




Apple. Apple Numerics Manual

Second Edition



 **Addison-Wesley Publishing Company, Inc.**
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo San Juan

RECTO



Chapter 1

About IEEE Standard Arithmetic

You can use this book to help you understand what goes on in IEEE Standard floating-point arithmetic. This chapter introduces some of the salient features of IEEE arithmetic and describes some of the ways you can use those features in your programs.

Starting to use IEEE standard arithmetic

You can get the benefit of many of the features described in this chapter without special programming techniques. Other features may require changes to your programs. If you are just starting out with SANE, you can approach the new features in three increments:

- ☐ Recompile your old programs with no changes; you'll get some of the benefits.
- ☐ Make small changes to obtain more benefits.
- ☐ Use the advanced features for special applications.

The rest of this chapter introduces features that are in the first two categories and concludes with a brief introduction to the advanced features. For complete descriptions of the features of SANE, refer to the other chapters in this book.

Well-behaved arithmetic

For most programs, using IEEE Standard arithmetic is quite simple: you just do arithmetic in the normal way. Usually, the results are as good as or better than those from other computer arithmetics.

These features make SANE a well-behaved arithmetic:

- ☐ extended precision and range
- ☐ accurate results
- ☐ careful rounding
- ☐ gradual underflow

Extended precision and range

To make expression evaluation simpler and more accurate, IEEE arithmetic allows operations to be performed and results to be delivered in an extended data format that has significantly greater precision and range than the single-precision and double-precision types. The increased precision provides more accuracy; the greater range avoids undesired overflow and underflow.

High-level languages evaluate expressions in extended format and allow the programmer to declare extended-precision temporary variables. Using the extended format throughout a computation and rounding the final results back to single or double often give improved accuracy.

Accurate results

All the fundamental operations in extended-format IEEE arithmetic produce the best possible results, limited only by the precision and range of the extended data type. Those operations (addition, subtraction, multiplication, division, square root, remainder, rounding to integer, and conversions between floating-point formats) deliver results that are accurate to the last bit. Conversions between binary and decimal are equally accurate, except when values are extremely large or extremely small, and even then they are almost that accurate.

Before IEEE Standard arithmetic, it would have been unwise to assume that your computer's arithmetic was this accurate. Now, with SANE, even for complicated calculations, you usually don't need to worry about accuracy.

Gradual underflow

When a nonzero result is too small for normal representation in the data format, it has to be replaced by something else. Many computers simply replace such tiny results with zero. When they do that, they produce an error that is much worse than the rounding error in slightly larger numbers. Gradual underflow avoids most of this error by using a denormalized value, as described in Chapter 5, "Infinities, NaNs, and Denormalized Numbers."

Because of gradual underflow, the following is always true in IEEE arithmetic:

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

That statement is not true on machines that lack gradual underflow.

Careful rounding

IEEE arithmetic normally rounds results to the nearest value. You can choose to round in other ways—see the section "Control of Rounding" later in this chapter—but generally the default rounding delivers the desired results. The following example is a simple demonstration of the advantages of careful rounding.

Example: Inverse operations

Suppose your program 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

$$y = \log(x), x = \exp(y)$$

$$y = 375x, x = y/375$$

The computed values $F(x)$ and $G(y)$ will sometimes 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 SANE very often returns the correct inverse value.

There are two reasons for SANE's good behavior with respect to inverse operations. One is that SANE normally uses the extended data type for intermediate values. When you store the result in a narrower format, SANE rounds the result to the nearest value, often rounding away the errors. Another reason is that SANE rounds so carefully. Even with all operations in, say, single precision, SANE evaluates the expression $3 \times (1/3)$ to 1.0 exactly; some other computers don't. If you find that surprising, you might enjoy running the following example on a computer that doesn't use IEEE arithmetic and on an Apple® computer using SANE. SANE's default rounding gives good results: the Apple computer prints 'No failures'. The program will fail on a computer that doesn't have IEEE arithmetic—in particular, IEEE arithmetic's treatment of halfway cases of rounding to nearest.

PROGRAM invop;

```

VAR
  x, y, a, b: single;
  ix, iy: integer;
  nofail: boolean;

BEGIN
  nofail := true;
  FOR ix := 1 TO 12 DO
    IF ix <> 7 AND ix <> 11 THEN ( so ix is a sum of two powers of 2 )
      FOR iy := 1 TO 50 DO
        BEGIN
          x := ix;
          y := iy;
          a := y/x;
          b := x*a;
          IF b <> y THEN
            BEGIN
              nofail := false;
              writeln('It failed for x =', ix, ' y =', iy)
            END;
          END;
        IF nofail THEN writeln('No failures');
      END.

```

- ♦ *Note:* This example deliberately avoids the use of extended in order to demonstrate one effect of careful rounding. Declaring the temporary variable *a* to be extended—normally good programming practice with SANE—removes the necessity for restricting *ix* to sums of two powers of 2.

Alternatives to stopping

There are limits to everything; when you exceed them, something exceptional happens. The exceptional events are

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

Many computers either stop on these exceptions or simply ignore them. IEEE Standard arithmetic gives programmers the choice of continuing, stopping, or executing special code.

IEEE arithmetic includes special values NaN (Not-a-Number) and Infinity. When a program encounters an invalid operation, overflow, or division by zero, the arithmetic returns the appropriate NaN or Infinity so that the program can continue. For detailed descriptions of NaN and Infinity, please see Chapter 5, "Infinities, NaNs, and Denormalized Numbers."

IEEE Standard arithmetic allows (and SANE provides) the option to stop computation when these situations arise, but there are good reasons why you might prefer not to have to stop. The following examples illustrate some of them.

Example: compound conditional statements

Suppose a programmer wants to write a simple statement to perform two tests, one of which can cause an invalid operation such as $0/0$. In Pascal, the statement might look like this:

```
if x = 0 or y/x < 3 then writeln ('Eureka!');
```

When *x* and *y* are both equal to 0, the programmer intends this statement to print "Eureka!" With a Pascal compiler that supports SANE, the statement will produce the desired result. To obtain the desired result on all computers, the programmer would have to be careful and write something more cumbersome. By allowing y/x when *x* and *y* are zero, SANE lets the programmer write simpler code.

This program fragment demonstrates the principal service performed by NaNs: permitting deferred judgments about variables whose values may 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 in the hope that whatever caused the NaN will turn out to be irrelevant to the solution.

- ♦ *Pascal note:* Apple's MPW Pascal compilers include a short-circuit option (\$SC+) that causes the program not to evaluate the second part of a compound conditional if the result value is already determined. Code compiled with that option avoids evaluating 0/0 in the fragment above, but not in the following similar one:

```
if y/x < 3 or x = 0 then writeln ('Eureka!');
```

Searching without stopping

Suppose your program has to search through a database for something like 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. For some records, data might be nonexistent or invalid. On many machines, that would cause the program to stop. To avoid having the program stop during the search, you'd have to add tests for all the exceptional cases. With SANE, the subroutine doesn't stop for nonexistent or invalid data; it simply returns a NaN.

This is another example of the way arithmetic that includes NaN 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 at your disposal, you can handle all exceptional cases after they have occurred.

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.

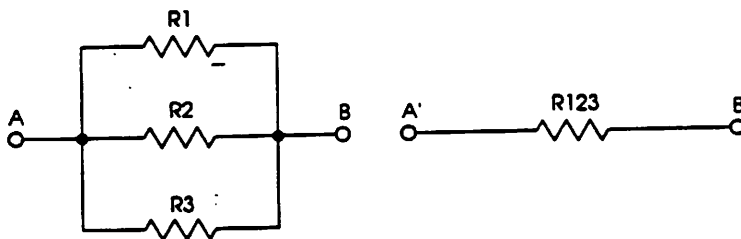


Figure 1-1
Parallel resistances

When three electrical resistances R_1 , R_2 , and R_3 are connected in parallel as shown in Figure 1-1, their effective resistance is the same as a single resistance whose value R_{123} is given by this formula:

$$R_{123} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

The formula gives correct results for positive resistance values between 0 (corresponding to a short circuit) and infinity (corresponding to an open circuit) inclusive. On computers that don't allow division by zero, the programmer 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.

Advanced features

SANE also includes more advanced features, such as control of rounding direction and precision, tools for handling exceptional cases, and a set of elementary (transcendental) functions suitable for use as core routines in mathematical functions. These features are only introduced here; for complete descriptions, please see Chapter 7, "Controlling the SANE Environment," and Chapter 8, "Elementary Functions in SANE."

Control of rounding

Rounding is normally carried out to the nearest value, but IEEE Standard arithmetic gives the programmer complete control of rounding precision and direction (see the section "Rounding Direction" in Chapter 7).

Sometimes you may want to know that roundoff 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 than the exact answer. IEEE arithmetic gives you a means of doing that. Fully developed, this strategy is called *interval arithmetic*. See Kahan [22].

Exception handling

There are three ways for a program to deal with exceptions:

- ☐ continue operation
- ☐ stop on exceptions, if you think they're causing trouble
- ☐ include code to do something special when exceptions happen

The features of IEEE arithmetic enable programs to deal with the exceptions in reasonable ways, as this book explains. There are the special values NaN and Infinity so a program can continue operation: see the sections "Infinities" and "NaNs" in Chapter 5. There are also flags, which a program can test to detect exceptional events, and halts, which transfer control to code for handling special cases: see the section "Exception Flags and Halts" in Chapter 7.

Elementary functions

SANE includes high-precision elementary functions that are consistent with the IEEE Standard and that can be used as building blocks in numerical functions. The elementary functions include the usual logarithmic and exponential functions, plus $\ln(1 + x)$ and $e^x - 1$; financial functions for compound interest and annuity calculations; trigonometric functions; and a random number generator.

RECTO



Chapter 7



Controlling the SANE Environment

Environmental controls include the rounding direction, rounding precision, exception flags, and halt settings.

Rounding direction

RoundDir = (ToNearest, Upward, Downward, TowardZero);

PROCEDURE SetRound(r: RoundDir);

FUNCTION GetRound: RoundDir;

The available rounding directions are

- ☐ to-nearest
- ☐ upward
- ☐ downward
- ☐ toward-zero

The rounding direction affects all conversions except conversions between decimal records and decimal strings and all arithmetic operations except remainder. Except for conversions between binary and decimal (described in Chapter 3, "Conversions in SANE"), all operations are computed as if with infinite precision and range and then rounded to the destination format according to the current rounding direction. The rounding direction may be interrogated and set by the user.

◆ *Note:* Transcendental functions are not arithmetic operations and do not produce the correctly rounded value described here.

The default rounding direction is to-nearest. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is the comp or a system-specific integer type or when the round-to-integral-value operation is used. If the magnitude of the infinitely precise result exceeds the format'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 format's value (possibly INF) closest to and no less than the infinitely precise result. When rounding downward, the result is the format's value (possibly -INF) closest to and no greater than the infinitely precise result. When rounding toward zero, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use toward-zero rounding either with conversion into an integer format or with the round-to-integral-value operation. (See also the sections on expressions in "Pascal SANE Extensions" and "C SANE Extensions" in Appendix A.)

Example: rounding upward

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

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

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 following code fragment changes the rounding direction to compute an upper bound for the expression, then restores the previous rounding.

```
VAR
  r: RoundDir;                (local storage for rounding direction)
  xUp: extended;
  . . .

  r := GetRound;              (save rounding direction)
  SetRound(Downward);         (downward rounding for denominator)
  xUp := f+g;
  SetRound(Upward);           (upward rounding for expression)
  xUp := (a*b+c*d)/xUp;
  SetRound(r)                 (restore previous rounding direction)
  . . .
```

Rounding precision

```
RoundPre = (ExtPrecision, DblPrecision, RealPrecision);
```

```
PROCEDURE SetPrecision(p: RoundPre);
```

```
FUNCTION GetPrecision: RoundPre;
```

Normally, SANE arithmetic computations produce results to extended precision and range. To facilitate simulations of arithmetic systems that are not extended-based, the IEEE Standard requires that the user be able to set the rounding precision to single or to double. If the SANE user sets rounding precision to single (or to double), then all arithmetic operations produce results that are correctly rounded and that overflow or underflow as if the destination were single (or double), even though results are typically delivered to extended formats. Conversions to double and to extended formats are affected if rounding precision is set to single, and conversions to extended formats are affected if rounding precision is set to double; conversions to decimal, comp, and system-specific integer types are not affected by the rounding precision. Rounding precision can be interrogated as well as set.

Setting rounding precision to single or to double does not significantly enhance performance, and in some SANE implementations may hinder performance.

Exception flags and halts

TYPE

```
Exception = integer;
```

CONST

```
Invalid = 1;
Underflow = 2;
Overflow = 4;
DivByZero = 8;
Inexact = 16;
```

```
PROCEDURE SetException(e: Exception; b: boolean);
```

```
FUNCTION TestException(e: Exception): boolean;
```

```
PROCEDURE SetHalt(e: Exception; b: boolean);
```

```
FUNCTION TestHalt(e: Exception): boolean;
```

♦ *Note:* The values of the exception constants and the type definition for `Exception` vary among different implementations of SANE, but code that uses the functions, procedures, and symbolic constant names to access exceptions and halts should port across the different implementations. Please refer to other parts of this manual for implementation-dependent information.

Exceptions are signaled when detected; if the corresponding halt is enabled, the SANE engine transfers control to a user-specified location. (A high-level language may not pass on to its user the facility to set this location, but instead may stop the user's program.) The user's program can examine or set individual exception flags and halts, and can save and get the entire environment (rounding direction, rounding precision, exception flags, and halt settings).

{If halt vector is to be made available to Pascal users:}

```
FUNCTION GetHaltVector: longint;
```

```
PROCEDURE SetHaltVector(v: longint);
```

A control mechanism such as this can also be provided by hardware—for example, the Motorola MC68881 floating-point coprocessor. On machines with hardware exception trapping, programs should use the hardware mechanism instead of the software-supported mechanism described here. For information about the halt (trap) mechanism on the MC68881, please refer to Chapter 30, "The MC68881 Trap Mechanism," and to Motorola's *MC68881 Floating-Point Coprocessor User's Manual*.

Types of exceptions

SANE supports five exception flags with corresponding halt settings:

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

Invalid operation

The invalid-operation exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination format is single, double, extended, or comp. The invalid conditions for the different operations are these:

- ☐ addition or subtraction: magnitude subtraction of Infinities, for example, $(+INF) + (-INF)$
- ☐ multiplication: $0 \times INF$
- ☐ division: $0/0$ or INF/INF
- ☐ remainder: $x \text{ rem } y$, where y is zero or x is infinite
- ☐ square root: if the operand is less than zero
- ☐ conversion: to the comp format or to a system-specific integer format when excessive magnitude, Infinity, or NaN precludes a faithful representation in that format (see Chapter 3, "Conversions in SANE," for details)
- ☐ comparison: with predicates involving less-than or greater-than, but not unordered, when at least one operand is a NaN
- ☐ any operation on a signaling NaN except the following: class and sign inquiries and, on some implementations, sign manipulations (Negate, Absolute Value, and CopySign)
- ♦ *Note:* Compilers for high-level languages may move extended-format numbers either by extended-to-extended conversions, which detect signaling NaNs, or by bit copies, which don't. Thus, some compiler-generated moves cause signaling NaNs to raise the invalid exception earlier than expected.

Underflow

The (unhalted) underflow exception is signaled when a floating-point result is both tiny and inexact (and therefore is perhaps significantly less accurate than it would be if the exponent range were unbounded). A result is considered tiny if its magnitude is smaller than its format's smallest positive normalized number.

- ◆ *Note:* Different SANE engines may test for a tiny result either before or after rounding the result to its destination format. If the underflow halt is set, the halt occurs either when the result is tiny and inexact or when the result is simply tiny; see "Example: Gradual Underflow" in Chapter 5. For details about the 65C816 and 6502 SANE engines, refer to Chapter 15. For details about the MC68000 SANE engine, refer to Chapter 23. For details about the MC68881 SANE engine, refer to Chapter 30 and to Motorola's *MC68881 Floating-Point Coprocessor User's Manual*.

Divide-by-zero

The divide-by-zero exception is signaled when a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result; for example, `Logb(0)` returns `-INF` and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

Overflow

The overflow exception is signaled when a floating-point destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (Invalid, rather than overflow, flags the production of an out-of-range value for an integral destination format.)

Inexact

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

Managing environmental settings

Environmental settings include the rounding direction, rounding precision, exception flags, and halt settings. These settings are global and can be explicitly changed by the program. Thus, all routines inherit these settings and are capable of changing them. Conventionally, routines that change these settings first save them, then restore when finished.

Environment = integer;

```
PROCEDURE SetEnvironment(e: Environment);
PROCEDURE GetEnvironment(var e: Environment);
PROCEDURE ProcEntry(var e: Environment);
PROCEDURE ProcExit(e: Environment);
```

♦ *Note:* The type definition of the Environment word can be different for different SANE implementations, but code that uses the procedures to access the environment should port across the different implementations. On a machine with an MC68881, the SANE environment is stored in the MC68881's control and status registers; see Chapter 29, "Controlling the MC68881 Environment."

Example: setting rounding direction

A subroutine that includes the following statements uses to-nearest rounding while not affecting its caller's rounding direction.

```
VAR
  r: RoundDir;           {local storage for rounding direction}

BEGIN
  r := GetRound;          {save caller's rounding direction}
  SetRound(ToNearest);    {set to-nearest rounding}

  { subroutine's operations here }

  SetRound(r)             {restore caller's rounding direction}
END;
```

Notice that if the subroutine is to be reentrant, then storage for the caller's environment must be local.

SANE implementations may provide two efficient procedures for managing the environment as a whole: the Procedure-Entry and Procedure-Exit procedures.

The Procedure-Entry procedure returns the current environment (for saving in local storage) and sets the default environment: rounding direction to-nearest, rounding precision extended, and exception flags and halts clear.

Example: setting environment

A subroutine that includes the following statements runs under the default environment while not affecting its caller's environment.

```
VAR                                     (local storage for environment)
  e: Environment;

BEGIN
  ProcEntry(e);                        (save caller's environment and set
                                       default environment)
  { subroutine's operations here }
  SetEnvironment(e)                   (restore caller's environment)
END;
```

Example: setting exceptions

The Procedure-Exit procedure 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 may be irrelevant or misleading. The Procedure-Exit procedure, which takes a saved environment as argument, does the following:

1. It temporarily saves the exception flags (raised by the subroutine).
2. It restores the environment received as argument.
3. It signals the temporarily saved exceptions. (Note that if enabled, halts could occur at this step.)

Thus, exceptions signaled between the Procedure-Entry and Procedure-Exit procedures are hidden from the calling program unless the exceptions remain raised when the Procedure-Exit procedure is called.

KCTC:

The following function signals underflow if its result is denormalized, overflow if its result is Infinity, and inexact always, but hides spurious exceptions occurring from internal computations:

FUNCTION NumFcn: extended;

```

VAR
  e:      Environment;      (local storage for environment)
  c:      NumClass;         (for class inquiry)

BEGIN
  ProcEntry(e);             (NumFcn)
                             (save caller's environment and set
                             default environment - now halts are
                             disabled)
  . . .                     (internal computation)

  NumFcn := Result;          (result to be returned)
  c := ClassExtended(Result); (class inquiry)

  SetException(Invalid + Underflow + Overflow + DivByZero, false);
                             (clear exceptions)
  SetException(Inexact, true); (signal inexact)

  IF c = Infinite THEN
    SetException(Overflow, true)
  ELSE IF c = DenormalNum THEN
    SetException(Underflow, true);
  ProcExit(e)                (restore caller's environment, including
                             any halt enables, and then signal
                             exceptions from subroutine)
END                           (NumFcn) ;

```