hypot(–∞, *y*) | +∞ | None |
| hypot(*x*, –∞) | +∞ | None |

* If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = hypot(2.0, 2.0); /* z = sqrt(8.0) ≈ 2.82843. The inexact
                        exception is raised. */
```

## cbrt

You can use the `cbrt` function to compute the real cube root of a number.

```
double_t cbrt(double_t x);
```

| | |
|---|---|
| x | Any floating-point number. |

**DESCRIPTION**

The `cbrt` function computes the real cube root of its argument. This is an ANSI standard C library function.

$$\mathrm{cbrt}(x) = \sqrt[3]{x}$$

**EXCEPTIONS**

When $x$ and $y$ are finite and nonzero the result of cbrt($x$) is inexact. It can also raise the following exception:

- underflow (if the result is inexact and must be represented as a denormalized number or 0)

**SPECIAL CASES**

Table 9-42 shows the results when the argument to the `cbrt` function is a zero, a NaN, or an Infinity.

**Table 9-43** Special cases for the `hypot` function

| Operation | Resul t | Exceptions raised |
|---|---|---|
| cbrt(+0) | +0 | None |
| cbrt(–0) | -0 | None |
| cbrt(NaN) | NaN | None* |
| cbrt(+∞) | +∞ | None |
| cbrt(-∞) | -∞ | None† |

* If the NaN is a signaling NaN, the invalid exception is raised.
† If the NaN is a signaling NaN, the invalid exception is raised.

**EXAMPLES**

```
z = cbrt(16.0);      /* z ≈ 2.519842099789746. The inexact
                        exception is raised. */
```

## fma

You can use the `fma` function to compute (x*y)+z, rounded as one ternary operation: it computes the value (as if) to infinite precision and round once to the result format, according to the current rounding mode.

```
double_t fma(double_t x, double_t y, double_t z);
```

| | |
|---|---|
| x | Any floating-point number. |
| y | Any floating-point number. |
| z | Any floating-point number. |

**DESCRIPTION**

The `fma` function computes (x*y)+z as a single ternary function, rounding it once according to the current rounding mode. This is an ANSI standard C library function.

fma($x$,$y$,$z$) = ($x$*$y$)+$z$

**EXCEPTIONS**

`fma` has the following exceptional cases:

- if x, y, or z is a NaN, the result is a NaN
- if one of x or y is a zero and the other is an infinity, the result is a NaN and the "invalid" flag is raised
- if x*y is an infinity and z is the infinity with the opposite sign, the result is a NaN and the "invalid" flag is raised.

**SPECIAL CASES**

See "Exceptions." The table is omitted in this case.

**EXAMPLES**

```
w = fma(2.0,3.0,5.0);/* z = 11.0. */
```

## Vector and Matrix Operations

Mac OS X provides a number of libraries for performing vector and matrix operations and solving systems of linear equations. These libraries are optimized to make effective, high-performance use of the PowerPC vector instruction set, which benefits from a high degree of data parallelism. These libraries include BLAS and LAPACK.

### BLAS

The Basic Linear Algebra Subroutines (BLAS) is a set of high-quality routines for performing basic vector and matrix operations. Level 1 BLAS consists of vector-vector operations, Level 2 BLAS consists of matrix-vector operations, and Level 3 BLAS consists of matrix-matrix operations. The efficiency, portability, and wide adoption of the BLAS have made them commonplace in the development of high-quality linear algebra software such as LAPACK, and in other technologies requiring fast vector and matrix calculations.

Through its vecLib framework, Mac OS X supports all the standard C BLAS entry points (known as the legacy C BLAS) and the industry-standard FORTRAN BLAS entry points. For more informations see http://www.netlib.org/blas/faq.html.

### LAPACK

LAPACK provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and similar value problems. Routines are provided to perform the associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, and generalized Schur) and related computations such as the reordering of the Schur factorizations and estimating condition numbers. LAPACK handles dense and banded matrices, but not general sparse matrices. Functionality is provided for both real and complex matrices, in both single and double precision. The Mac OS X LAPACK makes full use of the optimized BLAS for improved performance. Through its vecLib framework, Mac OS X supports all the industry-standard FORTRAN LAPACK entry points. C programs may make calls to the FORTRAN entry points using the prototypes found in /System/Library/Frameworks/vecLib.framework/Headers/clapack.h

For more information on LAPACK, see http://www.netlib.org/lapack/.

## Transcendental Functions Summary

This section summarizes the transcendental functions declared in the MathLib header file `fp.h` and the constants and data types that they use.

## C Summary

### Constants

```
extern const double_t pi;
```

### Data Types

```
typedef short relop;
```

```
enum
{
   GREATERTHAN = ((relop) (0)),
   LESSTHAN,
   EQUALTO,
   UNORDERED
};
```

## Transcendental Functions

### Comparison Functions

```
double_t fdim             (double_t x, double_t y);
double_t fmax             (double_t x, double_t y);
double_t fmin             (double_t x, double_t y);
relop relation            (double_t x, double_t y);
```

### Sign Manipulation Functions

```
double_t copysign         (double_t x, double_t y);
double_t fabs             (double_t x);
```

### Exponential Functions

```
double_t exp              (double_t x);
double_t exp2             (double_t x);
double_t expm1            (double_t x);
double_t ldexp            (double_t x, int n);
double_t pow              (double_t x, double_t y);
double_t scalb            (double_t x, long int n);
```

### Logarithmic Functions

```
double_t frexp            (double_t x, int *exponent);
double_t log              (double_t x);
double_t log10            (double_t x);
double_t log1p            (double_t x);
double_t log2             (double_t x);
double_t logb             (double_t x);
float modff               (float x, float *iptrf);
double modf               (double x, double *iptr);
```

## Trigonometric Functions

```
double_t cos                (double_t x);
double_t sin                (double_t x);
double_t tan                (double_t x);
double_t acos               (double_t x);
double_t asin               (double_t x);
double_t atan               (double_t x);
double_t atan2              (double_t y, double_t x);
```

## Hyperbolic Functions

```
double_t cosh               (double_t x);
double_t sinh               (double_t x);
double_t tanh               (double_t x);
double_t acosh              (double_t x);
double_t asinh              (double_t x);
double_t atanh              (double_t x);
```

## Error and Gamma Functions

```
double_t erf                (double_t x);
double_t erfc               (double_t x);
double_t gamma              (double_t x);
double_t lgamma             (double_t x);
```

## Nextafter Functions

```
float nextafter             (double_t x, double_t y);
```

## Miscellaneous Functions

```
double_t hypot              (double_t x, double_t y);
double_t cbrt               (double_t x);
double_t fma                (double_t x, double_t y, double_t z);
```

# Libm Reference

This appendix provides a reference for the numeric implementation in the C programming language. It summarizes the data formats available and tells how to determine the floating-point class for a value. It also lists functions that control the floating-point environment, functions that perform floating-point operations, and the exceptions those functions might raise.

## Floating-Point Data Formats

**Figure 0-1** Floating-point data formats

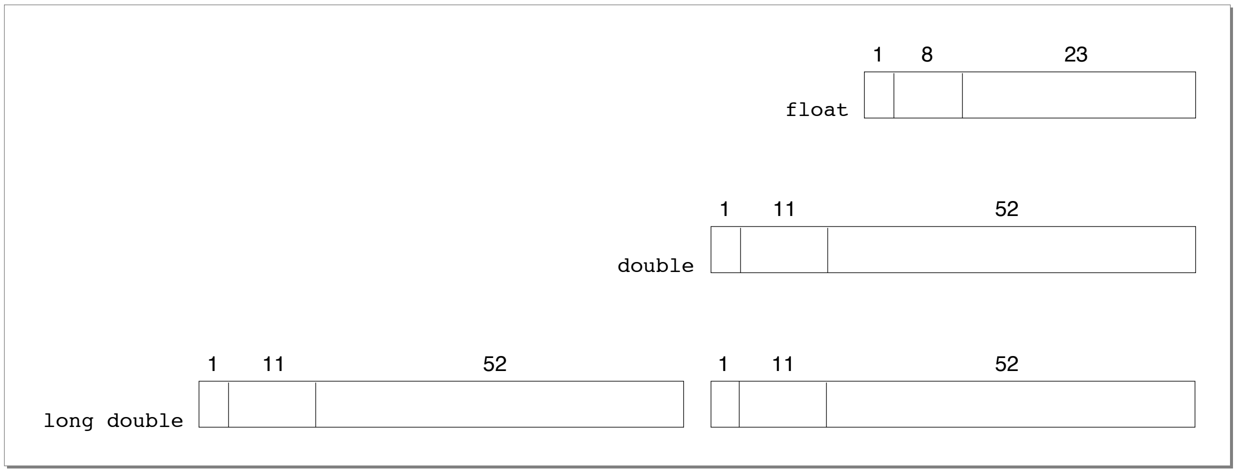


**Table 0-1** Interpreting floating-point values

| If biased[*] exponent *e* is: | And fraction *f* is: | Then value *v* is: | And class of *v* is: [†] |
|---|---|---|---|
| $0 < e < max$ [‡] | (any) | $v = (-1)^s \times 2^{(e - bias)} \times (1.f)$ [§] | `FP_NORMAL` |
| $e = 0$ | $f \neq 0$ | $v = (-1)^s \times 2^{minexp} \times (0.f)$ [¶] | `FP_SUBNORMAL` |
| $e = 0$ | $f = 0$ | $v = (-1)^s \times 0$ | `FP_ZERO` |
| $e = max$ | $f = 0$ | $v = (-1)^s \times \infty$ | `FP_INFINITE` |
| $e = max$ | $f \neq 0$ | $v = \text{NaN}$ | `FP_SNAN` (first bit is 0) `FP_QNAN` (first bit is 1) |

[*] *bias* = 127 for `float`; 1023 for `double` and `long double.`
[†] From enumerated type `NumKind`.
[‡] *max* = 255 for `float`; 2047 for `double` and `long double.`
[§] For `long double` both head and tail are evaluated this way and added together.
[¶] *minexp* = –126 for `float`; –1022 for `double` and `long double.`

**Table 0-2** Class and sign inquiry macros

```
fpclassify(x)
isnormal(x)
isfinite(x)
isnan(x)
signbit(x)
```

# Environmental Controls

Table 0-3 Environmental access

| Action | Function prototype |
|---|---|
| Get | `void fegetenv (fenv_t *envp);` |
| Set | `void fesetenv (const fenv_t *envp);` |
| Save | `int feholdexcept (fenv_t * envp);` |
| Restore | `void feupdateenv (const fenv_t *envp);` |

Table 0-4 Floating-point exceptions

| Action | Function prototype |
|---|---|
| Get | `void fegetexceptflag(fexcept_t *flagp, int excepts);` |
| Raise | `void feraiseexcept (int excepts);` |
| Clear | `void feclearexcept (int excepts);` |
| Set | `void fesetexceptflag (const fexcept_t *flagp, int excepts);` |
| Test | `int fetestexcept (int excepts);` |

Table 0-5 Rounding direction modes

| Action | Function prototype |
|---|---|
| Get | `int fegetround (void);` |
| Set | `int fesetround (int round);` |

**Table 0-6** Floating-point exceptions constants

| Exceptions | Value |
|---|---|
| `FE_INEXACT` | `0x02000000` |
| `FE_DIVBYZERO` | `0x04000000` |
| `FE_UNDERFLOW` | `0x08000000` |
| `FE_OVERFLOW` | `0x10000000` |
| `FE_INVALID` | `0x20000000` |
| `FE_ALL_EXCEPT` | `0x3E000000` |

**Table 0-7** Rounding direction constants

| Modes | Value |
|---|---|
| `FE_TONEAREST` | `0x00000000` |
| `FE_TOWARDZERO` | `0x00000001` |
| `FE_UPWARD` | `0x00000002` |
| `FE_DOWNWARD` | `0x00000003` |

## Operations and Functions

**Note**
Throughout the tables that follow, in the Exceptions column, I = invalid; X = inexact; O = overflow; U = underflow; D = divide-by-zero. ◆

**Table 0-8** Arithmetic operations

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Sum | `x + y` | $-\infty$ to $+\infty$ | `I X O U -` |
| Difference | `x — y` | $-\infty$ to $+\infty$ | `I X O U -` |
| Product | `x * y` | $-\infty$ to $+\infty$ | `I X O U -` |

**Table 0-8** Arithmetic operations

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Quotient | `x / y` | $-\infty$ to $+\infty$ | `I X O U D` |
| Square root | `sqrt(x)` | 0 to $+\infty$ | `I X - - -` |
| Remainder | `remainder(x,y)` `remquo(x,y,quo)` `fmod(x,y)` | $-\infty$ to $+\infty$ | `I - - - -` |

**Table 0-9** Conversions to integer type

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Round in current direction | `lrint(x)`* | $-2^{31}$ to $2^{31}-1$ | `I X - - -` |
| Add 1/2 to magnitude and chop | `lround(x)`* | $-2^{31}$ to $2^{31}-1$ | `I X - - -` |

* Return type of `long int`.

**Table 0-10** Conversions to integer in floating-point type

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Round in current direction | `rint(x)` | $-\infty$ to $+\infty$ | `- X - - -` |
| | `nearbyint(x)` | $-\infty$ to $+\infty$ | `- - - - -` |
| Round upward | `ceil(x)` | $-\infty$ to $+\infty$ | `- - - - -` |
| Round downward | `floor(x)` | $-\infty$ to $+\infty$ | `- - - - -` |
| Add 1/2 to magnitude and chop | `round(x)` | $-\infty$ to $+\infty$ | `- X - - -` |
| Round toward zero | `trunc(x)` | $-\infty$ to $+\infty$ | `- - - - -` |

**Table 0-11** Comparison operations

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Positive difference or 0 | `fdim(x,y)` | $-\infty$ to $+\infty$ | `- X O U -` |
| Maximum of 2 numbers | `fmax(x,y)` | $-\infty$ to $+\infty$ | `- - - - -` |
| Minimum of 2 numbers | `fmin(x,y)` | $-\infty$ to $+\infty$ | `- - - - -` |

Table 0-12 Sign manipulation functions

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Copy the sign | `copysign(x,y)` | $-\infty$ to $+\infty$ | - - - - - |
| $\lvert x \rvert$ | `fabs(x)` | $-\infty$ to $+\infty$ | - - - - - |

Table 0-13 Exponential functions

| Comput e | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| $e^x$ | `exp(x)` | $-\infty$ to $+\infty$ | - X O U - |
| $2^x$ | `exp2(x)` | $-\infty$ to $+\infty$ | - X O U - |
| $e^x - 1$ | `expm1(x)` | $-\infty$ to $+\infty$ | - X O U - |
| $x \times 2^n$ | `ldexp(x,n)` | $-\infty$ to $+\infty$ | - X O U - |
| | `scalb(x,n)` | | - X O U - |
| $x^y$ | `pow(x,y)` | $-\infty$ to $+\infty$ | I X O U D |

Table 0-14 Logarithmic functions

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Fraction and exponent fields of floating-point number | `frexp(x,&n)` | $-\infty$ to $+\infty$ | - - - - - |
| $\ln x$ | `log(x)` | 0 to $+\infty$ | I X - - D |
| $\log_{10} x$ | `log10(x)` | 0 to $+\infty$ | I X - - D |
| $\ln (x + 1)$ | `log1p(x)` | $> -1$ | I X - - D |
| $\log_2 x$ | `log2(x)` | 0 to $+\infty$ | I X - - D |
| Exponent field of floating-point number | `logb(x)` | $-\infty$ to $+\infty$ | - - - - D |
| Split real number into fractional part and integer part | `modf(x,&y)` | $-\infty$ to $+\infty$ | - - - - - |

**Table 0-15** Trigonometric functions

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| cos $x$ | `cos(x)` | Any finite number | `I X - - -` |
| sin $x$ | `sin(x)` | Any finite number | `I X - U -` |
| tan $x$ | `tan(x)` | Any finite number | `I X - U -` |
| arccos $x$ | `acos(x)` | –1 to +1 | `I X - - -` |
| arcsin $x$ | `asin(x)` | –1 to +1 | `I X - U -` |
| arctan $x$ | `atan(x)` | $-\infty$ to $+\infty$ | `- X - U -` |
| arctan $y/x$ | `atan2(x,y)` | $-\infty$ to $+\infty$ | `- X - U -` |

**Table 0-16** Hyperbolic functions

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| cosh $x$ | `cosh(x)` | $-\infty$ to $+\infty$ | `- X O - -` |
| sinh $x$ | `sinh(x)` | $-\infty$ to $+\infty$ | `- X O U -` |
| tanh $x$ | `tanh(x)` | $-\infty$ to $+\infty$ | `- X - - -` |
| arccosh $x$ | `acosh(x)` | 1 to $+\infty$ | `I X - - -` |
| arcsinh $x$ | `asinh(x)` | $-\infty$ to $+\infty$ | `- X - U -` |
| arctanh $x$ | `atanh(x)` | –1 to +1 | `I X - U -` |

**Table 0-17** Error and gamma functions

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| error | `erf(x)` | $-\infty$ to $+\infty$ | `- X - U -` |
| 1 – error | `erfc(x)` | $-\infty$ to $+\infty$ | `- X - U -` |
| $\Gamma(x)$ | `gamma(x)` | 0 to $+\infty$ | `I X O - -` |
| $\ln(\lvert\Gamma(x)\rvert)$ | `lgamma(x)` | 0 to $+\infty$ | `I X O - -` |

**Table 0-18** Miscellaneous functions

| Compute | Syntax | Valid input range | Exceptions |
|---|---|---|---|
| Create NaN | `nan(tagp)` | character string | `- - - - -` |
| Next representable number after *x* in direction of *y* | `nextafter(x,y)` | $-\infty$ to $+\infty$ | `- X O U -` |
| Hypotenuse | `hypot(x,y)` | $-\infty$ to $+\infty$ | `- X O U -` |
| Cube root of *x* | `cbrt(x)` | $-\infty$ to $+\infty$ | `- X O U -` |
| Multiply and add ($x*y+z$) | `fma(x,y,z)` | $-\infty$ to $+\infty$ | `I X O U -` |

# Glossary

**ANSI X3J11.1** A branch of the American National Standards Institute (ANSI) that is working on a numerics standard for the C programming language. This group is also called the *Numerical C Extensions Group* (NCEG) and has produced the Floating-Point C Extensions (FPCE) technical report.

**antisymmetric** Used to describe a function whose graph is not symmetrical across the y-axis; that is $func(x) \neq func(-x)$ for all $x$.

**atomic operations** Operations that pass extra information back to their callers by signaling exceptions but that hide internal exceptions, which might be irrelevant or misleading.

**bias** A number added to the binary exponent of a floating-point number so that the exponent field will always be positive. The bias is subtracted when the floating-point value is evaluated.

**binade** The collection of numbers that lie between two successive powers of 2.

**binary floating-point number** A collection of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and 2 raised to the power of the exponent.

**complex expression** An expression made up of more than one simple expression, that is, an expression with more than one floating-point operation.

**Condition Register** A 32-bit PowerPC register used to summarize the states of the fixed-point and floating-point processors and to store results of comparison operations.

**decimal format structure** A data type for specifying the formatting for decimal (base 10) numbers (of conversions). It specifies the decimal number's style and number of digits. It is defined by the `decform` data type.

**default environment** The environment settings when a PowerPC Numerics implementation starts up: rounding is to nearest and all exception flags are clear.

**denormalized number** A nonzero binary floating-point number whose significand has an implicit leading bit of 0 and whose exponent is the minimum exponent for the number's data format. Also called *denorm*. See also **normalized number.**

**divide-by-zero exception** A floating-point exception that occurs when a finite, nonzero number is divided by zero or some other improper operation on zero has occurred.

**double format** A 64-bit application data format for storing floating-point values of up to 15- or 16-decimal digit precision.

**double-double format** A 128-bit application data format made up of two double-format numbers. It has the same range as the double format but much greater precision.

**environmental access switch** A switch, recommended in the FPCE technical report, that specifies whether a program accesses the rounding direction modes and exception flags.

**environmental controls** The rounding direction modes and the exception flags.

**evaluation format** The data format used to evaluate the result of an expression. The evaluation format must be at least as wide as the expression's semantic type. (It may be the same as the semantic type.)

**exception** An error or other special condition detected by the microprocessor in the course of program execution. The floating-point exceptions are invalid, underflow, overflow, divide-by-zero, and inexact.

**exception flag** Each exception has a flag that can be set, cleared, and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

**exponent** The part of a binary floating-point number that indicates the power to which 2 is raised in determining the value of the number. The wider the exponent field in a numeric data format, the greater range the format will handle.

**expression evaluation method** The method by which an evaluation format is determined for an expression.

**floating-point operation** An operation that is performed on numbers in floating-point formats. The IEEE standard requires that a numerics environment support addition, subtraction, multiplication, division, square root, remainder, and round-to-integer as the basic floating-point arithmetic operations.

**Floating-Point Status and Control Register (FPSCR)** A 32-bit PowerPC register used to store the floating-point environment.

**flush-to-zero system** A system that excludes denormalized numbers. Results smaller than the smallest normalized number are rounded to zero.

**FPSCR** See **Floating-Point Status and Control Register.**

**fraction** A field in a floating-point data format that stores all but the leading bit of the significand of a floating-point number.

**gradual underflow** A process that occurs on a computer system that includes denormalized numbers.

**IEEE standard** A term used in this book to mean IEEE Standard 754.

**IEEE Standard 754** A standard that defines how computers should perform binary floating-point arithmetic.

**IEEE Standard 854** A standard that defines how computers should perform radix- independent floating-point arithmetic.

**inexact exception** A floating-point exception that occurs when the exact result of a floating-point operation must be rounded.

**Infinity** A special value produced when a floating-point operation should produce a mathematical infinity or when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

**integer types** System types for integral values. Integer types typically use 16- or 32-bit two's-complement integers. Integer types are not PowerPC Numerics formats but are available to PowerPC Numerics users.

**integral value** A value, perhaps in a numeric data format, that is exactly equal to a mathematical integer. For example, –2, –1, 0, 1, 2, and so on.

**invalid exception** A floating-point exception that occurs if an operand is invalid for the operation being performed.

**invalid-operation exception** See **invalid exception.**

**Machine State Register** A 32-bit PowerPC supervisor-level register that records the state of the processor, including if floating-point instructions and floating-point exceptions are enabled.

**mantissa** See **significand.**

**minimum evaluation format** The narrowest format in which a floating-point operation can be performed. Each implementation of PowerPC Numerics defines its own minimum evaluation format.

**multiply-add instruction** A type of instruction unique to the PowerPC architecture. Multiply-add instructions perform a multiply plus an addition or subtraction operation with at most a single roundoff error.

**NaN (Not-a-Number)** A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs propagate through arithmetic operations.

**NCEG (Numerical C Extensions Group)** See **ANSI X3J11.1.**

**nextafter functions** Functions that return the next value after the input value that is representable in one of the floating-point data formats. For example, nextafterd(0, +∞) returns the value that comes immediately after 0 in the direction of +∞ in double format.

**normalized number** A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1. Compare **denormalized number.**

**Numerical C Extensions Group (NCEG)** See **ANSI X3J11.1.**

**overflow exception** A floating-point exception that occurs when the magnitude of a floating-point result is greater than the largest finite number that the destination data format can represent.

**PowerPC processor** Any member of the family of PowerPC microprocessors. The MPC601 processor is the first PowerPC central processing unit.

**PowerPC processor-based Macintosh computer** Any computer containing a PowerPC central processing unit that runs Macintosh system software. Compare **680x0-based Macintosh computer.**

**precision** The number of digits required to accurately represent a number. For example, the value 3.2 requires two decimal digits of precision, and the value 3.002 requires four decimal digits. In numeric data formats, the precision is equal to the number of bits (both implicit and explicit) in the significand.

**quiet NaN** A NaN that propagates through arithmetic operations without signaling an exception.

**rounding** An action performed when a result of an arithmetic operation cannot be represented exactly in a numeric data format. With rounding, the computer changes the result to a close value that can be represented exactly.

**rounding direction modes** Modes that specify the direction a computer will round when the result of an arithmetic operation cannot be represented exactly in a numeric data format. Under PowerPC Numerics, the computer resolves rounding decisions in one of the four directions chosen by the user: to nearest (the default), upward, downward, and toward zero.

**roundoff error** The difference between the exact result of an IEEE arithmetic operation and the result as it is represented in the numeric data format if the result has been rounded.

**semantic type** The widest type of the operands of an expression.

**signaling NaN** A NaN that signals an invalid exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No PowerPC Numerics operation creates signaling NaNs.

**sign bit** The bit of a single, double, or double-double number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

**significand** The part of a binary floating-point number that indicates where the number falls between two successive powers of 2. The wider the significand field in a numeric format, the more precision the format has.

**simple expression** An expression containing one floating-point operation.

**single format** A 32-bit application data format for storing floating-point values that have a precision of up to seven or eight decimal digits. It is used by engineering applications, among others.

**sticky** Used to describe a condition in which a bit stays set until it is explicitly cleared. Floating-point exception flags in the FPSCR are sticky, so if one instruction sets an exception flag and another instruction is performed before the flag is tested, it is impossible to tell which instruction caused the exception.

**subnormal number** A denormalized number.

**symmetric** Used to describe a function whose graph looks the same on both sides of the y-axis; that is, $func(x) = func(-x)$ for all $x$.

**tiny** Used to describe a number whose magnitude is smaller than the smallest positive normalized number in the format of the number.

**transcendental functions** Functions that can be used as building blocks in numerical functions. All of the functions contained in the PowerPC Numerics library are transcendental functions.

**trigonometric functions** Functions that perform trigonometric operations, such as cosine, sine, and tangent.

**truncate** To chop off the fractional part of a real number so that only the integer part remains. For example, if the real number 1.99999999999 is truncated, the truncated value is 1.

**underflow exception** An exception that occurs when the result of an operation is both tiny and inexact.

**usual arithmetic conversions** Automatic conversions performed in the C programming language. The ANSI C specification defines these conversions.

**widest-need evaluation** An evaluation method in which the widest format of all of the operands in a complex expression is used as the format in which the expression is evaluated.

# Bibliography

Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

Alefeld, G., and J. Hertzberger. *Introduction to Interval Computations.* New York: Academic Press, 1983.

American National Standards Institute. *Floating-Point C Extensions*, prepared by the Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group. ANSI X3J11.1/93-028, 1993.

Apple Computer. *Apple Numerics Manual*, second edition. Reading, MA: Addison-Wesley, 1988.

Apple Computer. *Inside Macintosh: PowerPC System Software*. Reading, MA: Addison-Wesley, 1994.

Apple Computer. *Assembler for Macintosh With PowerPC*. Cupertino, CA: Apple Computer, 1994.

Apple Computer. *C/C++ Compiler for Macintosh With PowerPC*. Cupertino, CA: Apple Computer, 1994.

Brown, W. S. "A Simple but Realistic Model of Floating-Point Computation." *ACM Transactions on Mathematical Software* Vol. 7, No. 4 (1981).

Cody, W. J. "Floating-Point Standards—Theory and Practice." In *Reliability in Computing: The Role of Interval Methods on Scientific Computing*, edited by Ramon E. Moore. Boston, MA: Academic Press, 1988.

Cody, W. J., et al. "A Proposed Radix- and Word-Length-Independent Standard for Floating-Point Arithmetic." *IEEE Micro* Vol. 4, No. 4 (1984).

Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *IEEE Computer* Vol. 13, No. 1 (1980).

Coonen, Jerome T. "Underflow and the Denormalized Numbers." *IEEE Computer* Vol. 14, No. 3 (1981).

Coonen, Jerome T. "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic." Ph.D. Thesis, University of California at Berkeley, 1984. (Available from University Microfilm, Ann Arbor, MI.)

Dekker, T. J. "A Floating-Point Technique for Extending the Available Precision." *Numerisch Mathematik* Vol. 18, No. 3 (1971).

Demmel, James. "The Effects of Underflow on Numerical Computation." *SIAM Journal on Scientific and Statistical Computing* Vol. 5, No. 4 (1984).

Farnum, Charles. "Compiler Support for Floating-Point Computation." *Software Practices and Experience* Vol. 18, No. 7 (1988).

Fateman, Richard J. “High-Level Language Implications of the Proposed IEEE Floating-Point Standard.” *ACM Transactions on Programming Languages and Systems* Vol. 4, No. 2 (1982).

Floating-Point C Extensions (FPCE) technical report. *See* American National Standards Institute.

Forsythe, G. E., and C. B. Moler. *Computer Solution of Linear Algebraic Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1967.

FPCE technical report. *See* American National Standards Institute.

Goldberg, D. “Computer Arithmetic.” In *Computer Architecture: A Quantitative Approach*, edited by David Patterson and John L. Hennessy. Los Altos, CA: Morgan Kaufmann, 1990.

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. Baltimore, MD: Johns Hopkins University Press, 1989.

Hough, D. “Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic.” *IEEE Computer* Vol. 14, No. 3 (1981).

Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Standard 754-1985. New York: IEEE, 1985.

Institute of Electrical and Electronics Engineers. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*. IEEE Standard 854-1987. New York: IEEE, 1987.

Kahan, W. “Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard.” In *Interval Mathematics* 1980, edited by K. E. L. Nickel. New York: Academic Press, 1980.

Kahan, W. “Rational Arithmetic in Floating-Point.” Berkeley, CA: Report No. PAM-343, Center for Pure and Applied Mathematics, University of California, 1986a.

Kahan, W. “To Solve a Real Cubic Equation.” Berkeley, CA: Report No. PAM-352, Center for Pure and Applied Mathematics, University of California, 1986b.

Kahan, W. “Branch Cuts for Complex Elementary Functions.” In *The State of the Art in Numerical Analysis,* edited by A. Iserles and M. J. D. Powell. New York: Oxford University Press, 1987.

Kahan, W., and Jerome T. Coonen. “The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments.” In *The Relationship between Numerical Computation and Programming Languages,* edited by J. K. Reid. New York: North Holland, 1982.

Kulish, U. W., and W. L. Miranker. “The Arithmetic of the Digital Computers: A New Approach.” *SIAM Review* Vol. 28, No. 1 (1986).

Matula, D. W., and P. Kornerup. “Finite Precision Rational Arithmetic: Slash Number Systems.” *IEEE Transactions on Computing* Vol. C-34, No. 1 (1985).

Moore, R. E. *Methods and Applications of Interval Analysis.* Society for Industrial and Applied Mathematics, 1979.

Motorola Corporation. *PowerPC 601 RISC Microprocessor User's Manual*, Motorola Corporation, 1993.

Rice, John R. *Numerical Methods, Software, and Analysis*, second edition. New York: Academic Press, 1992.

Sterbenz, Pat H. *Floating-Point Computation.* Englewood Cliffs, NJ: Prentice-Hall, 1974.

Swartzlander, E. E., and G. Alexopoulos. "The Sign/Logarithm Number System." *IEEE Transactions on Computing* Vol. C-24, No. 12 (1975).

# Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe™ Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Jean Ostrem

LEAD WRITER
Tim Monroe

DEVELOPMENTAL EDITOR
Jeanne Woodward

ILLUSTRATOR
Shawn Morningstar

ART DIRECTOR
Betty Gee

PRODUCTION EDITOR
Lorraine Findlay

PROJECT LEADER
Patricia Eastman

COVER DESIGNER
Barbara Smyth

TECHNICAL CONTRIBUTORS
Ali Sazegari, Paul Finlayson, and Kenton Hanson

Special thanks to Scott Fraser, Liz Ghini, Brian Strull, and Allen Watson III.