Oracle® Developer Studio 12.5: Numerical Computation Guide

Part No: E60763
June 2016

ORACLE

Oracle Developer Studio 12.5: Numerical Computation Guide

Part No: E60763

Référence: E60763

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Figures

Tables

Using This Documentation

- **Overview** – Describes the floating-point environment supported by software and hardware on SPARC-based systems and x86-based systems running the Oracle Solaris Operating System.
- **Audience** – Application developers, system developers, architects, support engineers.
- **Required knowledge** – Programming experience, software development testing, aptitude to build and compile software products

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at http://docs.oracle.com/cd/E37069_01.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

♦ ♦ ♦ CHAPTER 1

Introduction

Oracle's floating-point environment on SPARC and Intel x86 systems enables you to develop robust, high-performance, portable numerical applications. The floating-point environment can also help investigate unusual behavior of numerical programs written by others. These systems implement the arithmetic model specified by IEEE Standard 754 for Binary Floating Point Arithmetic. This manual explains how to use the options and flexibility provided by the IEEE Standard on these systems.

1.1 Floating-Point Environment

The *floating-point environment* consists of data structures and operations made available to the applications programmer by hardware, system software, and software libraries that together implement IEEE Standard 754. IEEE Standard 754 makes it easier to write numerical applications. It is a solid, well-thought-out basis for computer arithmetic that advances the art of numerical programming.

For example, the hardware provides storage formats corresponding to the IEEE data formats, operations on data in such formats, control over the rounding of results produced by these operations, status flags indicating the occurrence of IEEE numeric exceptions, and the IEEE-prescribed result when such an exception occurs in the absence of a user-defined handler for it. System software supports IEEE exception handling. The software libraries, including the math libraries, `libm` and `libsunmath`, implement functions such as `exp(x)` and `sin(x)` in a way that follows the spirit of IEEE Standard 754 with respect to the raising of exceptions. When a floating-point arithmetic operation has no well-defined result, the system communicates this fact to the user by *raising an exception*. The math libraries also provide function calls that handle special IEEE values like `Inf` (infinity) or `NaN` (Not a Number).

The three constituents of the floating-point environment interact in subtle ways, and those interactions are generally invisible to the applications programmer. The programmer sees only the computational mechanisms prescribed or recommended by the IEEE standard. In general, this manual guides programmers to make full and efficient use of the IEEE mechanisms so that they can write application software effectively.

Many questions about floating-point arithmetic concern elementary operations on numbers. Consider the following questions:

- What is the result of an operation when the infinitely precise result is not representable in the computer system?
- Are elementary operations like multiplication and addition commutative?

Another class of questions is connected to exceptions and exception handling. For example, what happens when you do the following?:

- Multiply two very large numbers
- Divide by zero
- Attempt to compute the square root of a negative number

In some other arithmetics, the first class of questions might not have the expected answers, or the exceptional cases in the second class are treated the same: the program aborts on the spot. In some very old machines, the computation proceeds, but with non-useful results.

The IEEE Standard 754 ensures that operations yield the mathematically expected results with the expected properties. It also ensures that exceptional cases yield specified results, unless the user specifically makes other choices.

In this manual, there are references to terms like `NaN` or *subnormal number*. The "Glossary" on page 197 defines terms related to floating-point arithmetic.

♦ ♦ ♦ CHAPTER 2

IEEE Arithmetic

This chapter discusses the arithmetic model specified by the ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (“the IEEE standard” or “IEEE 754” for short). All SPARC® and x86 processors use IEEE arithmetic. Oracle Developer Studio compilers support the features of IEEE arithmetic. This chapter discusses the following topics:

2.1 IEEE Arithmetic Model

This section describes the IEEE 754-1985 specification. The IEEE Standard was substantially revised in 2008.

2.1.1 What Is IEEE Arithmetic?

IEEE 754 specifies:

- Two basic floating-point formats: *single* and *double*.

  The IEEE single format has a significand precision of 24 bits and occupies 32 bits overall. The IEEE double format has a significand precision of 53 bits and occupies 64 bits overall.
- Two classes of extended floating-point formats: *single extended* and *double extended*.

  The standard does not prescribe the exact precision and size of these formats, but it does specify the minimum precision and size. For example, an IEEE double extended format must have a significand precision of at least 64 bits and occupy at least 79 bits overall.
- Accuracy requirements on floating-point operations: *add, subtract, multiply, divide, square root, remainder, round numbers in floating-point format to integer values, convert between*

different floating-point formats, convert between floating-point and integer formats, and compare.

The remainder and compare operations must be exact. Each of the other operations must deliver to its destination the exact result, unless there is no such result or that result does not fit in the destination's format. In the latter case, the operation must minimally modify the exact result according to the rules of prescribed rounding modes, presented below, and deliver the result so modified to the operation's destination.

- Accuracy, monotonicity and identity requirements for conversions between decimal strings and binary floating-point numbers in either of the basic floating-point formats.

  For operands lying within specified ranges, these conversions must produce exact results, if possible, or minimally modify such exact results in accordance with the rules of the prescribed rounding modes. For operands not lying within the specified ranges, these conversions must produce results that differ from the exact result by no more than a specified tolerance that depends on the rounding mode.
- Five types of IEEE floating-point exceptions, and the conditions for indicating to the user the occurrence of exceptions of these types.

  The five types of floating-point exceptions are *invalid operation, division by zero, overflow, underflow,* and *inexact.*
- Four rounding directions: *toward the nearest representable value*, with "even" values preferred whenever there are two nearest representable values; *toward negative infinity* (down); *toward positive infinity* (up); and *toward 0* (chop).
- Rounding precision; for example, if a system delivers results in double extended format, the user should be able to specify that such results are to be rounded to the precision of either the single or double format.

The IEEE standard also recommends support for user handling of exceptions.

The features required by the IEEE standard make it possible to support interval arithmetic, the retrospective diagnosis of anomalies, efficient implementations of standard elementary functions like `exp` and `cos`, multiple precision arithmetic, and many other tools that are useful in numerical computation.

IEEE 754 floating-point arithmetic offers users greater control over computation than does any other kind of floating-point arithmetic. The IEEE standard simplifies the task of writing numerically sophisticated, portable programs not only by imposing rigorous requirements on conforming implementations, but also by allowing such implementations to provide refinements and enhancements to the standard itself.

2.2 IEEE Formats

This section describes how floating-point data is stored in memory. It summarizes the precisions and ranges of the different IEEE storage formats.

2.2.1 Storage Formats

A floating-point format is a data structure specifying the fields that comprise a floating-point numeral, the layout of those fields, and their arithmetic interpretation. A floating-point *storage* format specifies how a floating-point format is stored in memory. The IEEE standard defines the formats, but the choice of storage formats is left to the implementers.

Assembly language software sometimes relies on using the storage formats, but higher level languages usually deal only with the linguistic notions of floating-point data types. These types have different names in different high-level languages, and correspond to the IEEE formats as shown in Table 1, “IEEE Formats and Language Types,” on page 19.

TABLE 1 IEEE Formats and Language Types

IEEE Precision	C, C++	Fortran
single	float	`REAL` or `REAL*4`
double	double	`DOUBLE PRECISION` or `REAL*8`
double extended	long double (x86)	—
quadruple	long double (SPARC)	`REAL*16`

IEEE 754 specifies exactly the single and double floating-point formats, and it defines a class of extended formats for each of these two basic formats. The `long double` and `REAL*16` types shown in Table 1, “IEEE Formats and Language Types,” on page 19 refer to one of the class of double extended formats defined by the IEEE standard.

The following sections describe in detail each of the storage formats used for the IEEE floating-point formats on SPARC and x86 platforms.

2.2.2 Single Format

The IEEE single format consists of three fields: a 23-bit fraction `f`; an 8-bit biased exponent `e`; and a 1-bit sign `s`. These fields are stored contiguously in one 32-bit word, as shown in the following figure. Bits 0:22 contain the 23-bit fraction, `f`, with bit 0 being the least significant

bit of the fraction and bit 22 being the most significant; bits 23:30 contain the 8-bit biased exponent, `e`, with bit 23 being the least significant bit of the biased exponent and bit 30 being the most significant; and the highest-order bit 31 contains the sign bit, `s`.

FIGURE 1 Single Storage Format

Table 2, "Values Represented by Bit Patterns in IEEE Single Format," on page 20 shows the correspondence between the values of the three constituent fields `s`, `e` and `f`, on the one hand, and the value represented by the single- format bit pattern on the other; *u* means that the value of the indicated field is irrelevant to the determination of the value of the particular bit patterns in single format.

TABLE 2 Values Represented by Bit Patterns in IEEE Single Format

Single-Format Bit Pattern	Value
0 < e < 255	$(-1)^s \times 2^{e-127} \times 1.f$ (normal numbers)
e = 0; f ≠ 0 (at least one bit in f is nonzero)	$(-1)^s \times 2^{-126} \times 0.f$ (subnormal numbers)
e = 0; f = 0 (all bits in f are zero)	$(-1)^s \times 0.0$ (signed zero)
s = 0; e = 255; f = 0 (all bits in f are zero)	+INF (positive infinity)
s = 1; e = 255; f = 0 (all bits in f are zero)	–INF (negative infinity)
s = u; e = 255; f ≠ 0 (at least one bit in f is nonzero)	NaN (Not-a-Number)

Notice that when e < 255, the value assigned to the single format bit pattern is formed by inserting the binary radix point immediately to the left of the fraction's most significant bit, and inserting an implicit bit immediately to the left of the binary point, thus representing in binary positional notation a mixed number (whole number plus fraction, wherein 0 ≤ fraction < 1).

The mixed number thus formed is called the *single-format significand*. The implicit bit is so named because its value is not explicitly given in the single- format bit pattern, but is implied by the value of the biased exponent field.

For the single format, the difference between a normal number and a subnormal number is that the leading bit of the significand (the bit to left of the binary point) of a normal number is 1, whereas the leading bit of the significand of a subnormal number is 0. Single-format subnormal numbers were called single-format denormalized numbers in IEEE Standard 754.

The 23-bit fraction combined with the implicit leading significand bit provides 24 bits of precision in single-format normal numbers.

Examples of important bit patterns in the single-storage format are shown in Table 3, “Bit Patterns in Single-Storage Format and Their IEEE Values,” on page 21. The maximum positive normal number is the largest finite number representable in IEEE single format. The minimum positive subnormal number is the smallest positive number representable in IEEE single format. The minimum positive normal number is often referred to as the underflow threshold. (The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.)

TABLE 3 Bit Patterns in Single-Storage Format and Their IEEE Values

Common Name	Bit Pattern (Hex)	Decimal Value
+0	`00000000`	0.0
–0	`80000000`	–0.0
1	`3f800000`	1.0
2	`40000000`	2.0
maximum normal number	`7f7fffff`	3.40282347e+38
minimum positive normal number	`00800000`	1.17549435e–38
maximum subnormal number	`007fffff`	1.17549421e–38
minimum positive subnormal number	`00000001`	1.40129846e–45
+∞	`7f800000`	Infinity
–∞	`ff800000`	–Infinity
Not-a-Number	`7fc00000`	NaN

A NaN (Not a Number) can be represented with any of the many bit patterns that satisfy the definition of a NaN. The hex value of the NaN shown in Table 3, “Bit Patterns in Single-Storage Format and Their IEEE Values,” on page 21 is just one of the many bit patterns that can be used to represent a NaN.

2.2.3 Double Format

The IEEE double format consists of three fields: a 52-bit fraction, `f`; an 11-bit biased exponent, `e`; and a 1-bit sign, `s`. These fields are stored contiguously in two successively addressed 32-bit words, as shown in the following figure.

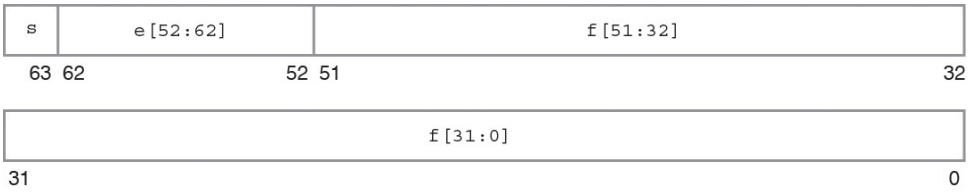
In the SPARC architecture, the higher address 32-bit word contains the least significant 32 bits of the fraction, while in the x86 architecture the lower address 32 - bit word contains the least significant 32 bits of the fraction.

If `f[31:0]` denotes the least significant 32 bits of the fraction, then bit 0 is the least significant bit of the entire fraction and bit 31 is the most significant of the 32 least significant fraction bits.

In the other 32-bit word, bits 0:19 contain the 20 most significant bits of the fraction, `f[51:32]`, with bit 0 being the least significant of these 20 most significant fraction bits, and bit 19 being the most significant bit of the entire fraction; bits 20:30 contain the 11-bit biased exponent, `e`, with bit 20 being the least significant bit of the biased exponent and bit 30 being the most significant; and the highest-order bit 31 contains the sign bit, `s`.

The following figure numbers the bits as though the two contiguous 32-bit words were one 64 - bit word in which bits 0:51 store the 52-bit fraction, `f`; bits 52:62 store the 11-bit biased exponent, `e`; and bit 63 stores the sign bit, `s`.

FIGURE 2 Double-Storage Format



The values of the bit patterns in these three fields determine the value represented by the overall bit pattern.

Table 4, “Values Represented by Bit Patterns in IEEE Double Format,” on page 23 shows the correspondence between the values of the bits in the three constituent fields, on the one hand, and the value represented by the double-format bit pattern on the other; *u* means the value of the indicated field is irrelevant to the determination of value for the particular bit pattern in double format.

TABLE 4 Values Represented by Bit Patterns in IEEE Double Format

Double-Format Bit Pattern	Value
0 < e < 2047	$(-1)^s \times 2^{e-1023} \times 1.f$ (normal numbers)
e = 0; f ≠ 0 (at least one bit in f is nonzero)	$(-1)^s \times 2^{-1022} \times 0.f$ (subnormal numbers)
e = 0; f = 0 (all bits in f are zero)	$(-1)^s \times 0.0$ (signed zero)
s = 0; e = 2047; f = 0 (all bits in f are zero)	+INF (positive infinity)
s = 1; e = 2047; f = 0 (all bits in f are zero)	–INF (negative infinity)
s = u; e = 2047; f ≠ 0 (at least one bit in f is nonzero)	NaN (Not-a-Number)

Notice that when e < 2047, the value assigned to the double-format bit pattern is formed by inserting the binary radix point immediately to the left of the fraction's most significant bit, and inserting an implicit bit immediately to the left of the binary point. The number thus formed is called the *significand*. The implicit bit is so named because its value is not explicitly given in the double-format bit pattern, but is implied by the value of the biased exponent field.

For the double format, the difference between a normal number and a subnormal number is that the leading bit of the significand (the bit to the left of the binary point) of a normal number is 1, whereas the leading bit of the significand of a subnormal number is 0. Double-format subnormal numbers were called double-format denormalized numbers in IEEE Standard 754.

The 52-bit fraction combined with the implicit leading significand bit provides 53 bits of precision in double-format normal numbers.

Examples of important bit patterns in the double-storage format are shown in Table 5, “Bit Patterns in Double-Storage Format and Their IEEE Values,” on page 24. The bit patterns in the second column appear as two 8-digit hexadecimal numbers. For the SPARC architecture, the left one is the value of the lower addressed 32-bit word, and the right one is the value of the higher addressed 32-bit word, while for the x86 architecture, the left one is the higher addressed word, and the right one is the lower addressed word. The maximum positive normal number is the largest finite number representable in the IEEE double format. The minimum positive subnormal number is the smallest positive number representable in IEEE double format. The minimum positive normal number is often referred to as the underflow threshold. (The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.)

TABLE 5 Bit Patterns in Double-Storage Format and Their IEEE Values

Common Name	Bit Pattern (Hex)	Decimal Value
+ 0	`00000000 00000000`	0.0
– 0	`80000000 00000000`	–0.0
1	`3ff00000 00000000`	1.0
2	`40000000 00000000`	2.0
max normal number	`7fefffff ffffffff`	1.7976931348623157e+308
min positive normal number	`00100000 00000000`	2.2250738585072014e–308
max subnormal number	`000fffff ffffffff`	2.2250738585072009e–308
min positive subnormal number	`00000000 00000001`	4.9406564584124654e–324
+∞	`7ff00000 00000000`	Infinity
–∞	`fff00000 00000000`	–Infinity
Not-a-Number	`7ff80000 00000000`	NaN

A NaN (Not a Number) can be represented by any of the many bit patterns that satisfy the definition of NaN. The hex value of the NaN shown in Table 5, “Bit Patterns in Double-Storage Format and Their IEEE Values,” on page 24 is just one of the many bit patterns that can be used to represent a NaN.

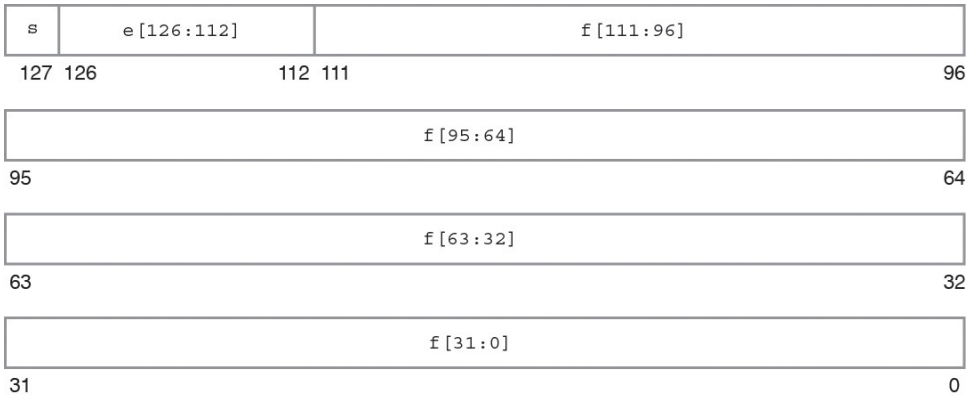
2.2.4 Quadruple Format

The floating-point environment's quadruple-precision format also conforms to the IEEE definition of double-extended format. This format is not in Oracle Developer Studio C/C++ compilers for x86. The quadruple-precision format occupies four 32-bit words and consists of three fields: a 112-bit fraction `f`; a 15-bit biased exponent `e`; and a 1-bit sign `s`. These are stored contiguously as shown in the following figure.

The highest addressed 32-bit word contains the least significant 32-bits of the fraction, denoted f[31:0]. The next two 32-bit words contain `f[63:32]` and `f[95:64]`, respectively. Bits 0:15 of the next word contain the 16 most significant bits of the fraction, `f[111:96]`, with bit 0 being the least significant of these 16 bits, and bit 15 being the most significant bit of the entire fraction. Bits 16:30 contain the 15-bit biased exponent, `e`, with bit 16 being the least significant bit of the biased exponent and bit 30 being the most significant; and bit 31 contains the sign bit, `s`.

The following figure numbers the bits as though the four contiguous 32-bit words were one 128-bit word in which bits 0:111 store the fraction, `f`; bits 112:126 store the 15-bit biased exponent, e; and bit 127 stores the sign bit, `s`.

FIGURE 3 Quadruple Format



The values of the bit patterns in the three fields `f`, `e`, and `s`, determine the value represented by the overall bit pattern.

Table 6, "Values Represented by Bit Patterns," on page 25 shows the correspondence between the values of the three constituent fields and the value represented by the bit pattern in quadruple-precision format. `u` means don't care, because the value of the indicated field is irrelevant to the determination of values for the particular bit patterns.

TABLE 6 Values Represented by Bit Patterns

Quadruple Bit Pattern	Value
$0 < e < 32767$	$(-1)^s \times 2^{e-16383} \times 1.f$ (normal numbers)
$e = 0$, $f \neq 0$ (at least one bit in `f` is nonzero)	$(-1)^s \times 2^{-16382} \times 0.f$ (subnormal numbers)
$e = 0$, $f = 0$ (all bits in `f` are zero)	$(-1)^s \times 0.0$ (signed zero)
$s = 0$, $e = 32767$, $f = 0$ (all bits in `f` are zero)	+INF (positive infinity)
$s = 1$, $e = 32767$; $f = 0$ (all bits in `f` are zero)	-INF (negative infinity)
$s = u$, $e = 32767$, $f \neq 0$ (at least one bit in `f` is nonzero)	NaN (Not-a-Number)

Examples of important bit patterns in the quadruple-precision double-extended storage format are shown in Table 7, “Bit Patterns in Quadruple Format,” on page 26. The bit patterns in the second column appear as four 8-digit hexadecimal numbers. The left-most number is the value of the lowest addressed 32-bit word, and the right-most number is the value of the highest addressed 32-bit word. The maximum positive normal number is the largest finite number representable in the quadruple precision format. The minimum positive subnormal number is the smallest positive number representable in the quadruple precision format. The minimum positive normal number is often referred to as the underflow threshold. (The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.)

TABLE 7 Bit Patterns in Quadruple Format

Common Name	Bit Pattern (SPARC)	Decimal Value
+0	`00000000 00000000 00000000 00000000`	0.0
–0	`80000000 00000000 00000000 00000000`	–0.0
1	`3fff0000 00000000 00000000 00000000`	1.0
2	`40000000 00000000 00000000 00000000`	2.0
max normal	`7ffeffff ffffffff ffffffff ffffffff`	1.189731495357231765085759326628007 0e+4932
min normal	`00010000 00000000 00000000 00000000`	3.362103143112093506262677817321752 6e–4932
max subnormal	`0000ffff ffffffff far-off ffffffff`	3.362103143112093506262677817321752 0e–4932
min pos subnormal	`00000000 00000000 00000000 00000001`	6.475175119438025110924438958227646 6e–4966
+∞	`7fff0000 00000000 00000000 00000000`	+∞
–∞	`ffff0000 00000000 00000000 00000000`	–∞
Not-a-Number	`7fff8000 00000000 00000000 00000000`	NaN

The hex value of the NaN shown in Table 7, “Bit Patterns in Quadruple Format,” on page 26 is just one of the many bit patterns that can be used to represent NaNs.

2.2.5 Double-Extended Format (x86)

This floating-point environment's double-extended format conforms to the IEEE definition of double-extended formats. It consists of four fields: a 63-bit fraction `f`; a 1-bit explicit leading significand bit `j`; a 15-bit biased exponent `e`; and a 1-bit sign `s`. This format is not available as a language type for Oracle Developer Studio Fortran or for C/C++ for SPARC.

In the family of x86 architectures, these fields are stored contiguously in ten successively addressed 8-bit bytes. However, the UNIX System V Application Binary Interface Intel 386

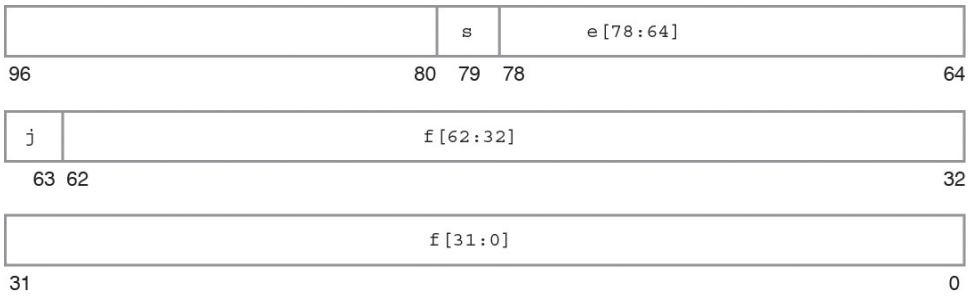
Processor Supplement (Intel ABI) requires that double-extended parameters and results occupy three consecutively addressed 32-bit words in the stack, with the most significant 16 bits of the highest addressed word being unused, as shown in the following figure.

The lowest addressed 32-bit word contains the least significant 32 bits of the fraction, `f[31:0]`, with bit 0 being the least significant bit of the entire fraction and bit 31 being the most significant of the 32 least significant fraction bits. In the middle addressed 32-bit word, bits 0:30 contain the 31 most significant bits of the fraction, `f[62:32]`, with bit 0 being the least significant of these 31 most significant fraction bits, and bit 30 being the most significant bit of the entire fraction; bit 31 of this middle addressed 32-bit word contains the explicit leading significand bit, `j`.

In the highest addressed 32-bit word, bits 0:14 contain the 15-bit biased exponent, `e`, with bit 0 being the least significant bit of the biased exponent and bit 14 being the most significant; and bit 15 contains the sign bit, `s`. Although the highest order 16 bits of this highest addressed 32-bit word are unused by the family of x86 architectures, their presence is essential for conformity to the Intel ABI, as indicated above.

The following figure numbers the bits as though the three contiguous 32-bit words were one 96-bit word in which bits 0:62 store the 63-bit fraction, `f`; bit 63 stores the explicit leading significand bit, `j`; bits 64:78 store the 15-bit biased exponent, `e`; and bit 79 stores the sign bit, `s`.

FIGURE 4 Double-Extended Format (x86)



The values of the bit patterns in the four fields `f`, `j`, `e` and `s`, determine the value represented by the overall bit pattern.

Table 8, “Values Represented by Bit Patterns (x86),” on page 28 shows the correspondence between the hex representations of the four constituent fields and the values represented by the

bit patterns. *u* means the value of the indicated field is irrelevant to the determination of value for the particular bit patterns.

TABLE 8 Values Represented by Bit Patterns (x86)

Double-Extended Bit Pattern (x86)	Value
j = 0, 0 < e <32767	Unsupported
j = 1, 0 < e < 32767	$(-1)^s \times 2^{e-16383} \times 1.f$ (normal numbers)
j = 0, e = 0; f ≠ 0	$(-1)^s \times 2^{-16382} \times 0.f$ (subnormal numbers)
(at least one bit in f is nonzero)	
j = 1, e = 0	$(-1)^s \times 2^{-16382} \times 1.f$ (pseudo-denormal numbers)
j = 0, e = 0, f = 0	$(-1)^s \times 0.0$ (signed zero)
(all bits in f are zero)	
j = 1; s = 0; e = 32767; f = 0 (all bits in f are zero)	+INF (positive infinity)
j = 1; s = 1; e = 32767; f = 0 (all bits in f are zero)	–INF (negative infinity)
j = 1; s = *u*; e = 32767; f = .1*uuu* — *uu*	QNaN (quiet NaNs)
j = 1; s = *u*; e = 32767; f = .0*uuu* — *uu* ≠ 0	SNaN (signaling NaNs)
(at least one of the *u* in f is nonzero)	

Notice that bit patterns in double-extended format do *not*have an implicit leading significand bit. The leading significand bit is given explicitly as a separate field, j, in the double-extended format. However, when e≠ 0, any bit pattern with j = 0 is unsupported in the sense that using such a bit pattern as an operand in floating-point operations provokes an invalid operation exception.

The union of the disjoint fields j and f in the double extended format is called the *significand*. When e < 32767 and j = 1, or when e = 0 and j = 0, the significand is formed by inserting the binary radix point between the leading significand bit, j, and the fraction's most significant bit.

In the x86 double-extended format, a bit pattern whose leading significand bit j is 0 and whose biased exponent field e is also 0 represents a subnormal number, whereas a bit pattern whose leading significand bit j is 1 and whose biased exponent field e is nonzero represents a normal number. Because the leading significand bit is represented explicitly rather than being inferred from the value of the exponent, this format also admits bit patterns whose biased exponent is 0, like the subnormal numbers, but whose leading significand bit is 1. Each such bit pattern actually represents the same value as the corresponding bit pattern whose biased exponent field is 1, i.e., a normal number, so these bit patterns are called *pseudo-denormals*. Subnormal numbers were called denormalized numbers in IEEE Standard 754-1985. Pseudo-denormals are merely an artifact of the x86 double-extended format's encoding; they are implicitly converted

to the corresponding normal numbers when they appear as operands, and they are never generated as results.

TABLE 9 Bit Patterns in Double-Extended Format and Their Values (x86)

Common Name	Bit Pattern (x86)	Decimal Value
+0	`0000 00000000 00000000`	0.0
–0	`8000 00000000 00000000`	–0.0
1	`3fff 80000000 00000000`	1.0
2	`4000 80000000 00000000`	2.0
max normal	`7ffe ffffffff ffffffff`	1.18973149535723176505e+4932
min positive normal	`0001 80000000 00000000`	3.36210314311209350626e–4932
max subnormal	`0000 7fffffff ffffffff`	3.36210314311209350608e–4932
min positive subnormal	`0000 00000000 00000001`	3.64519953188247460253e–4951
+∞	`7fff 80000000 00000000`	+∞
–∞	`ffff 80000000 00000000`	–∞
quiet NaN with greatest fraction	`7fff ffffffff ffffffff`	QNaN
quiet NaN with least fraction	`7fff c0000000 00000000`	QNaN
signaling NaN with greatest fraction	`7fff bfffffff ffffffff`	SNaN
signaling NaN with least fraction	`7fff 80000000 00000001`	SNaN

Examples of important bit patterns in the double-extended storage format appear in the preceding table. The bit patterns in the second column appear as one 4-digit hexadecimal number, which is the value of the 16 least significant bits of the highest addressed 32-bit word (recall that the most significant 16 bits of this highest addressed 32-bit word are unused, so their value is not shown), followed by two 8-digit hexadecimal numbers, of which the left one is the value of the middle addressed 32-bit word, and the right one is the value of the lowest addressed 32-bit word. The maximum positive normal number is the largest finite number representable in the x86 double-extended format. The minimum positive subnormal number is the smallest positive number representable in the double-extended format. The minimum positive normal number is often referred to as the underflow threshold. The decimal values for the maximum and minimum normal and subnormal numbers are approximate; they are correct to the number of figures shown.

A NaN (Not a Number) can be represented by any of the many bit patterns that satisfy the definition of NaN. The hex values of the NaNs shown in the preceding table illustrate that the leading (most significant) bit of the fraction field determines whether a NaN is quiet (leading fraction bit = 1) or signaling (leading fraction bit = 0).

2.2.6 Ranges and Precisions in Decimal Representation

This section covers the notions of range and precision for a given storage format. It includes the ranges and precisions corresponding to the IEEE single, double, and quadruple formats and to the implementations of IEEE double-extended format on x86 architectures. For concreteness, in defining the notions of range and precision, refer to the IEEE single format.

The IEEE standard specifies that 32 bits be used to represent a floating-point number in single format. Because there are only finitely many combinations of 32 zeroes and ones, only finitely many numbers can be represented by 32 bits.

It is natural to ask what are the decimal representations of the largest and smallest positive numbers that can be represented in this particular format.

If you introduce the concept of range, you can rephrase the question instead to ask what is the range, in decimal notation, of numbers that can be represented by the IEEE single format?

Taking into account the precise definition of IEEE single format, you can prove that the range of floating-point numbers that can be represented in IEEE single format, if restricted to positive normalized numbers, is as follows:

$1.175... \times (10^{-38})$ to $3.402... \times (10^{+38})$

A second question refers to the precision of the numbers represented in a given format. These notions are explained by looking at some pictures and examples.

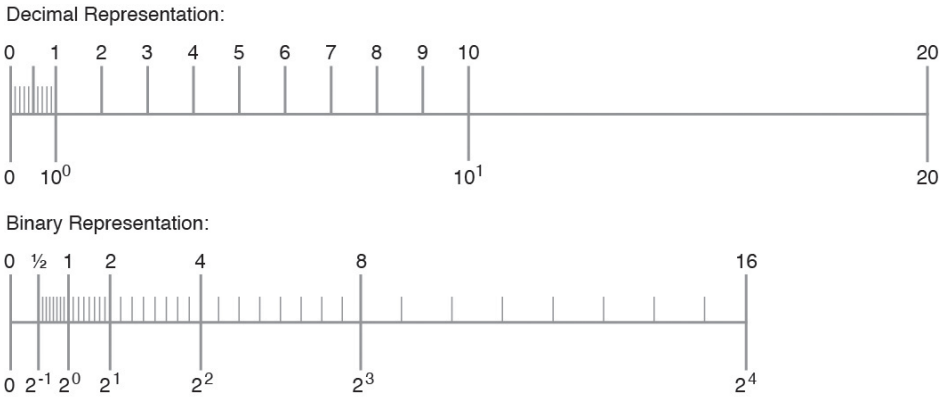
The IEEE standard for binary floating-point arithmetic specifies the set of numerical values representable in the single format. Remember that this set of numerical values is described as a set of binary floating-point numbers. The significand of the IEEE single format has 23 bits, which together with the implicit leading bit, yield 24 digits (bits) of (binary) precision.

One obtains a different set of numerical values by marking the numbers (representable by `q` decimal digits in the significand) on the number line:

$x = (x_1.x_2\, x_3...x_q) \times (10^n)$

The following figure exemplifies this situation:

FIGURE 5 Comparison of a Set of Numbers Defined by Digital and Binary Representation



Notice that the two sets are different. Therefore, estimating the number of significant decimal digits corresponding to 24 significant binary digits, requires reformulating the problem.

Reformulate the problem in terms of converting floating-point numbers between binary representations (the internal format used by the computer) and the decimal format (the format users are usually interested in). In fact, you might want to convert from decimal to binary and back to decimal, as well as convert from binary to decimal and back to binary.

It is important to notice that because the sets of numbers are different, conversions are in general inexact. If done correctly, converting a number from one set to a number in the other set results in choosing one of the two neighboring numbers from the second set (which one specifically is a question related to rounding).

Consider some examples. Suppose you are trying to represent a number with the following decimal representation in IEEE single format:

$x = x_1.x_2\ x_3... \times 10^n$

Because there are only finitely many real numbers that can be represented exactly in IEEE single format, and not all numbers of the above form are among them, in general it will be impossible to represent such numbers exactly. For example, let

$y = 838861.2$, $z = 1.3$

and run the following Fortran program:

```
REAL Y, Z
```

```
  Y = 838861.2
  Z = 1.3
 WRITE(*,40) Y
40  FORMAT("y: ",1PE18.11)
  WRITE(*,50) Z
50  FORMAT("z: ",1PE18.11)
```

The output from this program should be similar to the following:

```
y: 8.38861187500E+05
z: 1.29999995232E+00
```

The difference between the value 8.388612×10^5 assigned to *y* and the value printed out is 0.000000125, which is seven decimal orders of magnitude smaller than *y*. The accuracy of representing *y* in IEEE single format is about 6 to 7 significant digits, or that *y* has about *six significant* digits if it is to be represented in IEEE single format.

Similarly, the difference between the value 1.3 assigned to *z* and the value printed out is 0.00000004768, which is eight decimal orders of magnitude smaller than *z*. The accuracy of representing *z* in IEEE single format is about 7 to 8 significant digits, or that *z* has about seven *significant digits* if it is to be represented in IEEE single format.

Assume you convert a decimal floating point number *a* to its IEEE single format binary representation *b*, and then translate *b* back to a decimal number *c*; how many orders of magnitude are between *a* and *a* - *c*?

Rephrase the question:

What is the number of *significant decimal digits* of *a* in the IEEE single format representation, or how many decimal digits are to be trusted as accurate when one represents *x* in IEEE single format?

The number of significant decimal digits is always between 6 and 9, that is, at least 6 digits, but not more than 9 digits are accurate (with the exception of cases when the conversions are exact, when infinitely many digits could be accurate).

Conversely, if you convert a binary number in IEEE single format to a decimal number, and then convert it back to binary, generally, you need to use at least 9 decimal digits to ensure that after these two conversions you obtain the number you started from.

The complete picture is given in Table 10, “Range and Precision of Storage Formats,” on page 32:

TABLE 10 Range and Precision of Storage Formats

Format	Significant Digits (Binary)	Smallest Positive Normal Number	Largest Positive Number	Significant Digits (Decimal)
single	24	1.175... 10^{-38}	3.402... 10^{+38}	6-9

Format	Significant Digits (Binary)	Smallest Positive Normal Number	Largest Positive Number	Significant Digits (Decimal)
double	53	2.225... 10^{-308}	1.797... 10^{+308}	15-17
quadruple	113	3.362... 10^{-4932}	1.189... 10^{+4932}	33-36
double extended (x86)	64	3.362... 10^{-4932}	1.189... 10^{+4932}	18-21

2.2.7 Base Conversion in the Oracle Solaris Environment

Base conversion refers to the transformation of a number represented in one base to a number represented in another base. I/O routines such as `printf` and `scanf` in C and `read`, `write`, and `print` in Fortran involve base conversion between numbers represented in bases 2 and 10:

- Base conversion from base 10 to base 2 occurs when reading in a number in conventional decimal notation and storing it in internal binary format.
- Base conversion from base 2 to base 10 occurs when printing an internal binary value as an ASCII string of decimal digits.

In the Oracle Solaris environment, the fundamental routines for base conversion in all languages are contained in the standard C library, `libc`. These routines use table-driven algorithms that yield correctly rounded conversion between any input and output formats subject to modest restrictions on the lengths of the strings of decimal digits involved. In addition to their accuracy, table-driven algorithms reduce the worst-case times for correctly rounded base conversion.

The 1985 IEEE standard requires correct rounding for typical numbers whose magnitudes range from 10–44 to 10+44 but permits slightly incorrect rounding for larger exponents. See section 5.6 of IEEE Standard 754. The `libc` table-driven algorithms round correctly throughout the entire range of single, double, and double extended formats, as required by the revised 754-2008.

In C, conversions between decimal strings and binary floating-point values are always rounded correctly in accordance with IEEE 754: the converted result is the number representable in the result's format that is nearest to the original value in the direction specified by the current rounding mode. When the rounding mode is round-to-nearest and the original value lies exactly halfway between two representable numbers in the result format, the converted result is the one whose least significant digit is even. These rules apply to conversions of constants in source code performed by the compiler as well as to conversions of data performed by the program using standard library routines.

In Fortran, conversions between decimal strings and binary floating-point values are rounded correctly following the same rules as C by default. For I/O conversions, the “round-ties-to-even” rule in round-to-nearest mode can be overridden, either by using the `ROUNDING=` specifier in the program or by compiling with the `-iorounding` flag. See the *Oracle Developer Studio 12.5: Fortran User’s Guide* and the `f95`(1) man page for more information.

See Appendix F, “References ” for references on base conversion, particularly Coonen's thesis and Sterbenz's book.

2.3 Underflow

Underflow occurs, roughly speaking, when the result of an arithmetic operation is so small that it cannot be stored in its intended destination format without suffering a rounding error that is larger than usual.

2.3.1 Underflow Thresholds

Table 11, “Underflow Thresholds,” on page 34 shows the underflow thresholds for single, double, and double-extended precision.

TABLE 11 Underflow Thresholds

Destination Precision	Underflow Threshold	
single	smallest normal number	1.17549435e–38
	largest subnormal number	1.17549421e–38
double	smallest normal number	2.2250738585072014e–308
	largest subnormal number	2.2250738585072009e–308
quadruple	smallest normal number	3.362103143112093506262677817321752 6e–4932
	largest subnormal number	3.362103143112093506262677817321752 0e–4932
double-extended (x86)	smallest normal number	3.362103143112093506 26e–4932
	largest subnormal number	3.362103143112093505 90e–4932

The positive subnormal numbers are those numbers between the smallest normal number and zero. Subtracting two (positive) tiny numbers that are near the smallest normal number might produce a subnormal number. Or, dividing the smallest positive normal number by two produces a subnormal result.

The presence of subnormal numbers provides greater precision to floating-point calculations that involve small numbers, although the subnormal numbers themselves have fewer bits of precision than normal numbers. Producing subnormal numbers (rather than returning the answer zero) when the mathematically correct result has magnitude less than the smallest positive normal number is known as gradual underflow.

There are several other ways to deal with such *underflow* results. One way, common in the past, was to flush those results to zero. This method is known as *abrupt underflow* and was the default on most mainframes before the advent of the IEEE Standard.

The mathematicians and computer designers who drafted IEEE Standard 754 considered several alternatives while balancing the desire for a mathematically robust solution with the need to create a standard that could be implemented efficiently.

2.3.2 How Does IEEE Arithmetic Treat Underflow?

IEEE Standard 754 chooses gradual underflow as the preferred method for dealing with underflow results. This method amounts to defining two representations for stored values, normal and subnormal.

Recall that the IEEE format for a normal floating-point number is:

$(-1)^s \times (2^{(e-bias)}) \times 1.f$

where *s* is the sign bit, *e* is the biased exponent, and *f* is the fraction. Only *s*, *e*, and *f* need to be stored to fully specify the number. Because the implicit leading bit of the significand is defined to be 1 for normal numbers, it need not be stored.

The smallest positive normal number that can be stored, then, has the negative exponent of greatest magnitude and a fraction of all zeros. Even smaller numbers can be accommodated by considering the leading bit to be zero rather than one. In the double-precision format, this effectively extends the minimum exponent from 10^{-308} to 10^{-324}, because the fraction part is 52 bits long (roughly 16 decimal digits.) These are the *subnormal* numbers; returning a subnormal number, rather than flushing an underflowed result to zero, is *gradual underflow*.

Clearly, the smaller a subnormal number, the fewer nonzero bits in its fraction; computations producing subnormal results do not enjoy the same bounds on relative round-off error as computations on normal operands. However, the key fact about gradual underflow is that its use implies the following:

- Underflowed results need never suffer a loss of accuracy any greater than that which results from ordinary round-off error.
- Addition, subtraction, comparison, and remainder are always exact when the result is very small.

Recall that the IEEE format for a subnormal floating-point number is:

$(-1)^s \times (2^{(-bias+1)}) \times 0.f$

where *s* is the sign bit, the biased exponent *e* is zero, and *f* is the fraction. Note that the implicit power-of-two bias is one greater than the bias in the normal format, and the implicit leading bit of the fraction is zero.

Gradual underflow allows you to extend the lower range of representable numbers. It is not *smallness* that renders a value questionable, but its associated error. Algorithms exploiting subnormal numbers have smaller error bounds than other systems. The next section provides some mathematical justification for gradual underflow.

2.3.3 Why Gradual Underflow?

The purpose of subnormal numbers is not to avoid underflow/overflow entirely, as some other arithmetic models do. Rather, subnormal numbers eliminate underflow as a cause for concern for a variety of computations, typically, multiply followed by add. For a more detailed discussion, see *Underflow and the Reliability of Numerical Software* by James Demmel and *Combating the Effects of Underflow and Overflow in Determining Real Roots of Polynomials* by S. Linnainmaa.

The presence of subnormal numbers in the arithmetic means that untrapped underflow, which implies loss of accuracy, cannot occur on addition or subtraction. If *x* and *y* are within a factor of two, then $x - y$ is error-free. This is critical to a number of algorithms that effectively increase the working precision at critical places in algorithms.

In addition, gradual underflow means that errors due to underflow are no worse than usual round-off error. This is a much stronger statement than can be made about any other method of handling underflow, and this fact is one of the best justifications for gradual underflow.

2.3.4 Error Properties of Gradual Underflow

Most of the time, floating-point results are rounded:

computed result = true result + round-off

One convenient measure of how large the round-off can be is called a *unit in the last place*, abbreviated *ulp*. The least significant bit of the fraction of a floating-point number in its standard representation is its *last place*. The value represented by this bit (e.g., the absolute difference between the two numbers whose representations are identical except for this bit) is

a *unit in the last place* of that number. If the computed result is obtained by rounding the true result to the nearest representable number, then clearly the round-off error is no larger than half a unit in the last place of the computed result. In other words, in IEEE arithmetic with rounding mode to nearest, it will be the computed result of the following:

$0 \le |round\text{-}off| \le \frac{1}{2}\text{ulp}$

Note that an ulp is a relative quantity. An ulp of a very large number is itself very large, while an ulp of a tiny number is itself tiny. This relationship can be made explicit by expressing an ulp as a function: *ulp(x)* denotes a unit in the last place of the floating-point number *x*.

Moreover, an ulp of a floating-point number depends on the precision to which that number is represented. For example, Table 12, “ulp(1) in Four Different Precisions,” on page 37 shows the values of ulp(1) in each of the four floating-point formats described above:

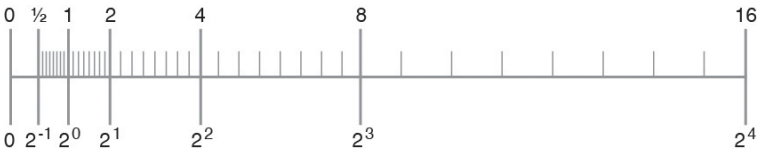
TABLE 12 ulp(1) in Four Different Precisions

Precision	Value
single	ulp(1) = 2^-23 ~ 1.192093e-07
double	ulp(1) = 2^-52 ~ 2.220446e-16
double extended (x86)	ulp(1) = 2^-63 ~ 1.084202e-19
quadruple	ulp(1) = 2^-112 ~ 1.925930e-34

Recall that only a finite set of numbers can be exactly represented in any computer arithmetic. As the magnitudes of numbers get smaller and approach zero, the gap between neighboring representable numbers narrows. Conversely, as the magnitude of numbers gets larger, the gap between neighboring representable numbers widens.

For example, imagine you are using a binary arithmetic that has only 3 bits of precision. Then, between any two powers of 2, there are $2^3 = 8$ representable numbers, as shown in the following figure.

FIGURE 6 Number Line



The number line shows how the gap between numbers doubles from one exponent to the next.

In the IEEE single format, the difference in magnitude between the two smallest positive subnormal numbers is approximately 10^{-45}, whereas the difference in magnitude between the two largest finite numbers is approximately 10^{31}!

In Table 13, "Gaps Between Representable Single-Format Floating-Point Numbers," on page 38, `nextafter(x,+∞)` denotes the next representable number after `x` as you move along the number line towards +∞.

TABLE 13 Gaps Between Representable Single-Format Floating-Point Numbers

x	nextafter(x, +•)	Gap
0.0	1.4012985e–45	1.4012985e–45
1.1754944e–38	1.1754945e–38	1.4012985e–45
1.0	1.0000001	1.1920929e–07
2.0	2.0000002	2.3841858e–07
16.000000	16.000002	1.9073486e–06
128.00000	128.00002	1.5258789e–05
1.0000000e+20	1.0000001e+20	8.7960930e+12
9.9999997e+37	1.0000001e+38	1.0141205e+31

Any conventional set of representable floating-point numbers has the property that the worst effect of one inexact result is to introduce an error no worse than the distance to one of the representable neighbors of the computed result. When subnormal numbers are added to the representable set and gradual underflow is implemented, the worst effect of one inexact *or underflow*ed result is to introduce an error no greater than the distance to one of the representable neighbors of the computed result.

In particular, in the region between zero and the smallest *normal* number, the distance between any two neighboring numbers equals the distance between zero and the smallest *subnormal* number. The presence of subnormal numbers eliminates the possibility of introducing a round-off error that is greater than the distance to the nearest representable number.

Because no calculation incurs round-off error greater than the distance to any of the representable neighbors of the computed result, many important properties of a robust arithmetic environment hold, including these three:

- $x \neq y$ if and only if $x - y \neq 0$
- $(x - y) + y \approx x$, to within a rounding error in the larger of x and y
- $1/(1/x) \approx x$, when x is a normalized number, implying $1/x \neq 0$ even for the largest normalized x

An alternative underflow scheme is abrupt underflow, which flushes underflow results to zero. Abrupt underflow violates the first and second properties whenever `x – y` underflows. Also, abrupt underflow violates the third property whenever `1/x` underflows.

Let λ represent the smallest positive normalized number, which is also known as the underflow threshold. Then the error properties of gradual underflow and abrupt underflow can be compared in terms of λ.

gradual underflow: |error| < ½*ulp* in λ

abrupt underflow: |error| ≈ λ

There is a significant difference between half a unit in the last place of λ, and λ itself.

2.3.5 Two Examples of Gradual Underflow Versus Abrupt Underflow

The following are two well-known mathematical examples. The first example is code that computes an inner product.

```
sum = 0;
for (i = 0; i < n; i++) {
 sum = sum + a[i] * y[i];
}
return sum;
```

With gradual underflow, the result is as accurate as round-off allows. In abrupt underflow, a small but nonzero sum could be delivered that looks plausible but is much worse. However, it must be admitted that to avoid just these sorts of problems, clever programmers scale their calculations if they are able to anticipate where minuteness might degrade accuracy.

The second example, deriving a complex quotient, is not amenable to scaling:

$$a + i \cdot b = \frac{p + i \cdot q}{r + i \cdot s} \quad \textit{assuming } |r / s| \leq 1$$

$$= \frac{(p(r/s) + q) + i(q(r/s) - p)}{s + r(r/s)} .$$

It can be shown that, despite round-off, the computed complex result differs from the exact result by no more than what would have been the exact result if $p + i \cdot q$ and $r + i \cdot s$ each had been perturbed by no more than a few *ulps*. This error analysis holds in the face of underflows, except that when both *a* and *b* underflow, the error is bounded by a few *ulps* of $|a + i \cdot b|$ Neither conclusion is true when underflows are flushed to zero.

This algorithm for computing a complex quotient is robust, and amenable to error analysis, in the presence of gradual underflow. A similarly robust, easily analyzed, and efficient algorithm for computing the complex quotient in the face of abrupt underflow *does not exist*. In abrupt underflow, the burden of worrying about low-level, complicated details shifts from the implementer of the floating-point environment to its users.

The class of problems that succeed in the presence of gradual underflow, but fail with abrupt underflow, is larger than the users of abrupt underflow might realize. Many of the frequently used numerical techniques fall in this class, such as the following:

- Linear equation solving
- Polynomial equation solving
- Numerical integration
- Convergence acceleration
- Complex division

2.3.6 Does Underflow Matter?

Despite these examples, it can be argued that underflow rarely matters, and so, why bother? However, this argument turns upon itself.

In the absence of gradual underflow, user programs need to be sensitive to the implicit inaccuracy threshold. For example, in single precision, if underflow occurs in some parts of a calculation, and abrupt underflow is used to replace underflowed results with `0`, then accuracy can be guaranteed only to around 10^{-31}, not 10^{-38}, the usual lower range for single-precision exponents.

This means that programmers need to implement their own method of detecting when they are approaching this inaccuracy threshold, or else abandon the quest for a robust, stable implementation of their algorithm.

Some algorithms can be scaled so that computations don't take place in the constricted area near zero. However, scaling the algorithm and detecting the inaccuracy threshold can be difficult and time-consuming, even if it is not necessary for most data.

2.4 IEEE Standard 754-2008

This section discusses differences between 754-1985 and its successor 754-2008. Oracle, like other system implementers, will conform to the recommendations of 754-2008 over time, as those recommendations are defined in standards for programming languages.

In 2008, the IEEE adopted a revised IEEE Standard for Floating-Point Arithmetic, which superseded the earlier 754-1985 IEEE Standard for Binary Floating-Point Arithmetic and the 854-1987 IEEE Standard for Radix-Independent Floating-Point Arithmetic.

From a hardware point of view, the new standard is upward compatible with the older ones; existing hardware can be made compatible with the new standard if augmented by enough software. However to fully implement all the recommendations of the new standard requires significant development of language definitions, which is occurring over a period of years; commercial implementations of those definitions will then follow. Future versions of this Numerical Computation Guide for Oracle Developer Studio will describe new features recommended by the revised standard as they become available in the Oracle Developer StudioC, C++, and Fortran compilers.

The following are some important changes already in Oracle Developer Studio 12.5 :

- 754-2008 specifies 128-bit binary (and decimal) floating-point formats; the 128-bit binary format corresponds to Studio Fortran `REAL*16` and to Studio C and C++ `long double` on SPARC.
- 754-2008 requires conversion between binary and decimal formats and character sequences to be correctly rounded, whereas 754-1985 allowed slightly larger error bounds for numbers with very large or very small exponents.
- 754-2008 recommends operations to convert between binary formats and hexadecimal character sequences.
- 754-2008 requires a number of operations that were optional in 754-1985; these are already in Oracle Developer Studio compilers.
- 54-2008 specifies fused multiply-add operations; these are available in hardware in recent SPARC servers supported by Oracle Developer Studio 12.5 .

The following are some important changes that are not in Oracle Developer Studio 12.5 :

- 754-2008 recommends a number of new operations.
- 754-2008 recommends that correctly-rounded versions of a number of elementary transcendental functions be available.
- 754-2008 recommends expression evaluation attributes.
- 754-2008 no longer specifies floating-point trap handling; instead higher-level alternate exception handling attributes are recommended that can be used in a machine-independent way.

♦ ♦ ♦ CHAPTER 3

The Math Libraries

This chapter describes the math libraries provided with the Oracle Solaris OS and Oracle Developer Studio software. Besides listing each of the libraries along with its contents, this chapter discusses some of the features supported by the math libraries provided with the compiler collection, including IEEE supporting functions, random number generators, and functions that convert data between IEEE and non-IEEE formats.

The contents of the `libm` and `libsunmath` libraries are also listed on the *Intro*(3M) man page.

This chapter divides the topics into the following sections:

3.1 Oracle Solaris Math Libraries

This section describes the math libraries that are bundled with the Oracle Solaris 10 OS. These libraries are provided as shared objects and are installed in the standard location for Oracle Solaris libraries.

3.1.1 Standard Math Library

The Oracle Solaris standard math library, `libm`, contains elementary mathematical functions and support routines required by the various standards to which the Oracle Solaris operating environment conforms.

The Oracle Solaris 10 OS includes two versions of `libm`: `libm.so.1` and `libm.so.2`. `libm.so.1` provides the functions required by those standards supported by the Oracle Solaris 9 OS and earlier versions. `libm.so.2` provides the functions required by those standards supported by the Oracle Solaris 10 OS (including C99). `libm.so.1` is provided for backward compatibility so that programs compiled and linked on the Oracle Solaris 9 OS and earlier systems will continue to run unchanged. The contents of `libm.so.1` are documented in the section 3M man pages on those systems. The remainder of this chapter refers to `libm.so.2`. See the *ld*(1) and compiler manual pages for more information about dynamic linking and the options and environment variables that determine which shared objects are loaded when a program is run.

Table 14, "Contents of `libm`," on page 44 lists the functions in `libm`. For each mathematical function, the table gives only the name of the double precision version of the function. The library also contains a single precision version having the same name followed by an `f` and an extended/quadruple precision version having the same name followed by an `l`.

TABLE 14 Contents of `libm`

Type	Function Name
Algebraic functions	`cbrt`, `fdim`, `fma`, `fmax`, `fmin`, `hypot`, `sqrt`
Elementary transcendental functions	`asin`, `acos`, `atan`, `atan2`, `asinh`, `acosh`, `atanh`, `exp`, `exp2`, `expm1`, `pow`, `log`, `log1p`, `log10`, `log2`, `sin`, `cos`, `sincos`, `tan`, `sinh`, `cosh`, `tanh`
Higher transcendental functions	`j0`, `j1`, `jn`, `y0`, `y1`, `yn`, `erf`, `erfc`, `gamma`, `lgamma`, `gamma_r`, `lgamma_r`, `tgamma`
Integral rounding functions	`ceil`, `floor`, `llrint`, `llround`, `lrint`, `lround`, `modf`, `nearbyint`, `rint`, `round`, `trunc`
IEEE standard recommended functions	`copysign`, `fmod`, `ilogb`, `nextafter`, `remainder`, `scalbn`, `fabs`
IEEE classification functions	`isnan`
Old style floating-point functions	`frexp`, `ldexp`, `logb`, `scalb`, `significand`
Error handling routine (user - defined)	`matherr`
Complex functions	`cabs`, `cacos`, `cacosh`, `carg`, `casin`, `casinh`, `catan`, `catanh`, `ccos`, `ccosh`, `cexp`, `cimag`, `clog`, `conj`, `cpow`, `cproj`, `creal`, `csin`, `csinh`, `csqrt`, `ctan`, `ctanh`
C99 floating-point environment functions	`feclearexcept`, `fegetenv`, `fegetexceptflag`, `fegetprec`, `fegetround`, `feholdexcept`, `feraiseexcept`, `fesetenv`, `fesetexceptflag`, `fesetprec`, `fesetround`, `fetestexcept`, `feupdateenv`
Floating-point exception handling functions	`fex_getexcepthandler`, `fex_get_handling`, `fex_get_log`, `fex_get_log_depth`, `fex_log_entry`, `fex_merge_flags`, `fex_setexcepthandler`, `fex_set_handling`, `fex_set_log`, `fex_set_log_depth`
Other C99 functions	`nan`, `nexttoward`, `remquo`, `scalbln`

Note the following about Table 14, “Contents of `libm`,” on page 44:

1. The functions `gamma_r` and `lgamma_r` are re-entrant versions of `gamma` and `lgamma`.

2. The functions `fegetprec` and `fesetprec` are only available on x86 systems. These functions are not specified by the C99 standard.

3. Error bounds and observed errors for the transcendental functions in `libm` are tabulated on the *libm*(3LIB) man page.

3.1.2 Vector Math Library

The library `libmvec` provides routines that evaluate common mathematical functions for an entire vector of arguments. An application might invoke the routines in `libmvec` explicitly, or the compiler might invoke these routines when the `-xvector` flag is used.

`libmvec` is implemented as a primary shared object, `libmvec.so.1`, and several auxiliary shared objects that provide alternate versions of some or all of the vector functions. When a program linked with `libmvec` is run, the runtime linker automatically selects the version that offers the best performance on the host platform. For this reason, a program that uses the functions in `libmvec` might deliver slightly different results when run on different systems.

Table 15, “Contents of `libmvec`,” on page 45 lists the functions in `libmvec`.

TABLE 15 Contents of `libmvec`

Type	Function Name
Algebraic functions	`vhypot_`, `vhypotf_`, `vrhypot_`, `vrhypotf_`, `vrsqrt_`, `vrsqrtf_`, `vsqrt_`, `vsqrtf_`
Exponential and related functions	`vexp_`, `vexpf_`, `vlog_`, `vlogf_`, `vpow_`, `vpowf_`
Trigonometric functions	`vatan_`, `vatanf_`, `vatan2_`, `vatan2f_`, `vcos_`, `vcosf_`, `vsin_`, `vsinf_`, `vsincos_`, `vsincosf_`
Complex functions	`vc_abs_`, `vc_exp_`, `vc_log_`, `vc_pow_`, `vz_abs_`, `vz_exp_`, `vz_log_`, `vz_pow_`

3.2 Oracle Developer Studio Math Libraries

This section describes the math libraries that are included with the Oracle Developer Studio compilers.

The default base installation directory for Oracle Developer Studio 12.5 is `/opt/developerstudio12.5` on Oracle Solaris and `/opt/oracle/developerstudio12.5` on Linux. However, you can specify a different base installation directory *install-dir* during installation.

Oracle Developer Studio static 32-bit libraries are installed by default in the directory *install-dir*`/lib/compilers` and its subdirectories. Static 64-bit libraries are installed in *install-dir*`/lib/compilers/sparcv9` on SPARC and *install-dir*`/lib/compilers/amd64` on x86.

This *Numerical Computation Guide* also describes the shared `libm` and `libmvec` libraries, which are available for Oracle Solaris only and are installed in `/usr/lib` for 32-bit versions and `/usr/lib/64` for 64-bit versions.

The `math.h` and `sunmath.h` library header files are installed in `/usr/include` for Oracle Solaris. The `math.h` library header file is installed in *install-dir*`/lib/compilers/include/cc` for Linux.

The *subdirectories* of your Oracle Developer Studio *install-dir* for static archives, shared objects, and include files are subject to change from release to release.

3.2.1 Oracle Math Library

The `libsunmath` math library contains functions that are not specified by any standard but are useful in numerical software. It also contains many of the functions that are in `libm.so.2` but not in `libm.so.1`. `libsunmath` is provided as both a shared object and a static archive.

Table 16, “Contents of `libsunmath`,” on page 46 lists the functions in `libsunmath` that are not in `libm.so.2`. For each mathematical function, the table gives only the name of the double precision version of the function as it would be called from a C program.

TABLE 16 Contents of `libsunmath`

Type	Function Name
Elementary transcendental functions	`exp10`
Trigonometric functions in degrees	`asind`, `acosd`, `atand`, `atan2d`, `sind`, `cosd`, `sincosd`, `tand`
Trigonometric functions scaled in `??`	`asinpi`, `acospi`, `atanpi`, `atan2pi`, `sinpi`, `cospi`, `sincospi`, `tanpi`
Trigonometric functions with double precision `??` argument reduction	`asinp`, `acosp`, `atanp`, `sinp`, `cosp`, `sincosp`, `tanp`
Financial functions	`annuity`, `compound`
Integral rounding functions	`aint`, `anint`, `irint`, `nint`
IEEE standard recommended functions	`signbit`

Type	Function Name
IEEE classification functions	`fp_class`, `isinf`, `isnormal`, `issubnormal`, `iszero`
Functions that supply useful IEEE values	`min_subnormal`, `max_subnormal`, `min_normal`, `max_normal`, `infinity`, `signaling_nan`, `quiet_nan`
Additive random number generators	`i_addran_`, `i_addrans_`, `i_init_addrans_`, `i_get_addrans_`, `i_set_addrans_`, `r_addran_`, `r_addrans_`, `r_init_addrans_`, `r_get_addrans_`, `r_set_addrans_`, `d_addran_`, `d_addrans_`, `d_init_addrans_`, `d_get_addrans_`, `d_set_addrans_`, `u_addrans_`
Linear congruential random number generators	`i_lcran_`, `i_lcrans_`, `i_init_lcrans_`, `i_get_lcrans_`, `i_set_lcrans_`, `r_lcran_`, `r_lcrans_`, `d_lcran_`, `d_lcrans_`, `u_lcrans_`
Multiply-with-carry random number generators	`i_mwcran_`, `i_mwcrans_`, `i_init_mwcrans_`, `i_get_mwcrans_`, `i_set_mwcrans`, `i_lmwcran_`, `i_lmwcrans_`, `i_llmwcran_`, `i_llmwcrans_`, `u_mwcran_`, `u_mwcrans_`, `u_lmwcran_`, `u_lmwcrans`, `u_llmwcran_`, `u_llmwcrans_`, `r_mwcran_`, `r_mwcrans_`, `d_mwcran_`, `d_mwcrans_`, `smwcran_`
Random number shufflers	`i_shufrans_`, `r_shufrans_`, `d_shufrans_`, `u_shufrans_`
Data conversion	`convert_external`
Control rounding mode and floating-point exception flags	`ieee_flags`
Floating-point trap handling	`ieee_handler`, `sigfpe`
Show status	`ieee_retrospective`
Enable/disable nonstandard arithmetic	`standard_arithmetic`, `nonstandard_arithmetic`

3.2.2 Optimized Libraries

The `libmopt` library provides faster versions of some of the functions in `libm` and `libsunmath`. `libmopt` is provided as a static archive only. The routines contained in `libmopt` replace corresponding routines in `libm`. Typically, the `libmopt` versions are noticeably faster. Unlike the `libm` versions, however, which support any of ANSI/POSIX®, SVID, X/Open, or C99/IEEE-style treatment of exceptional cases, the `libmopt` routines only support C99/IEEE-style handling of these cases. (See Appendix E, “Standards Compliance”.) Also, while all mathematical functions in `libm` deliver results with reasonable accuracy regardless of the floating-point rounding direction mode, the result of calling any function in `libmopt` with a rounding direction other than round-to-nearest is undefined. A program that uses `libmopt` must ensure that the default round-to-nearest mode is in effect whenever any standard math function is called. To link a program with `libmopt`, use the `-xlibmopt` flag.

3.3 Single, Double, and Extended/Quadruple Precision

Most numerical functions are available in single, double, and extended (x86) or quadruple precision. Examples of calling different precision versions of various functions from different languages are shown in Table 17, “Calling Single, Double, and Extended/Quadruple Functions,” on page 48.

TABLE 17 Calling Single, Double, and Extended/Quadruple Functions

Language	Single	Double	Extended/Quadruple
C, C++	`#include <math.h> float x, y,z; x = sinf(y); x = fmodf (y,z); #include <sunmath.h> float x; x = max_normalf(); x = r_addran_();`	`#include <math.h> double x,y, z; x = sin(y); x = fmod(y,z); #include <sunmath.h> double x; x = max_normal(); x = d_addran_();`	`#include <math.h> long double x, y,z; x = sinl(y); x = fmodl(y, z); #include <sunmath.h> long double x; x = max_normall();`
Fortran	`REAL x,y,z x = sin(y) x = r_fmod(y,z) x = r_max_normal() x = r_addran()`	`REAL*8 x,y,z x = sin(y) x = d_fmod(y,z) x = d_max_normal() x = d_addran()`	`REAL*16 x,y,z x = sin(y) x = q_fmod(y,z) x = q_max_normal()`

In C, names of single precision functions are formed by appending `f` to the double precision name, and names of extended or quadruple precision functions are formed by adding `l`. Because Fortran calling conventions differ, `libsunmath` provides `r_…`, `d_…`, and `q_…` functions for single, double, and quadruple precision, respectively. Fortran intrinsic functions can be called by the generic name for all three precisions.

Not all functions have `q_…` versions. Refer to `math.h` and `sunmath.h` for names and definitions of `libm` and `libsunmath` functions.

In Fortran programs, remember to declare `r_…` functions as `real`, `d_…` functions as double precision, and `q_…` functions as `REAL*16`. Otherwise, type mismatches might result.

Note - Oracle Developer Studio Fortran does not support extended double precision.

3.4 IEEE Support Functions

This section describes the IEEE recommended functions, the functions that supply useful values, `ieee_flags`, `ieee_retrospective`, and `standard_arithmetic` and `nonstandard_arithmetic`. Refer to Chapter 4, “Exceptions and Exception Handling” for more information on the functions `ieee_flags` and `ieee_handler`.

3.4.1 `ieee_functions`(3m) and `ieee_sun`(3m)

The functions described by `ieee_functions`(3m) and `ieee_sun`(3m) provide capabilities either required by the IEEE standard or recommended in its appendix. These are implemented as efficient bit mask operations.

TABLE 18 `ieee_functions`(3m)

Function	Description
`math.h`	Header file
`copysign(x,y)`	x with y's sign bit
`fabs(x)`	Absolute value of x
`fmod(x, y)`	Remainder of x with respect to y
`ilogb(x)`	Base 2 unbiased exponent of x in integer format
`nextafter(x,y)`	Next representable number after x, in the direction y
`remainder(x,y)`	Remainder of x with respect to y
`scalbn(x,n)`	$x \times 2^n$

TABLE 19 `ieee_sun`(3m)

Function	Description
`sunmath.h`	Header file
`fp_class(x)`	Classification function
`isinf(x)`	Classification function
`isnormal(x)`	Classification function
`issubnormal(x)`	Classification function
`iszero(x)`	Classification function
`signbit(x)`	Classification function
`nonstandard_arithmetic(void)`	Enable nonstandard mode
`standard_arithmetic(void)`	Enable standard mode
`ieee_retrospective(*f)`	n/a

The `remainder(x,y)` is the operation specified in IEEE Standard 754-1985. The difference between `remainder(x,y)` and `fmod(x,y)` is that the sign of the result returned by `remainder (x,y)` might not agree with the sign of either x or y, whereas `fmod(x,y)` always returns a result whose sign agrees with x. Both functions return exact results and do not generate inexact exceptions.

TABLE 20 Calling `ieee_functions` From Fortran

IEEE Function	Single Precision	Double Precision	Quadruple Precision
`copysign(x,y)`	`t=r_copysign(x,y)`	`z=d_copysign(x,y)`	`z=q_copysign(x,y)`
`ilogb(x)`	`i=ir_ilogb(x)`	`i=id_ilogb(x)`	`i=iq_ilogb(x)`
`nextafter(x,y)`	`t=r_nextafter(x,y)`	`z=d_nextafter(x,y)`	`z=q_nextafter(x,y)`
`scalbn(x,n)`	`t=r_scalbn(x,n)`	`z=d_scalbn(x,n)`	`z=q_scalbn(x,n)`
`signbit(x)`	`i=ir_signbit(x)`	`i=id_signbit(x)`	`i=iq_signbit(x)`

TABLE 21 Calling `ieee_sun` From Fortran

IEEE Function	Single Precision	Double Precision	Quadruple Precision
`signbit(x)`	`i=ir_signbit(x)`	`i=id_signbit(x)`	`i=iq_signbit(x)`

Note - You must declare `d_`*function* as double precision and `q_`*function* as `REAL*16` in the Fortran program that uses them.

3.4.2 `ieee_values`(3m)

IEEE values like infinity, NaN, maximum and minimum positive floating-point numbers are provided by the functions described by the `ieee_values`(3m) man page. Table 22, “IEEE Values: Single Precision,” on page 50, Table 23, “IEEE Values: Double Precision,” on page 51, Table 24, “IEEE Values: Quadruple Precision,” on page 51, and Table 25, “IEEE Values: Double-Extended Precision (x86),” on page 52 show the decimal values and hexadecimal IEEE representations of the values provided by `ieee_values`(3m) functions.

TABLE 22 IEEE Values: Single Precision

IEEE value	Decimal value hexadecimal representation	C, C++ Fortran
max normal	`3.40282347e+38 7f7fffff`	`r = max_normalf(); r = r_max_normal()`
min normal	`1.17549435e–38 00800000`	`r = min_normalf(); r = r_min_normal()`
max subnormal	`1.17549421e–38 007fffff`	`r = max_subnormalf(); r = r_max_subnormal()`
min subnormal	`1.40129846e–45 00000001`	`r = min_subnormalf(); r = r_min_subnormal()`
∞	`Infinity 7f800000`	`r = infinityf(); r = r_infinity()`
quiet NaN	`NaN 7fffffff`	`r = quiet_nanf(0); r = r_quiet_nan(0)`

IEEE value	Decimal value hexadecimal representation	C, C++ Fortran
signaling NaN	`NaN 7f800001`	`r = signaling_nanf(0); r = r_signaling_nan(0)`

TABLE 23 IEEE Values: Double Precision

IEEE value	Decimal Value hexadecimal representation	C, C++ Fortran
max normal	1.7976931348623157e+308 `7fefffff ffffffff`	`d = max_normal(); d = d_max_normal()`
min normal	2.2250738585072014e–308 `00100000 00000000`	`d = min_normal(); d = d_min_normal()`
max subnormal	2.2250738585072009e–308 `000fffff ffffffff`	`d = max_subnormal(); d = d_max_subnormal()`
min subnormal	4.9406564584124654e–324 `00000000 00000001`	`d = min_subnormal(); d = d_min_subnormal()`
∞	`Infinity` `7ff00000 00000000`	`d = infinity(); d = d_infinity()`
quiet NaN	`NaN` `7fffffff ffffffff`	`d = quiet_nan(0); d = d_quiet_nan (0)`
signaling NaN	`NaN` `7ff00000 00000001`	`d = signaling_nan(0); d = d_signaling_nan(0)`

TABLE 24 IEEE Values: Quadruple Precision

IEEE value	Decimal value hexadecimal representation	C, C++ (SPARC) Fortran (all)
max normal	`1.189731495357231765085759326628007 0e+4932` `7ffeffff ffffffff ffffffff ffffffff`	`q = max_normall(); q = q_max_normal()`
min normal	`3.362103143112093506262677817321752 6e–4932` `00010000 00000000 00000000 00000000`	`q = min_normall(); q = q_min_normal()`
max subnormal	`3.362103143112093506262677817321752 0e–4932` `0000ffff ffffffff ffffffff ffffffff`	`q = max_subnormall(); q = q_max_subnormal()`
min subnormal	`6.475175119438025110924438958227646 6e–4966` `00000000 00000000 00000000 00000001`	`q = min_subnormall(); q = q_min_subnormal()`
∞	`Infinity`	`q = infinityl(); q = q_infinity()`

IEEE value	Decimal value hexadecimal representation	C, C++ (SPARC) Fortran (all)

quiet NaN	`7fff0000 00000000 00000000 00000000`
	`NaN`
	`7fff8000 00000000 00000000 00000000`
		`q = quiet_nanl(0); q = q_quiet_nan(0)`

signaling NaN	`NaN` `7fff0000 00000000 00000000 00000001`	`q = signaling_nanl(0); q = q_signaling_nan(0)`

TABLE 25 IEEE Values: Double-Extended Precision (x86)

IEEE value	Decimal value hexadecimal representation (80 bits)	C, C++
max normal	`1.18973149535723176505e+4932` `7ffe ffffffff ffffffff`	`x = max_normall();`
min positive normal	`3.3621031431120935062 6e–4932` `0001 80000000 00000000`	`x = min_normall();`
max subnormal	`3.3621031431120935060 8e–4932` `0000 7fffffff ffffffff`	`x = max_subnormall();`
min positive subnormal	`1.8225997659412373012 6e–4951` `0000 00000000 00000001`	`x = min_subnormall();`
∞	`Infinity` `7fff 80000000 00000000`	`x = infinityl();`
quiet NaN	`NaN` `7fff c0000000 00000000`	`x = q`
signaling NaN	`NaN` `7fff 80000000 00000001`	`x = signaling_nanl(0);`

3.4.3 `ieee_flags`(3m)

`ieee_flags` (3m) is the Oracle interface to:

- Query or set rounding direction mode
- Query or set rounding precision mode
- Examine, clear, or set accrued exception flags

The syntax for a call to `ieee_flags`(3m) is:

```
i = ieee_flags(action, mode, in, out);
```

The ASCII strings that are the possible values for the parameters are shown in Table 26, "Parameter Values for `ieee_flags`," on page 53:

TABLE 26 Parameter Values for `ieee_flags`

Parameter	C or C++ Type	All Possible Values
`action`	`char *`	`get`, `set`, `clear`, `clearall`
`mode`	`char *`	`direction`, `precision`, `exception`
`in`	`char *`	`nearest`, `tozero`, `negative`, `positive`, `extended`, `double`, `single`, `inexact`, `division`, `underflow`, `overflow`, `invalid`, `all`, `common`
`out`	`char **`	`nearest`, `tozero`, `negative`, `positive`, `extended`, `double`, `single`, `inexact`, `division`, `underflow`, `overflow`, `invalid`, `all`, `common`

The *ieee_flags*(3m) man page describes the parameters in complete detail.

Some of the arithmetic features that can be modified by using `ieee_flags` are covered in the following paragraphs. Chapter 4 contains more information on `ieee_flags` and IEEE exception flags.

When *mode* is `direction`, the specified action applies to the current rounding direction. The possible rounding directions are: round towards nearest, round towards zero, round towards +•, or round towards -•. The IEEE default rounding direction is *round towards nearest*. This means that when the mathematical result of an operation lies strictly between two adjacent representable numbers, the one nearest to the mathematical result is delivered. (If the mathematical result lies exactly halfway between the two nearest representable numbers, then the result delivered is the one whose least significant bit is zero. The *round towards nearest* mode is sometimes called *round to nearest even* to emphasize this.)

Rounding towards zero is the way many pre-IEEE computers work, and corresponds mathematically to truncating the result. For example, if 2/3 is rounded to 6 decimal digits, the result is .666667 when the rounding mode is round towards nearest, but .666666 when the rounding mode is round towards zero.

When using `ieee_flags` to examine, clear, or set the rounding direction, possible values for the four input parameters are shown in Table 27, "`ieee_flags` Input Values for the Rounding Direction," on page 53.

TABLE 27 `ieee_flags` Input Values for the Rounding Direction

Parameter	Possible value (mode is `direction`)
`action`	`get`, `set`, `clear`, `clearall`

Parameter	Possible value (mode is `direction`)
`in`	`nearest`, `tozero`, `negative`, `positive`
`out`	`nearest`, `tozero`, `negative`, `positive`

When *mode* is `precision`, the specified action applies to the current rounding precision. On x86-based systems, the possible rounding precisions are: single, double, and extended. The default rounding precision is extended; in this mode, arithmetic operations that deliver a result to an x87 floating-point register round their result to the full 64-bit precision of the extended double register format. When the rounding precision is single or double, arithmetic operations that deliver a result to an x87 floating-point register round their result to 24 or 53 significant bits, respectively. Although most programs produce results that are at least as accurate, if not more so, when extended rounding precision is used, some programs that require strict adherence to the semantics of IEEE arithmetic will not work correctly in extended rounding precision mode and must be run with the rounding precision set to single or double as appropriate.

Rounding precision cannot be set on systems using SPARC processors. On these systems, calling `ieee_flags` with *mode* = `precision` has no effect on computation.

Finally, when *mode* is `exception`, the specified action applies to the current IEEE exception flags. See Chapter 4, "Exceptions and Exception Handling" for more information about using `ieee_flags` to examine and control the IEEE exception flags.

3.4.4 `ieee_retrospective`(3m)

The `libsunmath` function `ieee_retrospective` prints information about unrequited exceptions and nonstandard IEEE modes. It reports:

- Outstanding exceptions.
- Enabled traps.
- If rounding direction or precision is set to other than the default.
- If nonstandard arithmetic is in effect.

The necessary information is obtained from the hardware floating-point status register.

`ieee_retrospective` prints information about exception flags that are raised, and exceptions for which a *trap* is enabled. These two distinct, if related, pieces of information should not be confused. If an exception flag is raised, then that exception occurred at some point during program execution. If a trap is enabled for an exception, then the exception might not have actually occurred, but if it had, a SIGFPE signal would have been delivered. The `ieee_retrospective` message is meant to alert you about exceptions that might need to be investigated, if the exception flag is *raised*, or to remind you that exceptions might have been

handled by a signal handler, if the exception's *trap* is enabled. Chapter 4, "Exceptions and Exception Handling" discusses exceptions, signals, and traps, and shows how to investigate the cause of a raised exception.

A program can explicitly call `ieee_retrospective` at any time. Fortran programs compiled with `f95` in `-f77` compatibility mode automatically call `ieee_retrospective` before they exit. C/C++ programs and Fortran programs compiled with `f95` in the default mode do not automatically call `ieee_retrospective`.

Note, though, that the `f95` compiler enables trapping on common exceptions by default, so unless a program either explicitly disables trapping or installs a `SIGFPE` handler, it will immediately abort when such an exception occurs. In `-f77` compatibility mode, the compiler does not enable trapping, so when floating-point exceptions occur, the program continues execution and reports those exceptions via the `ieee_retrospective` output on exit.

The syntax for calling this function is as follows:

- C, C++ - `ieee_retrospective(`*fp*`);`
- Fortran - `call ieee_retrospective()`

For the C function, the argument *fp* specifies the file to which the output will be written. The Fortran function always prints output on `stderr`.

The following example shows four of the six `ieee_retrospective` warning messages:

```
Note: IEEE floating-point exception flags raised:
   Inexact; Underflow;
Rounding direction toward zero
IEEE floating-point exception traps enabled:
   overflow;
See the Numerical Computation Guide, ieee_flags(3M),
ieee_handler(3M), ieee_sun(3m)
```

A warning message appears only if trapping is enabled or an exception was raised.

You can suppress `ieee_retrospective` messages from Fortran programs by one of three methods. One approach is to clear all outstanding exceptions, disable traps, and restore round-to-nearest, extended precision, and standard modes before the program exits. To do this, call `ieee_flags`, `ieee_handler`, and `standard_arithmetic` as follows:

```
character*8 out
i = ieee_flags('clearall', '', '', out)
call ieee_handler('clear', 'all', 0)
call standard_arithmetic()
```

Note - Clearing outstanding exceptions without investigating their cause is not recommended.

Another way to avoid seeing `ieee_retrospective` messages is to redirect `stderr` to a file. Of course, this method should not be used if the program sends output other than `ieee_retrospective` messages to `stderr`.

The third approach is to include a dummy `ieee_retrospective` function in the program, for example:

```
subroutine ieee_retrospective
return
end
```

3.4.5 `nonstandard_arithmetic`(3m)

As discussed in Chapter 2, "IEEE Arithmetic", IEEE arithmetic handles underflowed results using gradual underflow. On some SPARC-based systems, gradual underflow is often implemented partly with software emulation of the arithmetic. If many calculations underflow, this can cause performance degradation.

To obtain some information about whether this is a case in a specific program, you can use `ieee_retrospective` or `ieee_flags` to determine if underflow exceptions occur, and check the amount of system time used by the program. If a program spends an unusually large amount of time in the operating system, and raises underflow exceptions, gradual underflow might be the cause. In this case, using non-IEEE arithmetic might speed up program execution.

The function `nonstandard_arithmetic` enables non-IEEE arithmetic modes on processors that support them. On SPARC systems, this function sets the NS (nonstandard arithmetic) bit in the floating-point status register. On x86 systems supporting the SSE instructions, this function sets the FTZ (flush to zero) bit in the MXCSR register; it also sets the DAZ (denormals are zero) bit in the MXCSR register on those processors that support this bit. Note that the effects of nonstandard modes vary from one processor to another and can cause otherwise robust software to malfunction. Nonstandard mode is not recommended for normal use.

The function `standard_arithmetic` resets the hardware to use the default IEEE arithmetic. Both functions have no effect on processors that provide only the default IEEE 754 style of arithmetic. SPARC T4 is one such processor.

3.5 C99 Floating-Point Environment Functions

This section describes the <`fenv.h`> floating-point environment functions in C99. In the Oracle Solaris 10 OS, these functions are available in `libm`. They provide many of the same

capabilities as the `ieee_flags` function, but they use a more natural C interface, and because they are defined by C99, they are more portable.

Note - For consistent behavior, do not use both C99 floating-point environment functions and exception handling extensions in `libm` and the `ieee_flags` and `ieee_handler` functions in `libsunmath` in the same program.

3.5.1 Exception Flag Functions

The `fenv.h` file defines macros for each of the five IEEE floating-point exception flags: `FE_INEXACT`, `FE_UNDERFLOW`, `FE_OVERFLOW`, `FE_DIVBYZERO`, and `FE_INVALID`. In addition, the macro `FE_ALL_EXCEPT` is defined to be the bitwise "or" of all five flag macros. In the following descriptions, the excepts parameter might be a bitwise "or" of any of the five flag macros or the value `FE_ALL_EXCEPT`. For the `fegetexceptflag` and `fesetexceptflag` functions, the *flagp* parameter must be a pointer to an object of type `fexcept_t`. This type is defined in `fenv.h`.

C99 defines the exception flag functions in the following table:

TABLE 28 C99 Standard Exception Flag Functions

Function	Action
`feclearexcept`(*excepts*)	clear specified flags
`fetestexcept`(*excepts*)	return settings of specified flags
`feraiseexcept`(*excepts*)	raise specified exceptions
`fegetexceptflag`(*flagp*, *excepts*)	save specified flags in *flagp
`fesetexceptflag`(*flagp*, *excepts*)	restore specified flags from *flagp

The `feclearexcept` function clears the specified flags. The `fetestexcept` function returns a bitwise "or" of the macro values corresponding to the subset of flags specified by the excepts argument that are set. For example, if the only flags currently set are inexact, underflow, and division by zero, then the following would set `i` to `FE_DIVBYZERO`.

```
i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
```

The `feraiseexcept` function causes a trap if any of the specified exceptions' trap is enabled. Otherwise, it merely sets the corresponding flags. See Chapter 4, "Exceptions and Exception Handling" for more information on exception traps.

The `fegetexceptflag` and `fesetexceptflag` functions provide a convenient way to temporarily save the state of certain flags and later restore them. In particular, the

`fesetexceptflag` function does not cause a trap; it merely restores the values of the specified flags.

3.5.2 Rounding Control

The `fenv.h` file defines macros for each of the four IEEE rounding direction modes: `FE_TONEAREST`, `FE_UPWARD` (toward positive infinity), `FE_DOWNWARD` (toward negative infinity), and `FE_TOWARDZERO`. C99 defines two functions to control rounding direction modes: `fesetround` sets the current rounding direction to the direction specified by its argument (which must be one of the four macros above), and `fegetround` returns the value of the macro corresponding to the current rounding direction.

On x86-based systems, the `fenv.h` file defines macros for each of three rounding precision modes: `FE_FLTPREC` (single precision), `FE_DBLPREC` (double precision), and `FE_LDBLPREC` (extended double precision). Although they are not part of C99, `libm` on x86 provides two functions to control the rounding precision mode: `fesetprec` sets the current rounding precision to the precision specified by its argument which must be one of the three macros above, and `fegetprec` returns the value of the macro corresponding to the current rounding precision.

3.5.3 Environment Functions

The `fenv.h` file defines the data type `fenv_t`, which represents the entire floating-point environment including exception flags, rounding control modes, exception handling modes, and, on SPARC, nonstandard mode. In the descriptions that follow, the *envp* parameter must be a pointer to an object of type `fenv_t`.

C99 defines four functions to manipulate the floating-point environment. `libm` provides an additional function that can be useful in multi-threaded programs. These functions are summarized in the following table:

TABLE 29 `libm` Floating-Point Environment Functions

Function	Action
`fegetenv`(*envp*)	save environment in `*envp`
`fesetenv`(*envp*)	restore environment from `*envp`
`feholdexcept`(*envp*)	save environment in `*envp` and establish nonstop mode
`feupdateenv`(*envp*)	restore environment from `*envp` and raise exceptions
`fex_merge_flags`(*envp*)	“or” exception flags from `*envp`

The fegetenv and fesetenv functions respectively save and restore the floating-point environment. The argument to fesetenv can be either a pointer to an environment previously saved by a call to fegetenv or feholdexcept or the constant FE_DFL_ENV defined in fenv.h. The latter represents the default environment with all exception flags clear, rounding to nearest and to extended double precision on x86-based systems, nonstop exception handling mode (i.e., traps disabled), and nonstandard mode disabled.

The feholdexcept function saves the current environment and then clears all exception flags and establishes nonstop exception handling mode for all exceptions. The feupdateenv function restores a saved environment (which might be one saved by a call to fegetenv or feholdexcept or the constant FE_DFL_ENV), then raises those exceptions whose flags were set in the previous environment. If the restored environment has traps enabled for any of those exceptions, a trap occurs; otherwise the flags are set. These two functions can be used in conjunction to make a subroutine call appear to be atomic with regard to exceptions, as the following code sample shows:

```
#include <fenv.h>

void myfunc(...) {
    fenv_t env;

    /* save the environment, clear flags, and disable traps */
    feholdexcept(&env);
    /* do a computation that may incur exceptions */
    ...
    /* check for spurious exceptions */
    if (fetestexcept(...)) {
        /* handle them appropriately and clear their flags */
        ...
        feclearexcept(...);
    }
    /* restore the environment and raise relevant exceptions */
    feupdateenv(&env);
}
```

The fex_merge_flags function performs a logical OR of the exception flags from the saved environment into the current environment without provoking any traps. This function can be used in a multi-threaded program to preserve information in the parent thread about flags that were raised by a computation in a child thread. See Appendix A, “Examples” for an example showing the use of fex_merge_flags.

3.6 Implementation Features of libm and libsunmath

This section describes implementation features of libm and libsunmath:

- Argument reduction using infinitely precise ??, and trigonometric functions scaled in ??
- Data conversion routines for converting floating-point data between IEEE and non-IEEE formats
- Random number generators

3.6.1 About the Algorithms

The elementary functions in libm and libsunmath on SPARC-based systems are implemented with table-driven and polynomial/rational approximation algorithms. These algorithms are subject to change between releases for better performance or accuracy. Some elementary functions in libm and libsunmath on x86-based systems are implemented using the elementary function kernel instructions provided in the x86 instruction set; other functions are implemented using the same table-driven or polynomial/rational approximation algorithms used on SPARC-based systems.

Both the table-driven and polynomial/rational approximation algorithms for the common elementary functions in libm and the common single precision elementary functions in libsunmath deliver results that are accurate to within one unit in the last place (*ulp*). On SPARC-based systems, the common quadruple precision elementary functions in libsunmath deliver results that are accurate to within one *ulp*, except for the expm1l and log1pl functions, which deliver results accurate to within two *ulps*. (The common functions include the exponential, logarithm, and power functions and circular trigonometric functions of radian arguments. Other functions, such as the hyperbolic trig functions and higher transcendental functions, are less accurate.) These error bounds have been obtained by direct analysis of the algorithms. Users can also test the accuracy of these routines using BeEF, the Berkeley Elementary Function test programs, available from netlib in the ucbtest package http://www.netlib.org/fp/ucbtest.tgz.

3.6.2 Argument Reduction for Trigonometric Functions

Trigonometric functions for radian arguments outside the range [–??/4,??/4] are usually computed by reducing the argument to the indicated range by subtracting integral multiples of ??/2.

Because ?? is not a machine-representable number, it must be approximated. The error in the final computed trigonometric function depends on the rounding errors in argument reduction with an approximate ?? as well as the rounding, and approximation errors in computing the trigonometric function of the reduced argument. Even for fairly small arguments, the relative error in the final result might be dominated by the argument reduction error, while even for

fairly large arguments, the error due to argument reduction might be no worse than the other errors.

There is widespread misapprehension that trigonometric functions of all large arguments are inherently inaccurate, and all small arguments relatively accurate. This is based on the simple observation that large enough machine-representable numbers are separated by a distance greater than ??.

There is no inherent boundary at which computed trigonometric function values suddenly become bad, nor are the inaccurate function values useless. Provided that the argument reduction be done consistently, the fact that the argument reduction is performed with an approximation to ?? is practically undetectable, because all essential identities and relationships are as well preserved for large arguments as for small.

libm and libsunmath trigonometric functions use an “infinitely” precise ?? for argument reduction. The value 2/?? is computed to 916 hexadecimal digits and stored in a lookup table to use during argument reduction.

The group of functions sinpi, cospi, and tanpi (see Table 16, “Contents of libsunmath,” on page 46) scales the input argument by ?? to avoid inaccuracies introduced by range reduction.

3.6.3 Data Conversion Routines

In libm and libsunmath, there is a flexible data conversion routine, convert_external, used to convert binary floating-point data between IEEE and non-IEEE formats.

Formats supported include those used by SPARC (IEEE), IBM PC, VAX, IBM S/370, and Cray.

Refer to the man page on *convert_external*(3m) for an example of taking data generated on a Cray, and using the function convert_external to convert the data into the IEEE format expected on SPARC-based systems.

3.6.4 Random Number Facilities

There are three facilities for generating uniform pseudo-random numbers in 32-bit integer, single precision floating-point, and double precision floating-point formats:

- The functions described in the *addrans*(3m) manual page are based on a family of table-driven additive random number generators.

- The functions described in the *lcrans*(3m) manual page are based on a linear congruential random number generator.
- The functions described in the *mwcrans*(3m) manual page are based on multiply-with-carry random number generators. These functions also include generators that supply uniform pseudo-random numbers in 64-bit integer formats.

In addition, the functions described on the *shufrans*(3m) manual page can be used in conjunction with any of these generators to shuffle an array of pseudo-random numbers, thereby providing even more randomness for applications that need it. Note that there is no facility for shuffling arrays of 64-bit integers.

Each of the random number facilities includes routines that generate one random number at a time, i.e., one per function call, as well as routines that generate an array of random numbers in a single call. The functions that generate one random number at a time deliver numbers that lie in the ranges shown in Table 30, “Intervals for Single-Value Random Number Generators,” on page 62.

TABLE 30 Intervals for Single-Value Random Number Generators

Function	Lower Bound	Upper Bound
i_addran_	-2147483648	2147483647
r_addran_	0	0.9999999403953552246
d_addran_	0	0.9999999999999998890
i_lcran_	1	2147483646
r_lcran_	4.656612873077392578E-10	1
d_lcran_	4.656612875245796923E-10	0.9999999995343387127
i_mwcran_	0	2147483647
u_mwcran_	0	4294967295
i_llmwcran_	0	9223372036854775807
u_llmwcran_	0	18446744073709551615
r_mwcran_	0	0.9999999403953552246
d_mwcran_	0	0.9999999999999998890

The functions that generate an entire array of random numbers in a single call allow the user to specify the interval in which the generated numbers will lie. Appendix A, “Examples” gives several examples that show how to generate arrays of random numbers uniformly distributed over different intervals.

Note that the addrans and mwcrans generators are generally more efficient than the lcrans generators, but their theory is not as refined. “Random Number Generators: Good Ones Are Hard To Find”, by S. Park and K. Miller, *Communications of the ACM*, October 1988, discusses

the theoretical properties of linear congruential algorithms. Additive random number generators are discussed in Volume 2 of Knuth's *The Art of Computer Programming*.

♦ ♦ ♦ CHAPTER 4

Exceptions and Exception Handling

This chapter describes IEEE floating-point exceptions and shows how to detect, locate, and handle them. This chapter also lists the exceptions defined by IEEE 754 along with their default results and describes the features of the floating-point environment that support status flags, trapping, and exception handling. This chapter divides these topics into the following sections:

4.1 Exception Handling Objectives

The floating-point environment provided by the Oracle Developer Studiocompilers and the Oracle Solaris OS on SPARC-based systems and x86-based systems supports all of the exception handling facilities required by the IEEE standard as well as many of the recommended optional facilities. One objective of these facilities is explained in the IEEE 754 Standard (IEEE 854, page 18) as follows:

> ... to minimize for users the complications arising from exceptional conditions. The arithmetic system is intended to continue to function on a computation as long as possible, handling unusual situations with reasonable default responses, including setting appropriate flags.

To achieve this objective, the standards specify default results for exceptional operations and require that an implementation provide status flags, which can be sensed, set, or cleared by a user, to indicate that exceptions have occurred. The standards also recommend that an implementation provide a means for a program to trap (i.e., interrupt normal control flow) when an exception occurs. The program can optionally supply a trap handler that handles the exception in an appropriate manner, for example by providing an alternate result for the

exceptional operation and resuming execution. The following sections describe in more detail about how the features of the floating-point environment supports these exceptions.

4.2 What Is an Exception?

It is hard to define exceptions. To quote W. Kahan,

> An arithmetic exception arises when an attempted atomic arithmetic operation has no result that would be acceptable universally. The meanings of atomic and acceptable vary with time and place. (See *Handling Arithmetic Exceptions* by W. Kahan.)

For example, an exception arises when a program attempts to take the square root of a negative number. This example is one case of an *invalid operation* exception. When such an exception occurs, the system responds in one of two ways:

- If the exception's trap is disabled (the default case), the system records the fact that the exception occurred and continues executing the program using the default result specified by IEEE 754 for the excepting operation.
- If the exception's trap is enabled, the system generates a `SIGFPE` signal. If the program has installed a `SIGFPE` signal handler, the system transfers control to that handler; otherwise, the program aborts.

IEEE 754 defines five basic types of floating-point exceptions: *invalid operation*, *division by zero*, *overflow*, *underflow* and *inexact*. The first three (invalid, division, and overflow) are sometimes collectively called *common exceptions*. These exceptions can seldom be ignored when they occur. The *ieee_handler*(3m) man page explains an easy way to trap on common exceptions only. The other two exceptions (underflow and inexact) are seen more often. In fact, most floating-point operations incur the inexact exception. These exceptions can usually, though not always, be safely ignored. Oracle Developer Studio 12.5 C, C++, and `f77` compilers disable all IEEE traps by default. The `f95` compiler enables traps for the common exceptions by default. 754-standard conforming can be restored by compiling with `f95 -ftrap=none`.

Table 31, “IEEE Floating-Point Exceptions,” on page 66 condenses information found in IEEE Standard 754. It describes the five floating-point exceptions and the default response of an IEEE arithmetic environment when these exceptions are raised.

TABLE 31 IEEE Floating-Point Exceptions

IEEE Exception	Reason Why This Arises	Example	Default Result When Trap is Disabled
Invalid operation	An operand is invalid for the operation about to be performed.	■ $0 \times \infty$	Quiet `NaN`

IEEE Exception	Reason Why This Arises	Example	Default Result When Trap is Disabled
	(On x86, this exception is also raised when the floating-point stack underflows or overflows, though that is not part of the IEEE standard.)	■ 0 / 0 ■ ∞ / ∞ ■ x REM 0 ■ Square root of negative operand ■ Any operation with a signaling NaN operand ■ Unordered comparison (see note 1) ■ Invalid conversion (see note 2)	
Division by zero	An exact infinite result is produced by an operation on finite operands.	■ x / 0 for finite, nonzero x ■ log(0)	Correctly signed infinity
Overflow	The correctly rounded result would be larger in magnitude than the largest finite number representable in the destination format (i.e., the exponent range is exceeded).	■ Double precision: ■ DBL_MAX + 1.0e294 ■ exp(709.8) ■ Single precision: ■ (float)DBL_MAX ■ FLT_MAX + 1.0e32 ■ expf(88.8)	Depends on rounding mode (RM), and the sign of the intermediate result. See item 4 in “4.2.1 Notes for Table 4-1” on page 67.
Underflow	Either the exact result or the correctly rounded result would be smaller in magnitude than the smallest normal number representable in the destination format (see note 3).	■ Double precision: ■ nextafter(min_normal,-•) ■ nextafter(min_subnormal,-•) ■ DBL_MIN §3.0 ■ exp(-708.5) ■ Single precision: ■ (float)DBL_MIN ■ nextafterf(FLT_MIN, -•) ■ expf(-87.4)	Subnormal or zero
Inexact	The rounded result of a valid operation is different from the infinitely precise result. (Most floating-point operations raise this exception.)	■ 2.0 / 3.0 ■ (float)1.12345678 ■ log(1.1) ■ DBL_MAX + DBL_MAX, when no overflow trap	The result of the operation (rounded, overflowed, or underflowed)

4.2.1 Notes for Table 4-1

1. Unordered comparison: Any pair of floating-point values can be compared, even if they are not of the same format. Four mutually exclusive relations are possible: less than, greater than, equal, or unordered. Unordered means that at least one of the operands is a NaN (not a number).

Every NaN compares “unordered” with everything, including itself. The following table shows which predicates cause the invalid operation exception when the relation is unordered.

Math Predicate	C, C++ Predicate	Fortran Predicate	Invalid Expression (If Unordered)
=	==	.EQ.	no
≠	!=	.NE.	no
>	>	.GT.	yes
≥	>=	.GE.	yes
<	<	.LT.	yes
≤	<=	.LE.	yes

2. Invalid conversion: Attempt to convert NaN or infinity to an integer, or integer overflow on conversion from floating-point format.
3. The smallest normal numbers representable in the IEEE single, double, and extended formats are 2-126, 2-1022, and 2-16382, respectively. See Chapter 2, “IEEE Arithmetic” for a description of the IEEE floating-point formats.
4. The following table lists the default result when the trap is disabled for overflow. The below results depend on the rounding mode and the sign of the intermediate result.

Rounding Mode	Positive	Negative
Nearest	+∞	-∞
Zero	+∞	-max
Down	+max	-∞
Up	+∞	-max

The x86 floating-point environment provides another exception not mentioned in the IEEE standards: the *denormal operand* exception. This exception is raised whenever a floating-point operation is performed on a subnormal number.

Exceptions are prioritized in the following order: invalid (highest priority), overflow, division, underflow, inexact (lowest priority). On x86-based systems, the denormal operand exception has the lowest priority of all.

The only combinations of standard exceptions that can occur simultaneously in a single operation are overflow with inexact and underflow with inexact. On x86-based systems, the denormal operand exception can occur with any of the five standard exceptions. If trapping on overflow, underflow, and inexact is enabled, the overflow and underflow traps take precedence over the inexact trap; they all take precedence over a denormal operand trap on x86-based systems.

4.3 Detecting Exceptions

As required by the IEEE standard, the floating-point environments on SPARC-based systems and x86-based systems provide status flags that record the occurrence of floating-point exceptions. A program can test these flags to determine which exceptions have occurred. The flags can also be explicitly set and cleared. The `ieee_flags` function provides one way to access these flags. In programs written in C or C++, the C99 floating-point environment functions provide another.

On SPARC-based systems, each exception has two flags associated with it, current and accrued. The current exception flags always indicate the exceptions raised by the last floating-point instruction to complete execution. These flags are also accumulated (i.e., "or" ‑ ed) into the accrued exception flags thereby providing a record of all untrapped exceptions that have occurred since the program began execution or since the accrued flags were last cleared by the program. When a floating-point instruction incurs a trapped exception, the current exception flag corresponding to the exception that caused the trap is set, but the accrued flags are unchanged. Both the current and accrued exception flags are contained in the floating-point status register, `%fsr`.

On x86-based systems, the floating-point status word (SW) provides flags for accrued exceptions as well as flags for the status of the floating-point stack. On x86-based systems that support SSE2 instructions, the MXCSR register contains flags that record accrued exceptions raised by those instructions.

4.3.1 `ieee_flags`(3m)

`ieee_flags` provides an interface for IEE 754 exception flags that is similar Oracle Developer Studio C, C++, and Fortran. However, this interface is only available on the Oracle Solaris OS. Use the "4.3.2 C99 Exception Flag Functions" on page 71 for C and C++ programs that are intended to be more widely portable.

The syntax for a call to *ieee_flags*(3m) is:

`i = ieee_flags(`*action*, *mode*, *in*, *out*`);`

A program can test, set, or clear the accrued exception status flags using the `ieee_flags` function by supplying the string `"exception"` as the second argument. For example, to clear the overflow exception flag from Fortran, write:

```
character*8 out
call ieee_flags('clear', 'exception', 'overflow', out)
```

To determine whether an exception has occurred from C or C++, use:

```
i = ieee_flags("get", "exception", in, out);
```

When the action is "get", the string returned in *out* is one of the following:

- "not available" — if information on exceptions is not available
- "" (an empty string) — if there are no accrued exceptions or, in the case of x86, the denormal operand is the only accrued exception
- The name of the exception named in the third argument, *in*, if that exception has occurred
- Otherwise, the name of the highest priority exception that has occurred

For example, in the following Fortran call the string returned in out is "division" if the division-by-zero exception has occurred. Otherwise it is the name of the highest priority exception that has occurred:

```
character*8 out
i = ieee_flags('get', 'exception', 'division', out)
```

Note that *in* is ignored unless it names a particular exception. For example, the argument "all" is ignored in the following C call:

```
i = ieee_flags("get", "exception", "all", out);
```

Besides returning the name of an exception in *out*, ieee_flags returns an integer value that combines all of the exception flags currently raised. This value is the bitwise "or" of all the accrued exception flags, where each flag is represented by a single bit as shown in Table 32, "Exception Bits," on page 70. The positions of the bits corresponding to each exception are given by the fp_exception_type values defined in the file sys/ieeefp.h. (Note that these bit positions are machine-dependent and need not be contiguous.)

TABLE 32 Exception Bits

Exception	Bit Position	Accrued Exception Bit
invalid	fp_invalid	i & (1 << fp_invalid)
overflow	fp_overflow	i & (1 << fp_overflow)
division	fp_division	i & (1 << fp_division)
underflow	fp_underflow	i & (1 << fp_underflow)
inexact	fp_inexact	i & (1 << fp_inexact)
denormalized	fp_denormalized	i & (1 << fp_denormalized) *(x86 only)*

This fragment of a C or C++ program shows one way to decode the return value.

```
/*
 *   Decode integer that describes all accrued exceptions.
 *   fp_inexact etc. are defined in <sys/ieeefp.h>
```

```
 */

char *out;
int invalid, division, overflow, underflow, inexact;

code = ieee_flags("get", "exception", "", &out);
printf ("out is %s, code is %d, in hex: 0x%08X\n",
  out, code, code);
inexact  =  (code >> fp_inexact) & 0x1;
division  =  (code >> fp_division) & 0x1;
underflow  =  (code >> fp_underflow) & 0x1;
overflow  =  (code >> fp_overflow) & 0x1;
invalid  =  (code >> fp_invalid) & 0x1;
printf("%d %d %d %d %d \n", invalid, division, overflow,
  underflow, inexact);
```

4.3.2 C99 Exception Flag Functions

C/C++ programs can test, set, and clear the floating-point exception flags using the C99 floating-point environment functions. The header file `fenv.h` defines five macros corresponding to the five standard exceptions: `FE_INEXACT`, `FE_UNDERFLOW`, `FE_OVERFLOW`, `FE_DIVBYZERO`, and `FE_INVALID`. It also defines the macro `FE_ALL_EXCEPT` to be the bitwise “or” of all five exception macros. These macros can be combined to test or clear any subset of the exception flags or raise any combination of exceptions. The following examples show the use of these macros with several of the C99 floating-point environment functions. See the *feclearexcept*(3M) manual page for more information.

Note - For consistent behavior, do not use both the C99 floating-point environment functions and extensions in `libm` and the `ieee_flags` and `ieee_handler` functions in `libsunmath` in the same program.

To clear all five exception flags, use the following:

```
feclearexcept(FE_ALL_EXCEPT);
```

To test whether the invalid operation or division by zero flags have been raised, use the following:

```
int i;

i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
if (i & FE_INVALID)
    /* invalid flag was raised */
else if (i & FE_DIVBYZERO)
    /* division-by-zero flag was raised */
```

The `fegetexceptflag` and `fesetexceptflag` functions provide a way to save and restore a subset of the flags. The next example shows one way to use these functions.

```
fexcept_t flags;

/* save the underflow, overflow, and inexact flags */
fegetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW | FE_INEXACT);
/* clear these flags */
feclearexcept(FE_UNDERFLOW | FE_OVERFLOW | FE_INEXACT);
/* do a computation that can underflow or overflow */
...
/* check for underflow or overflow */
if (fetestexcept(FE_UNDERFLOW | FE_OVERFLOW) != 0) {
    ...
}
/* restore the underflow, overflow, and inexact flags */
fesetexceptflag(&flags, FE_UNDERFLOW | FE_OVERFLOW, | FE_INEXACT);
```

4.4 Locating an Exception

One way to locate where an exception occurs is to test the exception flags at various points throughout a program. However, isolating an exception precisely by this approach can require many tests and carry a significant overhead.

An easier way to determine where an exception occurs is to enable its trap. When an exception whose trap is enabled occurs, the operating system notifies the program by sending a `SIGFPE` signal. See the *signal*(5) manual page. Thus, by enabling trapping for an exception, you can determine where the exception occurs either by running under a debugger and stopping on receipt of a `SIGFPE` signal or by establishing a `SIGFPE` handler that prints the address of the instruction where the exception occurred. Note that trapping must be enabled for an exception to generate a `SIGFPE` signal. When trapping is disabled and an exception occurs, the corresponding flag is set and execution continues with the default result specified in Table 31, “IEEE Floating-Point Exceptions,” on page 66, but no signal is delivered.

4.4.1 Using the Debugger to Locate an Exception

This section gives examples showing how to use `dbx` to investigate the cause of a floating-point exception and locate the instruction that raised it. Recall that in order to use the source-level debugging features of `dbx`, programs should be compiled with the `-g` flag. Refer to the *Oracle Developer Studio 12.5: Debugging a Program with dbx* for more information.

Consider the following C program:

```
#include <stdio.h>
#include <math.h>

double sqrtm1(double x)
{
   return sqrt(x) - 1.0;
}

int main(void)
{
   double x, y;

   x = -4.2;
   y = sqrtm1(x);
   printf("%g  %g\n", x, y);
   return 0;
}
```

Compiling and running this program produces:

```
-4.2  NaN
```

The appearance of a NaN in the output suggests that an invalid operation exception might have occurred. To determine whether this is the case, you can recompile with the `-ftrap` option to enable trapping on invalid operations and use `dbx` to run the program and stop when a `SIGFPE` signal is delivered. Alternatively, you can use `dbx` without recompiling the program by linking with a startup routine that enables the invalid operation trap or by manually enabling the trap.

4.4.1.1 Using `dbx` to Locate the Instruction Causing an Exception

The simplest way to locate the code that causes a floating-point exception is to recompile with the `-g` and `-ftrap` flags and then use `dbx` to track down the location where the exception occurs. First, recompile the program as follows:

```
example% cc -g -ftrap=invalid ex.c -lm
```

Compiling with `-g` allows you to use the source-level debugging features of `dbx`. Specifying `-ftrap=invalid` causes the program to run with trapping enabled for invalid operation exceptions. Next, invoke `dbx`, issue the `catch fpe` command to stop when a `SIGFPE` is issued, and run the program. On SPARC-based systems, the result resembles this:

```
example% dbx a.out
Reading a.out
Reading ld.so.1
Reading libm.so.2
```

```
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 2773)
signal FPE (invalid floating point operation) in __sqrt at 0x7fa9839c
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
Current function is sqrtm1
    5   return sqrt(x) - 1.0;
(dbx) print x
x = -4.2
(dbx)
```

The output shows that the exception occurred in the `sqrtm1` function as a result of attempting to take the square root of a negative number.

You can also use `dbx` to identify the cause of an exception in code that has not been compiled with `-g`, such as a library routine. In this case, `dbx` will not be able to give the source file and line number, but it can show the instruction that raised the exception. Again, the first step is to recompile the main program with `-ftrap`:

```
example% cc -ftrap=invalid ex.c -lm
```

Now invoke `dbx`, use the `catch fpe` command, and run the program. When an invalid operation exception occurs, `dbx` stops at an instruction following the one that caused the exception. To find the instruction that caused the exception, disassemble several instructions and look for the last floating-point instruction prior to the instruction at which `dbx` has stopped. On SPARC-based systems, the result might resemble the following transcript.

```
example% dbx a.out
Reading a.out
Reading ld.so.1
Reading libm.so.2
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 2931)
signal FPE (invalid floating point operation) in __sqrt at 0x7fa9839c
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
(dbx) dis __sqrt+0x50/4
dbx: warning: unknown language, 'c' assumed
0x7fa98390: __sqrt+0x0050:      neg      %o4, %o1
0x7fa98394: __sqrt+0x0054:      srlx     %o2, 63, %l6
0x7fa98398: __sqrt+0x0058:      fsqrtd   %f0, %f2
0x7fa9839c: __sqrt+0x005c:      srlx     %o1, 63, %l5
(dbx) print $f0f1
$f0f1 = -4.2
```

```
(dbx) print $f2f3
$f2f3 = -NaN.0
(dbx)
```

The output shows that the exception was caused by an `fsqrtd` instruction. Examining the source register shows that the exception was a result of attempting to take the square root of a negative number.

On x86-based systems, because instructions do not have a fixed length, finding the correct address from which to disassemble the code might involve some trial and error. In this example, the exception occurs close to the beginning of a function, so we can disassemble from there. Note that this output assumes the program has been compiled with the `-xlibmil` flag. The following output might be a typical result.

```
example% dbx a.out
Reading a.out
Reading ld.so.1
Reading libc.so.1
(dbx) catch fpe
(dbx) run
Running: a.out
(process id 18566)
signal FPE (invalid floating point operation) in sqrtm1 at 0x80509ab
0x080509ab: sqrtm1+0x001b:      fstpl    0xffffffe0(%ebp)
(dbx) dis sqrtm1+0x16/5
dbx: warning: unknown language, 'c' assumed
0x080509a6: sqrtm1+0x0016:      fsqrt
0x080509a8: sqrtm1+0x0018:      addl     $0x00000008,%esp
0x080509ab: sqrtm1+0x001b:      fstpl    0xffffffe0(%ebp)
0x080509ae: sqrtm1+0x001e:      fwait
0x080509af: sqrtm1+0x001f:      movsd    0xffffffe0(%ebp),%xmm0
(dbx) print $st0
$st0 = -4.2000000000000001776356839400250464 7e+00
(dbx)
```

The output reveals that the exception was caused by a `fsqrt` instruction. Examination of the floating-point registers reveals that the exception was a result of attempting to take the square root of a negative number.

4.4.1.2 Enabling Traps Without Recompilation

In the preceding examples, trapping on invalid operation exceptions was enabled by recompiling the main subprogram with the `-ftrap` flag. In some cases, recompiling the main program might not be possible, so you might need to resort to other means to enable trapping. There are several ways to do this.

When you are using `dbx`, you can enable traps manually by directly modifying the floating-point status register. This can be somewhat tricky because the operating system does not enable the floating-point unit until the first time it is used within a program, at which point the floating-point state is initialized with all traps disabled. Thus, you cannot manually enable trapping until after the program has executed at least one floating-point instruction. In our example, the floating-point unit has already been accessed by the time the `sqrtm1` function is called, so we can set a breakpoint on entry to that function, enable trapping on invalid operation exceptions, instruct `dbx` to stop on the receipt of a `SIGFPE` signal, and continue execution. On SPARC-based systems, the steps are as follows. Note the use of the `assign` command to modify the `%fsr` to enable trapping on invalid operation exceptions:

```
example% dbx a.out
Reading a.out
... etc.
(dbx) stop in sqrtm1
dbx: warning: 'sqrtm1' has no debugger info -- will trigger on first instruction
(2) stop in sqrtm1
(dbx) run
Running: a.out
(process id 23086)
stopped in sqrtm1 at 0x106d8
0x000106d8: sqrtm1       :      save    %sp, -0x70, %sp
(dbx) assign $fsr=0x08000000
dbx: warning: unknown language, 'c' assumed
(dbx) catch fpe
(dbx) cont
signal FPE (invalid floating point operation) in __sqrt at 0xff36b3c4
0xff36b3c4: __sqrt+0x003c:      be      __sqrt+0x98
(dbx)
```

On x86-based systems, the same process might look like this:

```
example% dbx a.out
Reading a.out
... etc.
(dbx) stop in sqrtm1
dbx: warning: 'sqrtm1' has no debugger info -- will trigger on first instruction
(2) stop in sqrtm1
(dbx) run
Running: a.out
(process id 25055)
stopped in sqrtm1 at 0x80506b0
0x080506b0: sqrtm1     :        pushl  %ebp
(dbx) assign $fctrl=0x137e
dbx: warning: unknown language, 'c' assumed
(dbx) catch fpe
(dbx) cont
signal FPE (invalid floating point operation) in sqrtm1 at 0x8050696
```

```
0x08050696: sqrtm1+0x0016:      fstpl  -16(%ebp)
(dbx)
```

In the example above, the `assign` command unmasks (that is, enables trapping on) the invalid operation exception in the floating-point control word. If a program uses SSE2 instructions, you must unmask exceptions in the MXCSR register to enable trapping on exceptions raised by those instructions.

You can also enable trapping without recompiling the main program or using `dbx` by establishing an *initialization routine* that enables traps. This might be useful, for example, if you want to abort the program when an exception occurs without running under a debugger. There are two ways to establish such a routine.

If the object files and libraries that comprise the program are available, you can enable trapping by relinking the program with an appropriate initialization routine. First, create a C source file similar to the following:

```
#include <ieeefp.h>

#pragma init (trapinvalid)

void trapinvalid()
{
     /* FP_X_INV et al are defined in ieeefp.h */
     fpsetmask(FP_X_INV);
}
```

Compile this file to create an object file and link the original program with this object file:

```
example% cc -c init.c
example% cc ex.o init.o -lm
example% a.out
Arithmetic Exception
```

If relinking is not possible but the program has been dynamically linked, you can enable trapping by using the shared object preloading facility of the runtime linker. To do this on SPARC-based systems, create the same C source file as above, but compile as follows:

```
example% cc -Kpic -G -ztext init.c -o init.so -lc
```

To enable trapping, add the path name of the `init.so` object to the list of preloaded shared objects specified by the environment variable `LD_PRELOAD`:

```
example% env LD_PRELOAD=./init.so a.out
Arithmetic Exception
```

See the *Oracle Solaris 11.3 Linkers and Libraries Guide* for more information about creating and preloading shared objects.

In principle, you can change the way any floating-point control modes are initialized by preloading a shared object as described above. However, initialization routines in shared objects, whether preloaded or explicitly linked, are executed by the runtime linker before it passes control to the startup code that is part of the main executable. The startup code then establishes any nondefault modes selected via the `-ftrap`, `-fround`, `-fns` (SPARC), or `-fprecision` (x86) compiler flags; executes any initialization routines that are part of the main executable, including those that are statically linked; and finally passes control to the main program. Therefore, on SPARC, remember the following:

- Any floating-point control modes established by initialization routines in shared objects, such as the traps enabled in the example above, will remain in effect throughout the execution of the program unless they are overridden.
- Any *nondefault* modes selected via the compiler flags will override modes established by initialization routines in shared objects (but default modes selected via compiler flags will not override previously established modes).
- Any modes established either by initialization routines that are part of the main executable or by the main program itself will override both.

On x86-based systems, the situation is slightly more complicated. In general, the startup code automatically supplied by the compiler resets all floating-point modes to the default by calling the `__fpstart` routine (found in the standard C library, `libc`) before establishing any nondefault modes selected by the `-fround`, `-ftrap`, or `-fprecision` flags and passing control to the main program. As a consequence, in order to enable trapping or change any other default floating-point mode on x86-based systems by preloading a shared object with an initialization routine, you must override the `__fpstart` routine so that it does not reset the default floating-point modes. The substitute `__fpstart` routine should still perform the rest of the initialization functions that the standard routine does, however. The following code shows one way to do this. This code assumes that the host platform is running the Oracle Solaris 10 OS or later releases.

```
#include <ieeefp.h>
#include <sys/sysi86.h>

#pragma init (trapinvalid)

void trapinvalid()
{
     /* FP_X_INV et al are defined in ieeefp.h */
     fpsetmask(FP_X_INV);
}

extern int  __fltrounds(), __flt_rounds;
extern int  _fp_hw, _sse_hw;

void __fpstart()
```

```
{
    /* perform the same floating point initializations as
       the standard __fpstart() function but leave all
       floating point modes as is */
    __flt_rounds = __fltrounds();
    (void) sysi86(SI86FPHW, &_fp_hw);

    /* set the following variable to 0 instead if the host
       platform does not support SSE2 instructions */
    _sse_hw = 1;
}
```

4.4.2 Using a Signal Handler to Locate an Exception

The previous section presented several methods for enabling trapping at the outset of a program in order to locate the first occurrence of an exception. In contrast, you can isolate any particular occurrence of an exception by enabling trapping within the program itself. If you enable trapping but do not install a `SIGFPE` handler, the program will abort on the next occurrence of the trapped exception. Alternatively, if you install a `SIGFPE` handler, the next occurrence of the trapped exception will cause the system to transfer control to the handler, which can then print diagnostic information, such as the address of the instruction where the exception occurred, and either abort or resume execution. In order to resume execution with any prospect for a meaningful outcome, the handler might need to supply a result for the exceptional operation as described in the next section.

You can use `ieee_handler` to simultaneously enable trapping on any of the five IEEE floating-point exceptions and either request that the program abort when the specified exception occurs or establish a `SIGFPE` handler. You can also install a `SIGFPE` handler using one of the lower-level functions *sigfpe*(3), *signal*(3c), or *sigaction*(2); however, these functions do not enable trapping as `ieee_handler` does. Remember that a floating-point exception triggers a `SIGFPE` signal only when its trap is enabled.

4.4.2.1 `ieee_handler` (3m)

The syntax of a call to `ieee_handler` is:

`i = ieee_handler`(*action*, *exception*, *handler*)

The two input parameters *action* and *exception* are strings. The third input parameter, *handler*, is of type `sigfpe_handler_type`, which is defined in `floatingpoint.h`.

The three input parameters can take the following values:

Input Parameter	C or C++ Type	Possible Value
action	`char *`	`get`, `set`, `clear`
exception	`char *`	`invalid`, `division`, `overflow`, `underflow`, `inexact`, `all`, `common`

handler	`sigfpe_handler_type`	user-defined routine
		`SIGFPE_DEFAULT`
		`SIGFPE_IGNORE`
		`SIGFPE_ABORT`

When the requested action is `"set"`, `ieee_handler` establishes the handling function specified by *handler* for the exceptions named by exception. The handling function can be `SIGFPE_DEFAULT` or `SIGFPE_IGNORE`, both of which select the default IEEE behavior, `SIGFPE_ABORT`, which causes the program to abort on the occurrence of any of the named exceptions, or the address of a user-supplied subroutine, which causes that subroutine to be invoked (with the parameters described in the `sigaction`(2) manual page for a signal handler installed with the `SA_SIGINFO` flag set) when any of the named exceptions occurs. If the handler is `SIGFPE_DEFAULT` or `SIGFPE_IGNORE`, `ieee_handler` also disables trapping on the specified exceptions; for any other handler, `ieee_handler` enables trapping.

On x86 platforms, the floating-point hardware traps whenever an exception's trap is enabled and its corresponding flag is raised. Therefore, to avoid spurious traps, a program should clear the flag for each specified *exception* before calling `ieee_handler` to enable trapping.

When the requested *action* is `"clear"`, `ieee_handler` revokes whatever handling function is currently installed for the specified *exception* and disables its trap. This is the same as `"set"`ting `SIGFPE_DEFAULT`. The third parameter is ignored when *action* is `"clear"`.

For both the `"set"` and `"clear"` actions, `ieee_handler` returns 0 if the requested action is available and a nonzero value otherwise.

When the requested *action* is `"get"`, `ieee_handler` returns the address of the handler currently installed for the specified *exception* or `SIGFPE_DEFAULT`, if no handler is installed.

The following examples show a few code fragments illustrating the use of `ieee_handler`. This C code causes the program to abort on division by zero:

```
#include <sunmath.h>
/* uncomment the following line on x86 systems */
    /* ieee_flags("clear", "exception", "division", NULL); */
    if (ieee_handler("set", "division", SIGFPE_ABORT) != 0)
```

```
        printf("ieee trapping not supported here \n");
```

The following is the equivalent Fortran code:

```
#include <floatingpoint.h>
c uncomment the following line on x86 systems
c     ieee_flags('clear', 'exception', 'division', %val(0))
      i = ieee_handler('set', 'division', SIGFPE_ABORT)
      if(i.ne.0) print *,'ieee trapping not supported here'
```

This C fragment restores IEEE default exception handling for all exceptions:

```
#include <sunmath.h>
    if (ieee_handler("clear", "all", 0) != 0)
        printf("could not clear exception handlers\n");
```

The following is the same action in Fortran:

```
      i = ieee_handler('clear', 'all', 0)
      if (i.ne.0) print *, 'could not clear exception handlers'
```

4.4.2.2 Reporting an Exception From a Signal Handler

When a `SIGFPE` handler installed via `ieee_handler` is invoked, the operating system provides additional information indicating the type of exception that occurred, the address of the instruction that caused it, and the contents of the machine's integer and floating-point registers. The handler can examine this information and print a message identifying the exception and the location at which it occurred.

To access the information supplied by the system, declare the handler as follows. The remainder of this chapter presents sample code in C; see Appendix A, “Examples” for examples of `SIGFPE` handlers in Fortran.

```
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    ...
}
```

When the handler is invoked, the sig parameter contains the number of the signal that was sent. Signal numbers are defined in `sys/signal.h`; the `SIGFPE` signal number is 8.

The sip parameter points to a structure that records additional information about the signal. For a `SIGFPE` signal, the relevant members of this structure are `sip->si_code` and `sip->si_addr`

(see /usr/include/sys/siginfo.h). The significance of these members depends on the system and on what event triggered the SIGFPE signal.

The sip->si_code member is one of the SIGFPE signal types listed in Table 33, "Types for Arithmetic Exceptions," on page 82. The tokens shown are defined in sys/machsig.h.

TABLE 33 Types for Arithmetic Exceptions

SIGFPE Type	IEEE Type
FPE_INTDIV	n/a
FPE_INTOVF	n/a
FPE_FLTRES	inexact
FPE_FLTDIV	division
FPE_FLTUND	underflow
FPE_FLTINV	invalid
FPE_FLTOVF	overflow

As the previous table shows, each type of IEEE floating-point exception has a corresponding SIGFPE signal type. Integer division by zero (FPE_INTDIV) and integer overflow (FPE_INTOVF) are also included among the SIGFPE types, but because they are not IEEE floating-point exceptions you cannot install handlers for them via ieee_handler. You can install handlers for these SIGFPE types via sigfpe(3); note, though, that integer overflow is ignored by default on all SPARC and x86 platforms. Special instructions can cause the delivery of a SIGFPE signal of type FPE_INTOVF, but Sun compilers do not generate these instructions.

For a SIGFPE signal corresponding to an IEEE floating-point exception, the sip->si_code member indicates which exception occurred. On x86-based systems, it actually indicates the highest priority unmasked exception whose flag is raised. This is normally the same as the exception that last occurred. The sip->si_addr member holds the address of the instruction that caused the exception on SPARC-based systems, and on x86-based systems it holds the address of the instruction at which the trap was taken, usually the next floating-point instruction following the one that caused the exception.

Finally, the *uap* parameter points to a structure that records the state of the system at the time the trap was taken. The contents of this structure are system-dependent; see /usr/include/sys/siginfo.h for definitions of some of its members.

Using the information provided by the operating system, we can write a SIGFPE handler that reports the type of exception that occurred and the address of the instruction that caused it. Example 1, "SIGFPE Handler," on page 83 shows such a handler.

EXAMPLE 1 `SIGFPE` Handler

```
#include <stdio.h>
#include <sys/ieeefp.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned    code, addr;

    code = sip->si_code;
    addr = (unsigned) sip->si_addr;
    fprintf(stderr, "fp exception %x at address %x\n", code,
        addr);
}

int main()
{
    double  x;

    /* trap on common floating point exceptions */
    if (ieee_handler("set", "common", handler) != 0)
        printf("Did not set exception handler\n");
    /* cause an underflow exception (will not be reported) */
    x = min_normal();
    printf("min_normal = %g\n", x);
    x = x / 13.0;
    printf("min_normal / 13.0 = %g\n", x);

    /* cause an overflow exception (will be reported) */
    x = max_normal();
    printf("max_normal = %g\n", x);
    x = x * x;
    printf("max_normal * max_normal = %g\n", x);
    ieee_retrospective(stderr);
    return 0;
}
```

On SPARC systems, the output from this program resembles the following:

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 10d0c
max_normal * max_normal = 1.79769e+308
Note: IEEE floating-point exception flags raised:
```

```
    Inexact;  Underflow;
IEEE floating-point exception traps enabled:
    overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

On x86 platforms, the operating system saves a copy of the accrued exception flags and then clears them before invoking a `SIGFPE` handler. Unless the handler takes steps to preserve them, the accrued flags are lost once the handler returns. Thus, the output from the preceding program does not indicate that an underflow exception was raised, when compiled with `-xarch=386`:

```
min_normal = 2.22507e-308
min_normal / 13.0 = 1.7116e-309
max_normal = 1.79769e+308
fp exception 4 at address 8048fe6
max_normal * max_normal = 1.79769e+308
 Note: IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_handler(3M)
```

But by default, or when compiled with `-xarch=sse2`, the test program loops because the PC never gets past the loop instruction. For Oracle Developer Studio12.5 , it would suffice to add a line of code to increment the PC:

```
uap → UC_mcontext.gregs[REG_PC= +=5;
```

The above code is only covered for `-xarch=sse2` and only if the SSE2 instruction happens to be five bytes long. A completely general SSE2 solution involves decoding the optimized code to find the beginning of the next instruction. Use `fex_set_handling` instead.

In most cases, the instruction that causes the exception does not deliver the IEEE default result when trapping is enabled: in the preceding outputs, the value reported for `max_normal * max_normal` is not the default result for an operation that overflows (i.e., a correctly signed infinity). In general, a `SIGFPE` handler must supply a result for an operation that causes a trapped exception in order to continue the computation with meaningful values. See “4.5 Handling Exceptions” on page 90 for one way to do this.

4.4.3 Using `libm` Exception Handling Extensions to Locate an Exception

C/C++ programs can use the exception handling extensions to the C99 floating-point environment functions in `libm` to locate exceptions in several ways. These extensions include functions that can establish handlers and simultaneously enable traps, just as `ieee_handler` does, but they provide more flexibility. They also support logging of retrospective diagnostic messages regarding floating-point exceptions to a selected file.

4.4.3.1 `fex_set_handling`(3m)

The `fex_set_handling` function allows you to select one of several options, or modes, for handling each type of floating-point exception. The syntax of a call to `fex_set_handling` is:

`ret = fex_set_handling(`*ex*, *mode*, *handler*`);`

The *ex* argument specifies the set of exceptions to which the call applies. It must be a bitwise "or" of the values listed in the first column of Table 34, "Exception Codes for `fex_set_handling`," on page 85. (These values are defined in `fenv.h`.)

TABLE 34 Exception Codes for `fex_set_handling`

Value	Exception
`FEX_INEXACT`	inexact result
`FEX_UNDERFLOW`	underflow
`FEX_OVERFLOW`	overflow
`FEX_DIVBYZERO`	division by zero
`FEX_INV_ZDZ`	0/0 invalid operation
`FEX_INV_IDI`	infinity/infinity invalid operation
`FEX_INV_ISI`	infinity-infinity invalid operation
`FEX_INV_ZMI`	0*infinity invalid operation
`FEX_INV_SQRT`	square root of negative number
`FEX_INV_SNAN`	operation on signaling NaN
`FEX_INV_INT`	invalid integer conversion
`FEX_INV_CMP`	invalid unordered comparison

For convenience, `fenv.h` also defines the following values: `FEX_NONE` (no exceptions), `FEX_INVALID` (all invalid operation exceptions), `FEX_COMMON` (overflow, division by zero, and all invalid operations), and `FEX_ALL` (all exceptions).

The *mode* argument specifies the exception handling mode to be established for the indicated exceptions. There are five possible modes:

- `FEX_NONSTOP` mode provides the IEEE 754 default nonstop behavior. This is equivalent to leaving the exception's trap disabled. Note that unlike `ieee_handler`, `fex_set_handling` allows you to establish nondefault handling for certain types of invalid operation exceptions and retain IEEE default handling for the rest.
- `FEX_NOHANDLER` mode is equivalent to enabling the exception's trap without providing a handler. When an exception occurs, the system transfers control to a previously installed `SIGFPE` handler, if present, or aborts.

- FEX_ABORT mode causes the program to call abort(3c) when the exception occurs.
- FEX_SIGNAL installs the handling function specified by the *handler* argument for the indicated exceptions. When any of these exceptions occurs, the handler is invoked with the same arguments as if it had been installed by ieee_handler.
- FEX_CUSTOM installs the handling function specified by *handler* for the indicated exceptions. Unlike FEX_SIGNAL mode, when an exception occurs, the handler is invoked with a simplified argument list. The arguments consist of an integer whose value is one of the values listed in Table 34, "Exception Codes for fex_set_handling," on page 85 and a pointer to a structure that records additional information about the operation that caused the exception. The contents of this structure are described in the next section and in the *fex_set_handling*(3m) manual page.

Note that the *handler* parameter is ignored if the specified *mode* is FEX_NONSTOP, FEX_NOHANDLER, or FEX_ABORT. fex_set_handling returns a nonzero value if the specified mode is established for the indicated exceptions, and returns zero otherwise. In the following examples, the return value is ignored.

The following examples suggest ways to use fex_set_handling to locate certain types of exceptions. To abort on a 0/0 exception, use the following:

```
fex_set_handling(FEX_INV_ZDZ, FEX_ABORT, NULL);
```

To install a SIGFPE handler for overflow and division by zero, use the following:

```
fex_set_handling(FEX_OVERFLOW | FEX_DIVBYZERO, FEX_SIGNAL,
    handler);
```

In the previous example, the handler function could print the diagnostic information supplied via the sip parameter to a SIGFPE handler, as shown in the previous subsection. By contrast, the following example prints the information about the exception that is supplied to a handler installed in FEX_CUSTOM mode. See the *fex_set_handling*(3m) manual page for more information.

EXAMPLE 2 Printing Information Supplied to Handler Installed in FEX_CUSTOM Mode

```
#include <fenv.h>

void handler(int ex, fex_info_t *info)
{
    switch (ex) {
    case FEX_OVERFLOW:
        printf("Overflow in ");
        break;
    case FEX_DIVBYZERO:
        printf("Division by zero in ");
```

```
    break;

default:
    printf("Invalid operation in ");
}
switch (info->op) {
case fex_add:
    printf("floating point add\n");
    break;
case fex_sub:
    printf("floating point subtract\n");
    break;
case fex_mul:
    printf("floating point multiply\n");
    break;
case fex_div:
    printf("floating point divide\n");
    break;
case fex_sqrt:
    printf("floating point square root\n");
    break;
case fex_cnvt:
    printf("floating point conversion\n");
    break;
case fex_cmp:
    printf("floating point compare\n");
    break;
default:
    printf("unknown operation\n");
}
switch (info->op1.type) {
case fex_int:
    printf("operand 1: %d\n", info->op1.val.i);
    break;
case fex_llong:
    printf("operand 1: %lld\n", info->op1.val.l);
    break;
case fex_float:
    printf("operand 1: %g\n", info->op1.val.f);
    break;
case fex_double:
    printf("operand 1: %g\n", info->op1.val.d);
    break;

case fex_ldouble:
    printf("operand 1: %Lg\n", info->op1.val.q);
    break;
}
```

```
    switch (info->op2.type) {
    case fex_int:
        printf("operand 2: %d\n", info->op2.val.i);
        break;
    case fex_llong:
        printf("operand 2: %lld\n", info->op2.val.l);
        break;
    case fex_float:
        printf("operand 2: %g\n", info->op2.val.f);
        break;
    case fex_double:
        printf("operand 2: %g\n", info->op2.val.d);
        break;
    case fex_ldouble:
        printf("operand 2: %Lg\n", info->op2.val.q);
        break;
    }
}
...
fex_set_handling(FEX_COMMON, FEX_CUSTOM, handler);
```

The handler in the preceding example reports the type of exception that occurred, the type of operation that caused it, and the operands. It does not indicate where the exception occurred. To find out where the exception occurred, you can use retrospective diagnostics.

4.4.3.2 Retrospective Diagnostics

Another way to locate an exception using the `libm` exception handling extensions is to enable logging of retrospective diagnostic messages regarding floating-point exceptions. When you enable logging of retrospective diagnostics, the system records information about certain exceptions. This information includes the type of exception, the address of the instruction that caused it, the manner in which it will be handled, and a stack trace similar to that produced by a debugger. The stack trace recorded with a retrospective diagnostic message contains only instruction addresses and function names; for additional debugging information such as line numbers, source file names, and argument values, you must use a debugger.

The log of retrospective diagnostics does not contain information about every single exception that occurs; if it did, a typical log would be huge, and it would be impossible to isolate unusual exceptions. Instead, the logging mechanism eliminates redundant messages. A message is considered redundant under either of the following two circumstances:

- The same exception has been previously logged at the same location, i.e., with the same instruction address and stack trace
- `FEX_NONSTOP` mode is in effect for the exception and its flag has been previously raised.

In particular, in most programs, only the first occurrence of each type of exception will be logged. When FEX_NONSTOP handling mode is in effect for an exception, clearing its flag via any of the C99 floating-point environment functions allows the next occurrence of that exception to be logged, provided it does not occur at a location at which it was previously logged.

To enable logging, use the fex_set_log function to specify the file to which messages should be delivered. For example, to log messages to the standard error file, use:

```
fex_set_log(stderr);
```

The following code example combines logging of retrospective diagnostics with the shared object preloading facility illustrated in the previous section. By creating the following C source file, compiling it to a shared object, preloading the shared object by supplying its path name in the LD_PRELOAD environment variable, and specifying the names of one or more exceptions (separated by commas) in the FTRAP environment variable, you can simultaneously abort the program on the specified exceptions and obtain retrospective diagnostic output showing where each exception occurs.

EXAMPLE 3 Combined Logging of Retrospective Diagnostics With Shared Object Preloading

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fenv.h>

static struct ftrap_string {
    const char  *name;
    int         value;
} ftrap_table[] = {
    { "inexact", FEX_INEXACT },
    { "division", FEX_DIVBYZERO },

    { "underflow", FEX_UNDERFLOW },
    { "overflow", FEX_OVERFLOW },
    { "invalid", FEX_INVALID },
    { NULL, 0 }
};

#pragma init (set_ftrap)
void set_ftrap()
{
    struct ftrap_string  *f;
    char                 *s, *s0;
    int                  ex = 0;

    if ((s = getenv("FTRAP")) == NULL)
        return;
```

```
    if ((s0 = strtok(s, ",")) == NULL)
        return;

    do {
        for (f = ftrap_table[0]; f->name != NULL; f++) {
            if (!strcmp(s0, f->name))
                ex |= f->value;
        }
    } while ((s0 = strtok(NULL, ",")) != NULL);

    fex_set_handling(ex, FEX_ABORT, NULL);
    fex_set_log(stderr);
}
```

Using the preceding code with the example program given at the beginning of this section produces the following results on SPARC-based systems:

```
env FTRAP=invalid LD_PRELOAD=./init.so a.out
Floating point invalid operation (sqrt) at 0x7fa98398 __sqrt, abort
  0x7fa9839c  __sqrt
  0x00010880  sqrtm1
  0x000108ec  main
Abort
```

The preceding output shows that the invalid operation exception was raised as a result of a square root operation in the routine `sqrtm1`.

As noted above, to enable trapping from an initialization routine in a shared object on x86 platforms, you must override the standard `__fpstart` routine.

Appendix A, “Examples” gives more examples showing typical log outputs. For general information, see the *fex_set_log*(3m) man page.

4.5 Handling Exceptions

Historically, most numerical software has been written without regard to exceptions, and many programmers have become accustomed to environments in which exceptions cause a program to abort immediately. High-quality software packages such as LAPACK are carefully designed to avoid exceptions such as division by zero and invalid operations and to scale their inputs aggressively to preclude overflow and potentially harmful underflow. Neither of these approaches to dealing with exceptions is appropriate in every situation. However, ignoring exceptions can pose problems when you write a program or subroutine that is intended to be used by someone else, for example, who might not have access to the source code. Attempting to avoid all exceptions can require many defensive tests and branches and carry a significant

cost. For more information, see Demmel and Li, “Faster Numerical Algorithms via Exception Handling,” IEEE Trans. Comput. 43 (1994), pp. 983–992.

The default exception response, status flags, and optional trapping facility of IEEE arithmetic are intended to provide a third alternative: continuing a computation in the presence of exceptions and either detecting them after the fact or intercepting and handling them as they occur. As described above, `ieee_flags` or the C99 floating-point environment functions can be used to detect exceptions after the fact, and `ieee_handler` or `fex_set_handling` can be used to enable trapping and install a handler to intercept exceptions as they occur. In order to continue the computation, however, the IEEE standard recommends that a trap handler be able to provide a result for the operation that incurred an exception. A `SIGFPE` handler installed via `ieee_handler` or `fex_set_handling` in `FEX_SIGNAL` mode can accomplish this using the *uap* parameter supplied to a signal handler by the Solaris operating environment. An `FEX_CUSTOM` mode handler installed via `fex_set_handling` can provide a result using the info parameter supplied to such a handler.

Recall that a `SIGFPE` signal handler can be declared in C as follows:

```
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    ...
}
```

When a `SIGFPE` signal handler is invoked as a result of a trapped floating-point exception, the *uap* parameter points to a data structure that contains a copy of the machine's integer and floating-point registers as well as other system-dependent information describing the exception. If the signal handler returns normally, the saved data are restored and the program resumes execution at the point at which the trap was taken. Thus, by accessing and decoding the information in the data structure that describes the exception and possibly modifying the saved data, a `SIGFPE` handler can substitute a user-supplied value for the result of an exceptional operation and continue computation.

An `FEX_CUSTOM` mode handler can be declared as follows:

```
#include <fenv.h>

void handler(int ex, fex_info_t *info)
{
    ...
}
```

When a `FEX_CUSTOM` handler is invoked, the ex parameter indicates which type of exception occurred (it is one of the values listed in Table 34, “Exception Codes for `fex_set_handling`,”

on page 85) and the *info* parameter points to a data structure that contains more information about the exception. Specifically, this structure contains a code representing the arithmetic operation that caused the exception and structures recording the operands, if they are available. It also contains a structure recording the default result that would have been substituted if the exception were not trapped and an integer value holding the bitwise "or" of the exception flags that would have accrued. The handler can modify the latter members of the structure to substitute a different result or change the set of flags that are accrued. (Note that if the handler returns without modifying these data, the program will continue with the default untrapped result and flags just as if the exception were not trapped.)

As an illustration, the following section shows how to substitute a scaled result for an operation that underflows or overflows. See Appendix A, "Examples" for further examples.

4.5.1 Substituting IEEE Trapped Under/Overflow Results

The IEEE standard recommends that when underflow and overflow are trapped, the system should provide a way for a trap handler to substitute an *exponent-wrapped* result, i.e., a value that agrees with what would have been the rounded result of the operation that underflowed or overflowed except that the exponent is wrapped around the end of its usual range, thereby effectively scaling the result by a power of two. The scale factor is chosen to map underflowed and overflowed results as nearly as possible to the middle of the exponent range so that subsequent computations will be less likely to underflow or overflow further. By keeping track of the number of underflows and overflows that occur, a program can scale the final result to compensate for the exponent wrapping. This under/overflow "counting mode" can be used to produce accurate results in computations that would otherwise exceed the range of the available floating-point formats. See P. Sterbenz, *Floating-Point Computation* for more information.

On SPARC-based systems, when a floating-point instruction incurs a trapped exception, the system leaves the destination register unchanged. Thus, in order to substitute the exponent-wrapped result, an under/overflow handler must decode the instruction, examine the operand registers, and generate the scaled result itself. The following example shows a handler that performs these steps.

EXAMPLE 4 Substituting IEEE Trapped Under/Overflow Handler Results for SPARC-Based Systems

```
#include <stdio.h>
#include <ieeefp.h>
#include <math.h>
#include <sunmath.h>
#include <siginfo.h>
```

```
#include <ucontext.h>

#ifdef V8PLUS
/* The upper 32 floating point registers are stored in an area
   pointed to by uap->uc_mcontext.xrs.xrs_prt. Note that this
   pointer is valid ONLY when uap->uc_mcontext.xrs.xrs_id ==
   XRS_ID (defined in sys/procfs.h). */
#include <assert.h>
#include <sys/procfs.h>
#define FPxreg(x)  ((prxregset_t*)uap->uc_mcontext.xrs.xrs_ptr)
->pr_un.pr_v8p.pr_xfr.pr_regs[(x)]
#endif

#define FPreg(x)   uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[(x)]

/*
*  Supply the IEEE 754 default result for trapped under/overflow
*/
void
ieee_trapped_default(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned    instr, opf, rs1, rs2, rd;
    long double qs1, qs2, qd, qscl;
    double      ds1, ds2, dd, dscl;
    float       fs1, fs2, fd, fscl;

    /* get the instruction that caused the exception */
    instr = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

    /* extract the opcode and source and destination register
       numbers */
    opf = (instr >> 5) & 0x1ff;
    rs1 = (instr >> 14) & 0x1f;
    rs2 = instr & 0x1f;
    rd = (instr >> 25) & 0x1f;
    /* get the operands */
    switch (opf & 3) {
    case 1: /* single precision */
        fs1 = *(float*)&FPreg(rs1);
        fs2 = *(float*)&FPreg(rs2);
        break;

    case 2: /* double precision */
#ifdef V8PLUS
        if (rs1 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            ds1 = *(double*)&FPxreg(rs1 & 0x1e);
```

```
        }
        else
            ds1 = *(double*)&FPreg(rs1);
        if (rs2 & 1)

        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            ds2 = *(double*)&FPxreg(rs2 & 0x1e);
        }
        else
            ds2 = *(double*)&FPreg(rs2);
#else
        ds1 = *(double*)&FPreg(rs1);
        ds2 = *(double*)&FPreg(rs2);
#endif
        break;

    case 3: /* quad precision */
#ifdef V8PLUS
        if (rs1 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            qs1 = *(long double*)&FPxreg(rs1 & 0x1e);
        }
        else
            qs1 = *(long double*)&FPreg(rs1);
        if (rs2 & 1)
        {
            assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
            qs2 = *(long double*)&FPxreg(rs2 & 0x1e);
        }
        else
            qs2 = *(long double*)&FPreg(rs2);
#else
        qs1 = *(long double*)&FPreg(rs1);
        qs2 = *(long double*)&FPreg(rs2);
#endif
        break;
    }

    /* set up scale factors */
    if (sip->si_code == FPE_FLTOVF) {
        fscl = scalbnf(1.0f, -96);
        dscl = scalbn(1.0, -768);
        qscl = scalbnl(1.0, -12288);

    } else {
        fscl = scalbnf(1.0f, 96);
```

```
    dscl = scalbn(1.0, 768);
    qscl = scalbnl(1.0, 12288);
}

/* disable traps and generate the scaled result */
fpsetmask(0);
switch (opf) {
case 0x41: /* add single */
    fd = fscl * (fscl * fs1 + fscl * fs2);
    break;

case 0x42: /* add double */
    dd = dscl * (dscl * ds1 + dscl * ds2);
    break;

case 0x43: /* add quad */
    qd = qscl * (qscl * qs1 + qscl * qs2);
    break;
case 0x45: /* subtract single */
    fd = fscl * (fscl * fs1 - fscl * fs2);
    break;

case 0x46: /* subtract double */
    dd = dscl * (dscl * ds1 - dscl * ds2);
    break;

case 0x47: /* subtract quad */
    qd = qscl * (qscl * qs1 - qscl * qs2);
    break;

case 0x49: /* multiply single */
    fd = (fscl * fs1) * (fscl * fs2);
    break;

case 0x4a: /* multiply double */
    dd = (dscl * ds1) * (dscl * ds2);
    break;

case 0x4b: /* multiply quad */
    qd = (qscl * qs1) * (qscl * qs2);
    break;

case 0x4d: /* divide single */
    fd = (fscl * fs1) / (fs2 / fscl);
    break;

case 0x4e: /* divide double */
    dd = (dscl * ds1) / (ds2 / dscl);
```

```
            break;

        case 0x4f: /* divide quad */
            qd = (qscl * qs1) / (qs2 / dscl);
            break;

        case 0xc6: /* convert double to single */
            fd = (float) (fscl * (fscl * ds1));
            break;
        case 0xc7: /* convert quad to single */
            fd = (float) (fscl * (fscl * qs1));
            break;

        case 0xcb: /* convert quad to double */
            dd = (double) (dscl * (dscl * qs1));
            break;
        }

        /* store the result in the destination */
        if (opf & 0x80) {
            /* conversion operation */
            if (opf == 0xcb) {
                /* convert quad to double */
#ifdef V8PLUS
                if (rd & 1)
                {
                    assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                    *(double*)&FPxreg(rd & 0x1e) = dd;
                }

                else
                    *(double*)&FPreg(rd) = dd;
#else
                *(double*)&FPreg(rd) = dd;
#endif
            } else
                /* convert quad/double to single */
                *(float*)&FPreg(rd) = fd;
        } else {
            /* arithmetic operation */
            switch (opf & 3) {
            case 1: /* single precision */
                *(float*)&FPreg(rd) = fd;
                break;
            case 2: /* double precision */
#ifdef V8PLUS
                if (rd & 1)
                {
```

```
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(double*)&FPxreg(rd & 0x1e) = dd;
            }
            else
                *(double*)&FPreg(rd) = dd;
#else
            *(double*)&FPreg(rd) = dd;
#endif
            break;

        case 3: /* quad precision */
#ifdef V8PLUS
            if (rd & 1)
            {
                assert(uap->uc_mcontext.xrs.xrs_id == XRS_ID);
                *(long double*)&FPxreg(rd & 0x1e) = qd;
            }
            else
                *(long double*)&FPreg(rd & 0x1e) = qd;

#else
            *(long double*)&FPreg(rd & 0x1e) = qd;
#endif
            break;
        }
    }
}

int
main()
{
    volatile float   a, b;
    volatile double  x, y;

    ieee_handler("set", "underflow", ieee_trapped_default);
    ieee_handler("set", "overflow", ieee_trapped_default);
    a = b = 1.0e30f;
    a *= b; /* overflow; will be wrapped to a moderate number */
    printf( "%g\n", a );
    a /= b;
    printf( "%g\n", a );
    a /= b; /* underflow; will wrap back */
    printf( "%g\n", a );

    x = y = 1.0e300;
    x *= y; /* overflow; will be wrapped to a moderate number */
    printf( "%g\n", x );
    x /= y;
```

```
    printf( "%g\n", x );
    x /= y; /* underflow; will wrap back */
    printf( "%g\n", x );

    ieee_retrospective(stdout);
    return 0;
}
```

In this example, the variables `a`, `b`, `x`, and `y` have been declared `volatile` only to prevent the compiler from evaluating `a * b`, etc., at compile time. In typical usage, the `volatile` declarations would not be needed.

The output from the preceding program is as follows:

```
159.309
1.59309e-28
1
4.14884e+137
4.14884e-163
1
 Note: IEEE floating-point exception traps enabled:
    underflow;  overflow;
 See the Numerical Computation Guide, ieee_handler(3M)
```

On x86-based systems, the floating-point hardware provides the exponent-wrapped result when a floating-point instruction incurs a trapped underflow or overflow and its destination is a register. When trapped underflow or overflow occurs on a floating-point store instruction, however, the hardware traps without completing the store and without popping the stack, if the store instruction is a store-and-pop. Thus, in order to implement counting mode, an under/overflow handler must generate the scaled result and fix up the stack when a trap occurs on a store instruction. Example 5, “Substituting IEEE Trapped Under/Overflow Handler Results for x86-Based Systems,” on page 98 illustrates such a handler.

EXAMPLE 5 Substituting IEEE Trapped Under/Overflow Handler Results for x86-Based Systems

```
#include <stdio.h>
#include <ieeefp.h>
#include <math.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

/* offsets into the saved fp environment */
#define CW    0    /* control word */
#define SW    1    /* status word */
#define TW    2    /* tag word */
#define OP    4    /* opcode */
```

```
#define EA    5    /* operand address */

#define FPenv(x)    uap->uc_mcontext.fpregs.fp_reg_set.
fpchip_state.state[(x)]
#define FPreg(x)    *(long double *)(10*(x)+(char*)&uap->
uc_mcontext.fpregs.fp_reg_set.fpchip_state.state[7])/*
*  Supply the IEEE 754 default result for trapped under/overflow

*/
void
ieee_trapped_default(int sig, siginfo_t *sip, ucontext_t *uap)
{
    double      dscl;
    float       fscl;
    unsigned    sw, op, top;
    int         mask, e;

    /* preserve flags for untrapped exceptions */
    sw = uap->uc_mcontext.fpregs.fp_reg_set.fpchip_state.status;
    FPenv(SW) |= (sw & (FPenv(CW) & 0x3f));
    /* if the excepting instruction is a store, scale the stack
 top, store it, and pop the stack if need be */
fpsetmask(0);
    op = FPenv(OP) >> 16;
    switch (op & 0x7f8) {
    case 0x110:
    case 0x118:
    case 0x150:
    case 0x158:
    case 0x190:
    case 0x198:
        fscl = scalbnf(1.0f, (sip->si_code == FPE_FLTOVF)?
 -96 : 96);
*(float *)FPenv(EA) = (FPreg(0) * fscl) * fscl;
        if (op & 8) {
            /* pop the stack */
            FPreg(0) = FPreg(1);
            FPreg(1) = FPreg(2);
            FPreg(2) = FPreg(3);
            FPreg(3) = FPreg(4);
            FPreg(4) = FPreg(5);
            FPreg(5) = FPreg(6);
            FPreg(6) = FPreg(7);
            top = (FPenv(SW) >> 10) & 0xe;
            FPenv(TW) |= (3 << top);
            top = (top + 2) & 0xe;
            FPenv(SW) = (FPenv(SW) & ~0x3800) | (top << 10);
        }
```

```
        break;

    case 0x510:
    case 0x518:

    case 0x550:
    case 0x558:
    case 0x590:
    case 0x598:
        dscl = scalbn(1.0, (sip->si_code == FPE_FLTOVF)?
 -768 : 768);
*(double *)FPenv(EA) = (FPreg(0) * dscl) * dscl;
        if (op & 8) {
            /* pop the stack */
            FPreg(0) = FPreg(1);
            FPreg(1) = FPreg(2);
            FPreg(2) = FPreg(3);
            FPreg(3) = FPreg(4);
            FPreg(4) = FPreg(5);
            FPreg(5) = FPreg(6);
            FPreg(6) = FPreg(7);
            top = (FPenv(SW) >> 10) & 0xe;
            FPenv(TW) |= (3 << top);
            top = (top + 2) & 0xe;
            FPenv(SW) = (FPenv(SW) & ~0x3800) | (top << 10);
        }
        break;
    }
}

int main()
{
    volatile float    a, b;
    volatile double    x, y;

    ieee_handler("set", "underflow", ieee_trapped_default);
    ieee_handler("set", "overflow", ieee_trapped_default);
    a = b = 1.0e30f;
    a *= b;
    printf( "%g\n", a );
    a /= b;
    printf( "%g\n", a );
    a /= b;
    printf( "%g\n", a );
    x = y = 1.0e300;
    x *= y;
    printf( "%g\n", x );
```

```
    x /= y;
    printf( "%g\n", x );
    x /= y;
    printf( "%g\n", x );

    ieee_retrospective(stdout);
    return 0;
}
```

As on SPARC-based systems and compiled with `-xarch=386`, the output from the preceding program on x86 is:

```
159.309
1.59309e-28
1
4.14884e+137
4.14884e-163
1
 Note: IEEE floating-point exception traps enabled:
    underflow;  overflow;
 See the Numerical Computation Guide, ieee_handler(3M)
```

Note - With `-xarch=sse2`, this program loops. It would have be completely rewritten for `-xarch=sse2`.

C/C++ programs can use the `fex_set_handling` function in `libm` to install a `FEX_CUSTOM` handler for underflow and overflow. On SPARC-based systems, the information supplied to such a handler always includes the operation that caused the exception and the operands, and this information is sufficient to allow the handler to compute the IEEE exponent-wrapped result, as shown above. On x86-based systems, the available information might not always indicate which particular operation caused the exception; when the exception is raised by one of the transcendental instructions, for example, the `info->op` parameter is set to `fex_other`. (See the `fenv.h` file for definitions.) Moreover, the x86 hardware delivers an exponent-wrapped result automatically, and this can overwrite one of the operands if the destination of the excepting instruction is a floating-point register.

Fortunately, the `fex_set_handling` feature provides a simple way for a handler installed in `FEX_CUSTOM` mode to substitute the IEEE exponent-wrapped result for an operation that underflows or overflows. When either of these exceptions is trapped, the handler can set

```
info->res.type = fex_nodata;
```

to indicate that the exponent-wrapped result should be delivered. The following is an example showing such a handler:

```
#include <stdio.h>
```

```
#include <fenv.h>

void handler(int ex, fex_info_t *info) {
    info->res.type = fex_nodata;
}
int main()
{
    volatile float  a, b;
    volatile double x, y;

    fex_set_log(stderr);
    fex_set_handling(FEX_UNDERFLOW | FEX_OVERFLOW, FEX_CUSTOM,
        handler);
    a = b = 1.0e30f;
    a *= b; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", a);
    a /= b;
    printf("%g\n", a);
    a /= b; /* underflow; will wrap back */
    printf("%g\n", a);

    x = y = 1.0e300;
    x *= y; /* overflow; will be wrapped to a moderate number */
    printf("%g\n", x);
    x /= y;

    printf("%g\n", x);
    x /= y; /* underflow; will wrap back */
    printf("%g\n", x);

    return 0;
}
```

The output from the preceding program resembles the following:

```
Floating point overflow at 0x00010924 main, handler: handler
  0x00010928 main
159.309
1.59309e-28
Floating point underflow at 0x00010994 main, handler: handler
  0x00010998 main
1
Floating point overflow at 0x000109e4 main, handler: handler
  0x000109e8 main
4.14884e+137
4.14884e-163
Floating point underflow at 0x00010a4c main, handler: handler
  0x00010a50 main
1
```

The previous program example works on SPARC, on x86 with `-xarch=sse2`, and on x86 with `-xarch=386`.

♦ ♦ ♦ CHAPTER 5

Compiler Code Generation

This chapter describes the compiler code generation features of the Oracle Developer Studio 12.5 compilers specifically related to numerical computation. The chapter is divided into the following topics:

5.1 Supported Operation Systems, Hardware, and Memory Model

Oracle Developer Studio 12.5 supports Oracle Solaris 10 starting with update 10 and Oracle Solaris 11, and Oracle and Red Hat Enterprise Linux releases 5 and 6.

Oracle Developer Studio supports the same hardware as the corresponding Oracle Solaris release. For SPARC, Oracle Solaris 10 and 11 support only SPARC processors that support a 64-bit address space memory model. For x86, Oracle Solaris 11 supports only x86 processors that support a 64-bit address space memory model. Oracle Solaris 10 also supports many x86 processors that only support a 32-bit address space memory model.

All 64-bit processors can execute programs compiled for either 32-bit or 64-bit address spaces. Oracle Solaris 10 and 11 support executing 32-bit programs on a 64-bit operating system.

32-bit and 64-bit addressing is selected at compile time with the `-m32` and `-m64` command-line options. These affect the size of C integer and pointer variables. The operating system provides some 32-bit and 64-bit runtime libraries, and the compilers provide additional libraries for specific languages.

A program that requires a 64-bit address space must be compiled with -m64. Many programs could be compiled with either one of the address models and run correctly, so it is natural to ask which model is faster. A C program that moves a lot of integer and pointer data to and from memory might be half as fast when using -m64. But the 64-bit application binary interfaces (ABIs) have more registers than the 32-bit ABIs, so fewer memory moves might be necessary. For most programs, the performance difference is not significant but to be sure of a particular program, it is best to compile both ways and test for correctness and performance.

Note - In Sun Studio 11 and older releases, the memory model was not an explicit option like -m32 or -m64, but was built into the -xarch options which had different names corresponding to the memory model. With Sun Studio 12, the memory model options and architecture options have been separate.

5.2 Code Generation Options

Oracle Developer Studio 12.5 supports many different SPARC and x86 processor chips. For each of these processor chips, there is an -xtarget= command-line compiler option. The --xtarget= option specifies the instruction set implemented, the particular processor chip implementing that instruction set, and the size of the various caches. -xtarget serves as a macro for the following:

-xarch=*architecture*	The compiler optimizes code by using the -xarch= option to determine which instructions are implemented in hardware and thus suitable for code generation.
-xchip=*chip*	The compiler optimizes code by using the -xchip= option to determine which specific chip is intended and thus how instructions should be scheduled.
-xcache=*cache-size*	The compiler optimizes code by using the -xcache= option to determine how to block loops to minimize memory traffic.

By optimizing for a particular target, you get the best code for that target, which might be very bad for a different target with a different instruction set and scheduling constraints. When one executable is intended to be run on many different target systems, then the default generic code generation is best, and it can also be explicitly selected with the option -xtarget=generic.

Some of the -xtarget= names are less than obvious. To specify a particular target, you can use the -native option with the Oracle Developer Studio compilers, which will select the -xtarget= automatically for the system being compiled on. On SPARC systems, similar

information is displayed by the `fpversion` command. For more information, see Appendix B, "SPARC Behavior and Implementation".

5.3 Default Address Model and Code Generation

In Oracle Developer Studio 12.5 and previous releases, the default address space model is `-m32` for 32-bits for Oracle Solaris OS. However, on Linux the default is `-m32` for 32-bit hardware and `-m64` for 64-bit hardware.

For SPARC, the default is `-xarch=sparc`.

For x86, the default is `-xarch=sse2`.

Note - In previous Oracle Developer Studio releases, the x86 default was `-xarch-386` for `-m32` and `-xarch=sse2` for `-m64`.

The new x86 default `-m32 -xarch=sse2` implements the same ABI as the previous default `-m32 -xarch=386`. Floating-point operands and results are passed in x87 floating-point registers. However, the following single-precision and double-precision floating-point operations are usually performed in `sse2` registers.

- `+`
- `-`
- `*`
- `/`
- `sqrt`
- `convert`

x87 registers are still used for long double operations and hardware elementary transcendental function evaluations.

This means that the results of x86 default-compiled floating-point computations might vary slightly with Oracle Developer Studio 12.5 relative to previous Studio releases, even on the same hardware and operating system.

5.4 Compilation Options

Oracle Developer Studio 12.5 compilers accept many options that affect code generation. The following list highlights certain code generation options that are not valid for all programs. In

the case of large programs written by many authors over many years, it is often the case that nobody can say with certainty which code transformations do and do not adversely affect the logic of the program, and so the following must be used with care.

`-fast` Macro for many different generally useful transformations. It is easier to remember `-fast` than all its constituents. The definition of `-fast` varies from release to release. Not all its transformations are legitimate for all programs. If `-fast` is used, it might be followed by additional options that undo some of its effects, e.g. using `-fast -fsimple=0 -fns=no -xvector=no` to disable three options discussed below. Compiling with `-dryrun` is a good way to see which options are actually enabled by a particular compiler's command line options.

`-fsimple` Permits certain simplifying assumptions about floating-point modes, exceptions, and roundoff that are not true for all programs. The compilers use these assumptions to justify various value-changing transformations, which vary between releases.`-fsimple=0` is the default and is safest. `-fsimple=1` is safe for many programs and `-fsimple=2` is risky for many. Simply testing program execution with a few inputs is not sufficient to verify suitability of `-fsimple=2`. Its effects might be beneficial for some input data and not others.

`-fns` Does not affect code generation but causes programs to begin execution with nonstandard underflow mode enabled. This is contrary to the IEEE standard and so nonstandard results might be obtained, including invalid results and infinite loops. Many programs run the same no matter how underflow is handled. These programs might run faster with `-fns` if the run-time hardware is slower for standard underflow. Recent SPARC servers have no performance advantage for nonstandard mode.

`-ftrap=common` Does not affect code generation but causes programs to begin execution with traps enabled for overflow, division by zero, and invalid IEEE exceptions; this is contrary to the IEEE standard and might cause early termination of programs that depend on nonstop IEEE exception handling. `-ftrap=none` is the default for C, C++, and F77, but not F95.

`-fnonstd` Macro for `-fns` and `-ftrap=common`.

`-xvector` Used to enable vector math library transformations with `-xvector=lib`, and SIMD transformations with `-xvector=simd`. `-xvector=lib` will change numerical results as slightly different vector-oriented `libmvec` implementations of common elementary transcendental functions are

used instead of the `libm` versions. These vector versions assume default rounding is in effect.

`-xreduction` Enables a broader range of program loops to be parallelized when parallelization is enabled with `-xautopar` or `-xopenmp`. Reduction operations are those like summing a vector or computing the dot product of two vectors; the sums can be taken in any order in exact arithmetic, but will produce slightly different results in finite precision floating-point arithmetic. Indeed, the order of accumulation might not even be deterministic and so results might vary slightly from run to run even with the same program and same data and same hardware.

See the man pages and user's guides of each compiler for more information.

5.5 Reproducible Results

As is discussed in section D.11 of the Numerical Computation Guide, even standard-conforming implementations of IEEE arithmetic might produce different results. Often these results are equally good, but equally often it is tedious or difficult to prove that. For many purposes, it's better to sacrifice some performance to reduce the amount of error analysis necessary to validate results. It's not obvious when minor or major differences in output are equally good, and whether they are due to user program errors, compiler optimization errors, or hardware errors.

There are several principal root causes of varying results of IEEE floating-point arithmetic. The following lists these causes and describes some approaches to reducing gratuitous variation across the releases and supported platforms of Oracle Developer Studio. Note that each approach increases reproducibility while potentially reducing performance. Sometimes the performance loss can be noticeable.

5.5.1 Transcendental Functions

Most of the common math library functions standardized by programming languages, such ass exponential, logarithmic, and trigonometric functions, are expensive to round correctly, compared to rational arithmetic or algebraic functions like square root (`sqrt()`). Nearly correctly rounded functions are suitable for most purposes, and much faster. But the fastest nearly-correctly-rounded functions differ on different platforms.

- Use portable code for the functions used by the application. One source of such code is the Freely-Distributable Math Library, `fdlibm`. It can be obtained from the Netlib software repository.

- Avoid the `-xvector` option. The vectorized versions of transcendental functions are optimized for a particular platform and produce slightly different results on different platforms.
- Avoid the x86 hardware transcendental instructions. Even though these instructions have error bounds almost as small as possible, they are not quite correctly rounded. Also, the Intel and AMD versions differ occasionally, even though both are quite good. With Oracle Developer Studio C/C++ compilers, `-xbuiltin=%default` can be used, especially after `-fast`, to make sure that none of the transcendental instructions are substituted inline by the compiler for built-in transcendental functions. Likewise the `-xnolibmil` option after `-fast` disables inline templates; `libm.il` from Oracle Developer Studio might have some templates that invoke the transcendental instructions.

5.5.2 Associative Operations

Addition and multiplication are associative in real arithmetic - sums and products may be computed in any order. But in the presence of roundoff, the order of evaluation affects the computed answer.

- Avoid the `-xreduction` parallelization option. Oracle Developer Studio optimizes reductions in a way that is not deterministic.
- Avoid Fortran's `DOT` and `MATMUL` operations. These intrinsics in Fortran 90 and later are implemented by different methods on different platforms and will round differently. If parallelization is also enabled, the results might not be deterministic due to reduction optimizations. Dot product and matrix multiplication operations can be coded in portable Fortran such as that available in the LAPACK library from the Netlib software repository.

5.5.3 Indeterminate Evaluation

In many languages, the order of external expression evaluation is not specified by the language. Thus if `ranf(x)()` is a random number generator, the expression `ranf(x) * a + ranf(x) * b()` might give different results for different compilers, or different optimization levels of the same compiler, if the order of evaluation of the two `ranf(x)()` invocations changes.

Avoid expressions with two external references - split such expressions into several statements with at most one external reference in each. Thus

```
z = ranf(x) * a + ranf(x) * b()
```

can be replaced by

```
t = ranf(x) * a()
```

```
z = t + ranf(x) * b().
```

5.5.4 Non-Portable Types

`long double` in C/C++ is implemented differently on SPARC and x86 in Studio, with 113 significant bits and 64 significant bits respectively. Programs with explicit `long double` variables are thus bound to behave differently on SPARC and x86.

5.5.5 Implicit Higher Precision

In some situations, expressions might be evaluated in higher precision than is explicit in the source code. This can happen when x87 extended precision registers are used to evaluate expressions involving single or double precision variables. It can also happen when fused multiply-add operations are substituted for pairs of multiplications and additions.

- Avoid optimizing multiply-add pairs as fused multiply-add operations. Use `-fma=none` after `-fast`.
- If `-xarch=386` must be used and there is no explicit use of long double types, then it might be possible to mitigate the effects of extended-precision expression evaluation by compiling with `-fprecision=single` if all variables are float, or `-fprecision=double` if all variables are double. However if Fortran `complex*8` variables are in use under `-xarch=386`, then there is no way to insure that all expression evaluations occur in single precision. Using `-m64` is preferable to `-m32` because function values are passed in registers of the same precision as the functions.

5.6 Independent Confirmation

The foregoing discussion of reproducibility is based on the assumption that the result to be reproduced is correct, perhaps one of many correct results. But how do we ultimately know for sure? Some programs have proofs, but often the proof is more complicated than the program. Why believe it more than the program? Some programs model physical systems that have conservation laws that can be checked, but what if the interesting physical discovery to be made is that the conservation law is incomplete or incorrect?

Any important decision should be confirmed by independent means. In the most important cases of decisions made with the aid of computers, *independent means* might need to encompass a different machine, with a different instruction set, running a different operating system, with a program written in a different computer language, implementing a different

algorithm, then all done by a different investigator on another continent who thinks in a different natural language. How far one wants to go in this direction depends on how much an incorrect conclusion would cost.

Thus when writing a program to test base conversion, for instance, at least take care to use test algorithms utterly different from any likely to be used in the base conversion functions to be tested. Thus even slow, simple algorithms have their place in Computer Science — for testing fast complicated algorithms.

♦ ♦ ♦ APPENDIX A

Examples

This appendix provides examples of how to accomplish some popular tasks. The examples are written either in Fortran or ANSI C, and many depend on the current versions of `libm` and `libsunmath`. These examples were tested with Oracle Developer Studio 12.5 on an Oracle Solaris 10 Update 10 OS or later release. C examples are compiled using the `-lsunmath -lm` options.

A.1 IEEE Arithmetic

The following examples show one way you can examine the hexadecimal representations of floating-point numbers. Note that you can also use the debuggers to look at the hexadecimal representations of stored data.

The following C program prints a double precision approximation to `??` and single precision infinity:

EXAMPLE 6 Double Precision Example

```
#include <math.h>
#include <sunmath.h>

int main() {
 union {
   float         flt;
   unsigned     un;
 } r;
 union {
   double      dbl;
   unsigned    un[2];
 } d;

 /* double precision */
 d.dbl = M_PI;
```

```
 (void) printf("DP Approx pi = %08x %08x = %18.17e \n",
     d.un[0], d.un[1], d.dbl);

 /* single precision */
 r.flt = infinityf();
 (void) printf("Single Precision %8.7e : %08x \n",
     r.flt, r.un);

 return 0;
}
```

On SPARC-based systems, compiled with `-lsunmath`, the output of the preceding program looks like the following:

```
DP Approx pi = 400921fb 54442d18 = 3.14159265358979312e+00
Single Precision Infinity: 7f800000
```

The following Fortran program prints the smallest normal numbers in each format:

EXAMPLE 7 Print Smallest Normal Numbers in Each Format (Continued)

```
 program print_ieee_values
c
c the purpose of the implicit statements is to ensure
c that the floatingpoint pseudo-intrinsic functions
c are declared with the correct type
c
 implicit real*16 (q)
 implicit double precision (d)
 implicit real (r)
 real*16           z
 double precision  x
 real              r
c
 z = q_min_normal()
 write(*,7) z, z
 7 format('min normal, quad: ',1pe47.37e4,/,' in hex ',z32.32)
c
 x = d_min_normal()

 write(*,14) x, x
 14 format('min normal, double: ',1pe23.16,' in hex ',z16.16)
c
 r = r_min_normal()
 write(*,27) r, r
 27 format('min normal, single: ',1pe14.7,' in hex ',z8.8)
c
```

```
 end
```

On SPARC-based systems, the corresponding output reads as follows:

```
min normal, quad:   3.362103143112093506262677817321752602 6E-4932
 in hex 00010000000000000000000000000000
min normal, double:  2.2250738585072014-308 in hex 0010000000000000
min normal, single:  1.1754944E-38 in hex 00800000
```

A.2 The Math Libraries

This section shows examples that use functions from the math library.

A.2.1 Random Number Generator

The following example calls a random number generator to generate an array of numbers and uses a timing function to measure the time it takes to compute the EXP of the given numbers:

EXAMPLE 8 Random Number Generator

```
#ifdef DP
#define GENERIC double precision
#else
#define GENERIC real
#endif
#define SIZE 400000

 program example
c
 implicit GENERIC (a-h,o-z)
 GENERIC x(SIZE), y, lb, ub
 real tarray(2), u1, u2
c
c compute EXP on random numbers in [-ln2/2,ln2/2]
 lb = -0.3465735903
 ub = 0.3465735903
c
c generate array of random numbers
#ifdef DP
 call d_init_addrans()
 call d_addrans(x,SIZE,lb,ub)
#else
 call r_init_addrans()
```

```
 call r_addrans(x,SIZE,lb,ub)
#endif
c
c start the clock
 call dtime(tarray)
 u1 = tarray(1)
c
c compute exponentials
 do 16 i=1,SIZE
  y = exp(x(i))
 16 continue
c
c get the elapsed time
 call dtime(tarray)
 u2 = tarray(1)
 print *,'time used by EXP is ',u2-u1
 print *,'last values for x and exp(x) are ',x(SIZE),y
c
 call flush(6)
 end
```

To compile the preceding example, place the source code in a file with the suffix `F` (not `f`) so that the compiler will automatically invoke the preprocessor, and specify either `-DSP` or `-DDP` on the command line to select single or double precision.

This example shows how to use the `d_addrans` function to generate blocks of random data uniformly distributed over a user-specified range:

EXAMPLE 9 Using the `d_addrans` Function

```
/*
 * test SIZE*LOOPS random arguments to sin in the range
 * [0, threshold] where
 * threshold = 3E30000000000000 (3.72529029846191406e-09)
 */

#include <math.h>
#include <sunmath.h>
#define SIZE 10000
#define LOOPS 100
int main()
{
 double  x[SIZE], y[SIZE];
 int  i, j, n;
 double  lb, ub;
 union {
  unsigned    u[2];
```

```
 double    d;
}  upperbound;

upperbound.u[0] = 0x3e300000;
upperbound.u[1] = 0x00000000;

/* initialize the random number generator */
d_init_addrans_();

/* test (SIZE * LOOPS) arguments to sin */
for (j = 0; j < LOOPS; j++) {

  /*
  * generate a vector, x, of length SIZE,
  * of random numbers to use as
  * input to the trig functions.
  */
  n = SIZE;
  ub = upperbound.d;
  lb = 0.0;

  d_addrans_(x, &n, &lb, &ub);

  for (i = 0; i < n; i++)
    y[i] = sin(x[i]);

  /* is sin(x) == x?  It ought to, for tiny x. */
  for (i = 0; i < n; i++)
   if (x[i] != y[i])
     printf(
     " OOPS: %d sin(%18.17e)=%18.17e \n",
     i, x[i], y[i]);
}
printf(" comparison ended; no differences\n");
ieee_retrospective_();
return 0;
}
```

A.2.2 IEEE Recommended Functions

The following Fortran example uses some functions recommended by the IEEE standard:

EXAMPLE 10 IEEE Recommended Functions

```
c
```

```
c Demonstrate how to call 5 of the more interesting IEEE
c recommended functions from Fortran. These are implemented
c with "bit-twiddling", and so are as efficient as you could
c hope. The IEEE standard for floating-point arithmetic
c doesn't require these, but recommends that they be
c included in any IEEE programming environment.
c
c For example, to accomplish
c  y = x * 2**n,
c since the hardware stores numbers in base 2,
c shift the exponent by n places.
c

c Refer to
c ieee_functions(3m)
c libm_double(3f)
c libm_single(3f)
c
c The 5 functions demonstrated here are:
c
c ilogb(x): returns the base 2 unbiased exponent of x in
c   integer format
c signbit(x): returns the sign bit, 0 or 1
c copysign(x,y): returns x with y's sign bit
c nextafter(x,y): next representable number after x, in
c   the direction y
c scalbn(x,n): x * 2**n
c
c function    double precision      single precision
c ------------------------------------------------------
c ilogb(x)     i = id_ilogb(x)      i = ir_ilogb(r)
c signbit(x)    i = id_signbit(x)      i = ir_signbit(r)
c copysign(x,y)    x = d_copysign(x,y)      r = r_copysign(r,s)
c nextafter(x,y)    z = d_nextafter(x,y)      r = r_nextafter(r,s)
c scalbn(x,n)    x = d_scalbn(x,n)      r = r_scalbn(r,n)
 program ieee_functions_demo
 implicit double precision (d)
 implicit real (r)
 double precision   x, y, z, direction
 real               r, s, t, r_direction
 integer            i, scale

 print *
 print *, 'DOUBLE PRECISION EXAMPLES:'
 print *

 x = 32.0d0
 i = id_ilogb(x)
```

```
write(*,1) x, i
1 format(' The base 2 exponent of ', F4.1, ' is ', I2)

x = -5.5d0
y = 12.4d0
z = d_copysign(x,y)
write(*,2) x, y, z
2    format(F5.1, ' was given the sign of ', F4.1,
    *   ' and is now ', F4.1)


x = -5.5d0
i = id_signbit(x)
print *, 'The sign bit of ', x, ' is ', i

x = d_min_subnormal()
direction = -d_infinity()
y = d_nextafter(x, direction)
write(*,3) x
3 format(' Starting from ', 1PE23.16E3,
    -   ', the next representable number ')
write(*,4) direction, y
4 format('    towards ', F4.1, ' is ', 1PE23.16E3)

x = d_min_subnormal()
direction = 1.0d0
   y = d_nextafter(x, direction)
write(*,3) x
write(*,4) direction, y
x = 2.0d0
scale = 3
y = d_scalbn(x, scale)
write (*,5) x, scale, y
5 format(' Scaling ', F4.1, ' by 2**', I1, ' is ', F4.1)
print *
print *, 'SINGLE PRECISION EXAMPLES:'
print *

r = 32.0
i = ir_ilogb(r)
write (*,1) r, i

r = -5.5
i = ir_signbit(r)
print *, 'The sign bit of ', r, ' is ', i

r = -5.5
s = 12.4
```

```
 t = r_copysign(r,s)
 write (*,2) r, s, t

 r = r_min_subnormal()
 r_direction = -r_infinity()
 s = r_nextafter(r, r_direction)
 write(*,3) r
 write(*,4) r_direction, s

 r = r_min_subnormal()
 r_direction = 1.0e0
 s = r_nextafter(r, r_direction)
 write(*,3) r
 write(*,4) r_direction, s

 r = 2.0
 scale = 3
 s = r_scalbn(r, scale)
 write (*,5) r, scale, y

 print *
 end
```

The output from this program is shown in the following example.

EXAMPLE 11 Output of Example A-5

```
DOUBLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is  5
-5.5 was given the sign of 12.4 and is now  5.5
The sign bit of    -5.5 is   1
Starting from  4.9406564584124654E-324, the next representable
   number towards -Inf is  0.0000000000000000E+000
Starting from  4.9406564584124654E-324, the next representable
   number towards  1.0 is  9.8813129168249309E-324
Scaling  2.0 by 2**3 is 16.0

SINGLE PRECISION EXAMPLES:

The base 2 exponent of 32.0 is  5
The sign bit of    -5.5 is   1
-5.5 was given the sign of 12.4 and is now  5.5
Starting from  1.4012984643248171E-045, the next representable
   number towards -Inf is  0.0000000000000000E+000
Starting from  1.4012984643248171E-045, the next representable
   number towards  1.0 is  2.8025969286496341E-045
Scaling  2.0 by 2**3 is 16.0
```

If using the `f95` compiler with the `-f77` compatibility option, the following additional messages are displayed.

```
 Note: IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

A.2.3 IEEE Special Values

The following C program calls several of the *ieee_values*(3m) functions:

```
#include <math.h>
#include <sunmath.h>

int main()
{
 double       x;
 float       r;

 x = quiet_nan(0);
 printf("quiet NaN: %.16e = %08x %08x \n",
    x, ((int *) &x)[0], ((int *) &x)[1]);

 x = nextafter(max_subnormal(), 0.0);
 printf("nextafter(max_subnormal,0) = %.16e\n",x);
 printf("                           = %08x %08x\n",
   ((int *) &x)[0], ((int *) &x)[1]);

 r = min_subnormalf();
 printf("single precision min subnormal = %.8e = %08x\n",
      r, ((int *) &r)[0]);

 return 0;
}
```

Remember to specify both `-lsunmath` and `-lm` when linking.

On SPARC-based systems, the output looks like the following:

```
quiet NaN: NaN = 7ff80000 00000000
nextafter(max_subnormal,0) = 2.2250738585072004e-308
 = 000fffff fffffffe
single precision min subnormal = 1.40129846e-45 = 00000001
```

Because the x86 architecture is “little-endian”, the output on x86 is slightly different, such that the high and low order words of the hexadecimal representations of the double precision numbers are reversed:

```
quiet NaN: NaN = ffffffff 7fffffff
nextafter(max_subnormal,0) = 2.2250738585072004e-308
                           = fffffffe 000fffff
single precision min subnormal = 1.40129846e-45 = 00000001
```

Fortran programs that use `ieee_values` functions should take care to declare those functions' types:

```
 program print_ieee_values
c
c the purpose of the implicit statements is to insure
c that the floating-point pseudo-instrinsic
c functions are declared with the correct type
c
 implicit real*16 (q)
 implicit double precision (d)
 implicit real (r)
 real*16 z, zero, one
 double precision    x
 real                r
c
 zero = 0.0
 one = 1.0
 z = q_nextafter(zero, one)
 x = d_infinity()
 r = r_max_normal()
c
 print *, z
 print *, x
 print *, r
c
 end
```

On SPARC, the output reads as follows:

```
6.475175119438025110924438958227646E-4966
Inf
3.4028235E+38
```

A.2.4 `ieee_flags` — Rounding Direction

The following example demonstrates how to set the rounding mode to *round towards zero*:

```
#include <math.h>
#include <sunmath.h>

int main()
{
 int         i;
 double      x, y;
 char        *out_1, *out_2, *dummy;

 /* get prevailing rounding direction */
 i = ieee_flags("get", "direction", "", &out_1);

 x = sqrt(.5);
 printf("With rounding direction %s, \n", out_1);
 printf("sqrt(.5) = 0x%08x 0x%08x = %16.15e\n",
        ((int *) &x)[0], ((int *) &x)[1], x);

 /* set rounding direction */
 if (ieee_flags("set", "direction", "tozero", &dummy) != 0)
   printf("Not able to change rounding direction!\n");
 i = ieee_flags("get", "direction", "", &out_2);

 x = sqrt(.5);
 /*
  * restore original rounding direction before printf, since
  * printf is also affected by the current rounding direction
  */
 if (ieee_flags("set", "direction", out_1, &dummy) != 0)
   printf("Not able to change rounding direction!\n");
 printf("\nWith rounding direction %s,\n", out_2);
 printf("sqrt(.5) = 0x%08x 0x%08x = %16.15e\n",
        ((int *) &x)[0], ((int *) &x)[1], x);

 return 0;
}
```

The following output of the previous rounding direction short program shows the effects of rounding towards zero on SPARC:

```
demo% cc rounding_direction.c -lsunmath -lm
demo% a.out
With rounding direction nearest,
sqrt(.5) = 0x3fe6a09e 0x667f3bcd  = 7.071067811865476e-01

With rounding direction tozero,
sqrt(.5) = 0x3fe6a09e 0x667f3bcc  = 7.071067811865475e-01
demo%
```

The following output of the previous rounding direction short program shows the effects of rounding towards zero on x86:

```
demo% cc rounding_direction.c -lsunmath -lm
demo% a.out
With rounding direction nearest,
sqrt(.5) = 0x667f3bcd 0x3fe6a09e  = 7.071067811865476e-01

With rounding direction tozero,
sqrt(.5) = 0x667f3bcc 0x3fe6a09e  = 7.071067811865475e-01
demo%
```

To set rounding direction towards zero from a Fortran program, use the following example:

```
program ieee_flags_demo
character*16 out

i = ieee_flags('set', 'direction', 'tozero', out)
if (i.ne.0) print *, 'not able to set rounding direction'

i = ieee_flags('get', 'direction', '', out)
print *, 'Rounding direction is: ', out

end
```

The output is as follows:

```
demo% f95 ieee_flags_demo.f
demo% a.out
 Rounding direction is: tozero
```

If the program is compiled using the `f95` compiler with the `-f77` compatibility option, the output includes the following additional messages.

```
demo% f95 ieee_flags_demo.f -f77
demo% a.out
 Note: Rounding direction toward zero
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M), ieee_handler(3M)
```

A.2.5 C99 Floating-Point Environment Functions

The next example illustrates the use of several of the C99 floating-point environment functions. The `norm` function computes the Euclidean norm of a vector and uses the environment functions to handle underflow and overflow. The main program calls this function with vectors that are

scaled to ensure that underflows and overflows occur, as the retrospective diagnostic output shows

EXAMPLE 12 C99 Floating-Point Environment Functions

```
#include <stdio.h>
#include <math.h>
#include <sunmath.h>
#include <fenv.h>

/*
*  Compute the euclidean norm of the vector x avoiding
*  premature underflow or overflow
*/
double norm(int n, double *x)
{
    fenv_t  env;
    double  s, b, d, t;
    int     i, f;

    /* save the environment, clear flags, and establish nonstop
       exception handling */
    feholdexcept(&env);

    /* attempt to compute the dot product x.x */
    d = 1.0; /* scale factor */
    s = 0.0;
    for (i = 0; i < n; i++)
        s += x[i] * x[i];

    /* check for underflow or overflow */
    f = fetestexcept(FE_UNDERFLOW | FE_OVERFLOW);
    if (f & FE_OVERFLOW) {
        /* first attempt overflowed, try again scaling down */
        feclearexcept(FE_OVERFLOW);
        b = scalbn(1.0, -640);
        d = 1.0 / b;
        s = 0.0;
        for (i = 0; i < n; i++) {
            t = b * x[i];
            s += t * t;
        }
    }
    else if (f & FE_UNDERFLOW && s < scalbn(1.0, -970)) {
        /* first attempt underflowed, try again scaling up */
        b = scalbn(1.0, 1022);
        d = 1.0 / b;
```

```
        s = 0.0;
        for (i = 0; i < n; i++) {
            t = b * x[i];
            s += t * t;
        }
    }

    /* hide any underflows that have occurred so far */
    feclearexcept(FE_UNDERFLOW);

    /* restore the environment, raising any other exceptions
       that have occurred */
    feupdateenv(&env);

    /* take the square root and undo any scaling */
    return d * sqrt(s);
}

int main()
{
    double x[100], l, u;
    int    n = 100;

    fex_set_log(stdout);

    l = 0.0;
    u = min_normal();
    d_lcrans_(x, &n, &l, &u);
    printf("norm: %g\n", norm(n, x));
    l = sqrt(max_normal());
    u = l * 2.0;
    d_lcrans_(x, &n, &l, &u);
    printf("norm: %g\n", norm(n, x));

    return 0;
}
```

On SPARC-based systems, compiling and running this program produces output like the following:

```
demo% cc norm.c -lsunmath -lm
demo% a.out
Floating point underflow at 0x000153a8 __d_lcrans_, nonstop mode
  0x000153b4  __d_lcrans_
  0x00011594  main
Floating point underflow at 0x00011244 norm, nonstop mode
  0x00011248  norm
  0x000115b4  main
norm: 1.32533e-307
```

```
Floating point overflow at 0x00011244 norm, nonstop mode
  0x00011248  norm
  0x00011660  main
norm: 2.02548e+155
```

The following code example shows the effect of the `fesetprec` function on x86-based systems. This function is not available on SPARC-based systems. The `while` loops attempt to determine the available precision by finding the largest power of two that rounds off entirely when it is added to one. As the first loop shows, this technique does not always work as expected on architectures like x86-based systems that evaluate all intermediate results in extended precision. Thus, the `fesetprec` function can be used to guarantee that all results will be rounded to the desired precision, as the second loop shows.

EXAMPLE 13 `fesetprec` Function (x86)

```
#include <math.h>
#include <fenv.h>

int main()
{
    double  x;

    x = 1.0;
    while (1.0 + x != 1.0)
        x *= 0.5;
    printf("%d significant bits\n", -ilogb(x));

    fesetprec(FE_DBLPREC);
    x = 1.0;
    while (1.0 + x != 1.0)
        x *= 0.5;
    printf("%d significant bits\n", -ilogb(x));

    return 0;
}
```

Compiling on x86 systems with `cc A8.c -lm -xarch=386` creates

```
64 significant bits
53 significant bit
```

Finally, the following example shows one way to use the environment functions in a multithreaded program to propagate floating-point modes from a parent thread to a child thread and recover exception flags raised in the child thread when it joins with the parent. See the *Multithreaded Programming Guide* for more information on writing multi-threaded programs.

EXAMPLE 14 Using Environment Functions in a Multithread Program

```
#include <thread.h>
#include <fenv.h>

fenv_t  env;

void * child(void *p)
{
    /* inherit the parent's environment on entry */
    fesetenv(&env);
    ...
    /* save the child's environment before exit */
    fegetenv(&env);
}

void parent()
{
    thread_t tid;
    void *arg;
    ...
    /* save the parent's environment before creating the child */
    fegetenv(&env);
    thr_create(NULL, NULL, child, arg, NULL, &tid);
    ...
    /* join with the child */
    thr_join(tid, NULL, &arg);
    /* merge exception flags raised in the child into the
       parent's environment */
    fex_merge_flags(&env);
    ...
}
```

A.3 Exceptions and Exception Handling

A.3.1 `ieee_flags` — Accrued Exceptions

Generally, a user program *examines* or *clears* the accrued exception bits. The following example is a C program that examines the accrued exception flags.

EXAMPLE 15 Examining the Accrued Exception Flags

```
#include <sunmath.h>
#include <sys/ieeefp.h>

int main()
{
 int     code, inexact, division, underflow, overflow, invalid;
 double  x;
 char    *out;

 /* cause an underflow exception */
 x = max_subnormal() / 2.0;

 /* this statement insures that the previous */
 /* statement is not optimized away          */
 printf("x = %g\n",x);

 /* find out which exceptions are raised */
 code = ieee_flags("get", "exception", "", &out);

 /* decode the return value */
 inexact =      (code >> fp_inexact)     & 0x1;
 underflow =    (code >> fp_underflow)   & 0x1;
 division =     (code >> fp_division)    & 0x1;
 overflow =     (code >> fp_overflow)    & 0x1;
 invalid =      (code >> fp_invalid)     & 0x1;

      /* "out" is the raised exception with the highest priority */
 printf(" Highest priority exception is: %s\n", out);
 /* The value 1 means the exception is raised, */
 /* 0 means it isn't.                          */
 printf("%d %d %d %d %d\n", invalid, overflow, division,
  underflow, inexact);
 ieee_retrospective_();
 return 0;
}
```

The output from running the previous program is as follows:

```
demo% a.out
x = 1.11254e-308
 Highest priority exception is: underflow
0 0 0 1 1
 Note:IEEE floating-point exception flags raised:
    Inexact;  Underflow;
 See the Numerical Computation Guide, ieee_flags(3M)
```

The same can be done from Fortran:

EXAMPLE 16 Examining the Accrued Exception Flags – Fortran

```
/*
A Fortran example that:
    *  causes an underflow exception
    *  uses ieee_flags to determine which exceptions are raised
    *  decodes the integer value returned by ieee_flags
    *  clears all outstanding exceptions
Remember to save this program in a file with the suffix .F, so that
the c preprocessor is invoked to bring in the header file
floatingpoint.h.
*/
#include <floatingpoint.h>

    program decode_accrued_exceptions
    double precision   x
    integer            accrued, inx, div, under, over, inv
    character*16       out
    double precision   d_max_subnormal
c Cause an underflow exception
    x = d_max_subnormal() / 2.0

c Find out which exceptions are raised
    accrued = ieee_flags('get', 'exception', '', out)

c Decode value returned by ieee_flags using bit-shift intrinsics
    inx   = and(rshift(accrued, fp_inexact)  , 1)
    under = and(rshift(accrued, fp_underflow), 1)
    div   = and(rshift(accrued, fp_division) , 1)
    over  = and(rshift(accrued, fp_overflow) , 1)
    inv   = and(rshift(accrued, fp_invalid)  , 1)

c The exception with the highest priority is returned in "out"
    print *, "Highest priority exception is ", out

c The value 1 means the exception is raised; 0 means it is not
    print *, inv, over, div, under, inx

c Clear all outstanding exceptions
    i = ieee_flags('clear', 'exception', 'all', out)
    end
```

The output is as follows:

```
 Highest priority exception is underflow
   0  0  0  1  1
```

While it is unusual for a user program to *set* exception flags, it can be done. This is demonstrated in the following C example.

```
#include <sunmath.h>

int main()
{
 int     code;
 char    *out;

 if (ieee_flags("clear", "exception", "all", &out) != 0)
     printf("could not clear exceptions\n");
 if (ieee_flags("set", "exception", "division", &out) != 0)
     printf("could not set exception\n");
 code = ieee_flags("get", "exception", "", &out);
 printf("out is: %s , fp exception code is: %X \n",
  out, code);

 return 0;
}
```

On SPARC, the output from the preceding program is:

```
out is: division , fp exception code is: 2
```

On x86, the output is:

```
out is: division , fp exception code is: 4
```

A.3.2 `ieee_handler`: Trapping Exceptions

Note - The following examples apply only to the Oracle Solaris OS.

The following is a Fortran program that installs a signal handler to locate an exception, for SPARC-based systems only:

EXAMPLE 17 Trap on Underflow (SPARC)

```
  program demo
c declare signal handler function
 external fp_exc_hdl
 double precision d_min_normal
 double precision x
c set up signal handler
 i = ieee_handler('set', 'common', fp_exc_hdl)
 if (i.ne.0) print *, 'ieee trapping not supported here'
c cause an underflow exception (it will not be trapped)
 x = d_min_normal() / 13.0
```

```
 print *, 'd_min_normal() / 13.0 = ', x
c cause an overflow exception
c the value printed out is unrelated to the result
 x = 1.0d300
 x = x * x
 print *, '1.0d300*1.0d300 = ', x
 end
c
c the floating-point exception handling function
c
 integer function fp_exc_hdl(sig, sip, uap)
 integer sig, code, addr
 character label*16
c
c The structure /siginfo/ is a translation of siginfo_t
c from <sys/siginfo.h>
c
 structure /fault/
 integer address
 end structure
 structure /siginfo/
 integer si_signo
 integer si_code
 integer si_errno
 record /fault/ fault
 end structure

 record /siginfo/ sip
c See <sys/machsig.h> for list of FPE codes
c Figure out the name of the SIGFPE
 code = sip.si_code
 if (code.eq.3) label = 'division'
 if (code.eq.4) label = 'overflow'
 if (code.eq.5) label = 'underflow'
 if (code.eq.6) label = 'inexact'
 if (code.eq.7) label = 'invalid'
 addr = sip.fault.address
c Print information about the signal that happened
 write (*,77) code, label, addr
 77  format ('floating-point exception code ', i2, ',',
     *  a17, ',', ' at address ', z8 )
 end
```

When the previous code is compiled with `-f77`, the output is as follows:

```
 d_min_normal() / 13.0 =     1.7115952757748-309
floating-point exception code  4, overflow        , at address    1131C
 1.0d300*1.0d300 =     1.0000000000000+300
 Note: IEEE floating-point exception flags raised:
```

```
   Inexact;  Underflow;
 IEEE floating-point exception traps enabled:
   overflow; division by zero; invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
 ieee_handler(3M)
```

The following is a more complex C example on a SPARC-based system:

EXAMPLE 18 Trap on Invalid, Division by 0, Overflow, Underflow, and Inexact (SPARC)

```
/*
 * Generate the 5 IEEE exceptions: invalid, division,
 * overflow, underflow and inexact.
 *
 * Trap on any floating point exception, print a message,
* and continue.*
 * Note that you could also inquire about raised exceptions by
*    i = ieee("get","exception","",&out);* where out contains the name of the highest
 exception
 * raised, and i can be decoded to find out about all the
 * exceptions raised.
 */

#include <sunmath.h>
#include <signal.h>
#include <siginfo.h>
#include <ucontext.h>

extern void trap_all_fp_exc(int sig, siginfo_t *sip,
  ucontext_t *uap);

int main()
{
 double  x, y, z;
 char  *out;

 /*
  * Use ieee_handler to establish "trap_all_fp_exc"
  * as the signal handler to use whenever any floating
  * point exception occurs.
  */

 if (ieee_handler("set", "all", trap_all_fp_exc) != 0)
  printf(" IEEE trapping not supported here.\n");
 /* disable trapping (uninteresting) inexact exceptions */
 if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
   printf("Trap handler for inexact not cleared.\n");
 /* raise invalid */
```

```
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("1. Invalid: signaling_nan(0) * 2.5\n");
x = signaling_nan(0);
y = 2.5;
z = x * y;

/* raise division */
if (ieee_flags("clear", "exception", "all", &out) != 0)
 printf(" could not clear exceptions\n");
printf("2. Div0: 1.0 / 0.0\n");
x = 1.0;
y = 0.0;
z = x / y;

/* raise overflow */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("3. Overflow: -max_normal() - 1.0e294\n");
x = -max_normal();
y = -1.0e294;
z = x + y;

/* raise underflow */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("4. Underflow: min_normal() * min_normal()\n");
x = min_normal();
y = x;
z = x * y;

/* enable trapping on inexact exception */
if (ieee_handler("set", "inexact", trap_all_fp_exc) != 0)
  printf("Could not set trap handler for inexact.\n");
/* raise inexact */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
printf("5. Inexact: 2.0 / 3.0\n");
x = 2.0;
y = 3.0;
z = x / y;
/* don't trap on inexact */
if (ieee_handler("set", "inexact", SIGFPE_IGNORE) != 0)
  printf(" could not reset inexact trap\n");

/* check that we're not trapping on inexact anymore */
if (ieee_flags("clear", "exception", "all", &out) != 0)
  printf(" could not clear exceptions\n");
```

```
 printf("6. Inexact trapping disabled; 2.0 / 3.0\n");
 x = 2.0;
 y = 3.0;
 z = x / y;

 /* find out if there are any outstanding exceptions */
 ieee_retrospective_();

 /* exit gracefully */
 return 0;
}

void trap_all_fp_exc(int sig, siginfo_t *sip, ucontext_t *uap) {
 char  *label = "undefined";

/* see /usr/include/sys/machsig.h for SIGFPE codes */
 switch (sip->si_code) {
 case FPE_FLTRES:
   label = "inexact";
   break;
 case FPE_FLTDIV:
   label = "division";
   break;
 case FPE_FLTUND:
   label = "underflow";
   break;
 case FPE_FLTINV:
   label = "invalid";
   break;
 case FPE_FLTOVF:
   label = "overflow";
   break;
 }

 printf(
 " signal %d, sigfpe code %d: %s exception at address %x\n",
   sig, sip->si_code, label, sip->__data.__fault.__addr);
}
```

The output is similar to the following:

```
1. Invalid: signaling_nan(0) * 2.5
   signal 8, sigfpe code 7: invalid exception at address 10da8
2. Div0: 1.0 / 0.0
   signal 8, sigfpe code 3: division exception at address 10e44
3. Overflow: -max_normal() - 1.0e294
   signal 8, sigfpe code 4: overflow exception at address 10ee8
4. Underflow: min_normal() * min_normal()
   signal 8, sigfpe code 5: underflow exception at address 10f80
```

```
5. Inexact: 2.0 / 3.0
   signal 8, sigfpe code 6: inexact exception at address 1106c
6. Inexact trapping disabled; 2.0 / 3.0
Note: IEEE floating-point exception traps enabled:
   underflow; overflow; division by zero; invalid operation;
See the Numerical Computation Guide, ieee_handler(3M)
```

The following code shows how you can use `ieee_handler` and the include files to modify the default result of certain exceptional situations on SPARC:

EXAMPLE 19 Modifying the Default Result of Exceptional Situations

```
/*
 * Cause a division by zero exception and use the
 * signal handler to substitute MAXDOUBLE (or MAXFLOAT)
 * as the result.
 *
 * compile with the flag -Xa
 */

#include <values.h>
#include <siginfo.h>
#include <ucontext.h>

void division_handler(int sig, siginfo_t *sip, ucontext_t *uap);

int main() {
 double     x, y, z;
 float     r, s, t;
 char     *out;

 /*
  * Use ieee_handler to establish division_handler as the
  * signal handler to use for the IEEE exception division.
  */
 if (ieee_handler("set","division",division_handler)!=0) {
  printf(" IEEE trapping not supported here.\n");
 }

 /* Cause a division-by-zero exception */
 x = 1.0;
 y = 0.0;
 z = x / y;

 /*
  * Check to see that the user-supplied value, MAXDOUBLE,
  * is indeed substituted in place of the IEEE default
```

```
 * value, infinity.
 */
printf("double precision division: %g/%g = %g \n",x,y,z);

/* Cause a division-by-zero exception */
r = 1.0;
s = 0.0;
t = r / s;

/*
 * Check to see that the user-supplied value, MAXFLOAT,
 * is indeed substituted in place of the IEEE default
 * value, infinity.
 */
printf("single precision division: %g/%g = %g \n",r,s,t);

ieee_retrospective_();

return 0;
}

void division_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
int       inst;
unsigned       rd, mask, single_prec=0;
float       f_val = MAXFLOAT;
double       d_val = MAXDOUBLE;
long       *f_val_p = (long *) &f_val;

/* Get instruction that caused exception. */
inst = uap->uc_mcontext.fpregs.fpu_q->FQu.fpq.fpq_instr;

/*
 * Decode the destination register. Bits 29:25 encode the
 * destination register for any SPARC floating point
 * instruction.
 */
mask = 0x1f;
rd = (mask & (inst >> 25));

/*
 * Is this a single precision or double precision
 * instruction?  Bits 5:6 encode the precision of the
 * opcode; if bit 5 is 1, it's sp, else, dp.
 */

mask = 0x1;
single_prec = (mask & (inst >> 5));
```

```
 /* put user-defined value into destination register */
 if (single_prec) {
  uap->uc_mcontext.fpregs.fpu_fr.fpu_regs[rd] =
   f_val_p[0];
 } else {
 uap->uc_mcontext.fpregs.fpu_fr.fpu_dregs[rd/2] = d_val;
 }
}
```

The following output is as expected:

```
double precision division: 1/0 = 1.79769e+308
single precision division: 1/0 = 3.40282e+38
Note: IEEE floating-point exception traps enabled:
   division by zero;
See the Numerical Computation Guide, ieee_handler(3M)
```

A.3.3 `ieee_handler`: Abort on Exceptions

You can use `ieee_handler` to force a program to abort in case of certain floating-point exceptions:

```
#include <floatingpoint.h>
 program abort
c
 ieeer = ieee_handler('set', 'division', SIGFPE_ABORT)
 if (ieeer .ne. 0) print *, ' ieee trapping not supported'
 r = 14.2
 s = 0.0
 r = r/s
c
 print *, 'you should not see this; system should abort'
c
 end
```

A.3.4 `libm` Exception Handling Features

The following examples show how to use some of the exception handling features provided by `libm`. The first example is based on the following task: given a number x and coefficients a_0, a_1,..., a_N, and b_0, b_1,..., b_{N-1}, evaluate the function `f(x)` and its first derivative `f'(x)`, where `f()` is the continued fraction:

$$f(x) = a_0 + b_0 / (x + a_1 + b_1 / (x + ... / (x + a_{N-1} + b_{N-1} / (x + a_N))...)).$$

Computing `f()` is straightforward in IEEE arithmetic: even if one of the intermediate divisions overflows or divides by zero, the default value specified by the standard (a correctly signed infinity) turns out to yield the correct result. Computing `f'()`, on the other hand, can be more difficult because the simplest form for evaluating it can have removable singularities. If the computation encounters one of these singularities, it will attempt to evaluate one of the indeterminate forms 0/0, 0*infinity, or infinity/infinity, all of which raise invalid operation exceptions. W. Kahan has proposed a method for handling these exceptions via a feature called *presubstitution*.

Presubstitution is an extension of the IEEE default response to exceptions that lets the user specify in advance the value to be substituted for the result of an exceptional operation. Using the exception handling facilities in `libm`, a program can implement presubstitution easily by installing a handler in the `FEX_CUSTOM` exception handling mode. This mode allows the handler to supply any value for the result of an exceptional operation simply by storing that value in the data structure pointed to by the info parameter passed to the handler. The following example is a sample program to compute the continued fraction and its derivative using presubstitution implemented with a `FEX_CUSTOM` handler.

EXAMPLE 20 Computing the Continued Fraction and Its Derivative Using the `FEX_CUSTOM` Handler

```
#include <stdio.h>
#include <sunmath.h>
#include <fenv.h>
volatile double p;
void handler(int ex, fex_info_t *info)
{
    info->res.type = fex_double;
    if (ex == FEX_INV_ZMI)
        info->res.val.d = p;
    else
        info->res.val.d = infinity();
}

/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double x, double *pf, double *pf1)
{
    fex_handler_t    oldhdl; /* for saving/restoring handlers */
    volatile double  t;
    double           f, f1, d, d1, q;
    int              j;
```

```
    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    fex_set_handling(FEX_DIVBYZERO, FEX_NONSTOP, NULL);
    fex_set_handling(FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI,
        FEX_CUSTOM, handler);

    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
        q = b[j] / d;
        /* the following assignment to the volatile variable t
           is needed to maintain the correct sequencing between
           assignments to p and evaluation of f1 */
        t = f1 = (-d1 / d) * q;
        p = b[j-1] * d1 / b[j];
        f = a[j] + q;
    }

    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}

/* For the following coefficients, x = -3, 1, 4, and 5 will all
   encounter intermediate exceptions */
double a[] = { -1.0, 2.0, -3.0, 4.0, -5.0 };
double b[] = { 2.0, 4.0, 6.0, 8.0 };

int main()
{
    double  x, f, f1;
    int     i;

    feraiseexcept(FE_INEXACT); /* prevent logging of inexact */
    fex_set_log(stdout);
    fex_set_handling(FEX_COMMON, FEX_ABORT, NULL);
    for (i = -5; i <= 5; i++) {
        x = i;
        continued_fraction(4, a, b, x, &f, &f1);
        printf("f(% g) = %12g, f'(% g) = %12g\n", x, f, x, f1);
    }
    return 0;
}
```

Several comments about the program are in order. On entry, the function `continued_fraction` saves the current exception handling modes for division by zero and all invalid operation exceptions. It then establishes nonstop exception handling for division by zero and a `FEX_CUSTOM` handler for the three indeterminate forms. This handler will substitute infinity for both 0/0 and infinity/infinity, but it will substitute the value of the global variable `p` for 0*infinity. Note that `p` must be recomputed each time through the loop that evaluates the function in order to supply the correct value to substitute for a subsequent 0*infinity invalid operation. Note also that `p` must be declared `volatile` to prevent the compiler from eliminating it, since it is not explicitly mentioned elsewhere in the loop. Finally, to prevent the compiler from moving the assignment to `p` above or below the computation that can incur the exception for which `p` provides the presubstitution value, the result of that computation is also assigned to a `volatile` variable (called `t` in the program). The final call to `fex_setexcepthandler` restores the original handling modes for division by zero and the invalid operations.

The main program enables logging of retrospective diagnostics by calling the `fex_set_log` function. Before it does so, it raises the inexact flag; this has the effect of preventing the logging of inexact exceptions. Recall that in `FEX_NONSTOP` mode, an exception is not logged if its flag is raised, as explained in the section “4.4.3.2 Retrospective Diagnostics” on page 88. The main program also establishes `FEX_ABORT` mode for the common exceptions to ensure that any unusual exceptions not explicitly handled by `continued_fraction` will cause program termination. Finally, the program evaluates a particular continued fraction at several different points. As the following sample output shows, the computation does indeed encounter intermediate exceptions:

```
f(-5) =     -1.59649,   f'(-5) =      -0.1818
f(-4) =     -1.87302,   f'(-4) =    -0.428193
Floating point division by zero at 0x08048dbe continued_fraction, nonstop mode
  0x08048dc1  continued_fraction
  0x08048eda  main
Floating point invalid operation (inf/inf) at 0x08048dcf continued_fraction, handler:
 handler
  0x08048dd2  continued_fraction
  0x08048eda  main
Floating point invalid operation (0*inf) at 0x08048dd2 continued_fraction, handler:
 handler
  0x08048dd8  continued_fraction
  0x08048eda  main
f(-3) =           -3,   f'(-3) =     -3.16667
f(-2) = -4.44089e-16,   f'(-2) =     -3.41667
f(-1) =     -1.22222,   f'(-1) =    -0.444444
f( 0) =     -1.33333,   f'( 0) =     0.203704
f( 1) =           -1,   f'( 1) =     0.333333
f( 2) =    -0.777778,   f'( 2) =      0.12037
f( 3) =    -0.714286,   f'( 3) =    0.0272109
f( 4) =    -0.666667,   f'( 4) =     0.203704
f( 5) =    -0.777778,   f'( 5) =    0.0185185
```

The exceptions that occur in the computation of `f'(x)` at x = 1, 4, and 5 do not result in retrospective diagnostic messages because they occur at the same site in the program as the exceptions that occur when x = –3.

The preceding program might not represent the most efficient way to handle the exceptions that can occur in the evaluation of a continued fraction and its derivative. One reason is that the presubstitution value must be recomputed in each iteration of the loop regardless of whether or not it is needed. In this case, the computation of the presubstitution value involves a floating-point division, and on modern SPARC and x86 processors, floating-point division is a relatively slow operation. Moreover, the loop itself already involves two divisions, and because most SPARC and x86 processors cannot overlap the execution of two different division operations, divisions are likely to be a bottleneck in the loop; adding another division would exacerbate the bottleneck.

It is possible to rewrite the loop so that only one division is needed, and in particular, the computation of the presubstitution value need not involve a division. To rewrite the loop in this way, one must precompute the ratios of adjacent elements of the coefficients in the `b` array. This would remove the bottleneck of multiple division operations, but it would not eliminate all of the arithmetic operations involved in the computation of the presubstitution value. Furthermore, the need to assign both the presubstitution value and the result of the operation to be presubstituted to `volatile` variables introduces additional memory operations that slow the program. While those assignments are necessary to prevent the compiler from reordering certain key operations, they effectively prevent the compiler from reordering other unrelated operations, too. Thus, handling the exceptions in this example via presubstitution requires additional memory operations and precludes some optimizations that might otherwise be possible. Can these exceptions be handled more efficiently?

In the absence of special hardware support for fast presubstitution, the most efficient way to handle exceptions in this example may be to use flags, as the following version does:

EXAMPLE 21 Using Flags to Handle Exceptions (Continued)

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

/*
*  Evaluate the continued fraction given by coefficients a[j] and
*  b[j] at the point x; return the function value in *pf and the
*  derivative in *pf1
*/
void continued_fraction(int N, double *a, double *b,
    double         x, double *pf, double *pf1)
{
    fex_handler_t  oldhdl;
```

```
    fexcept_t      oldinvflag;
    double         f, f1, d, d1, pd1, q;
    int            j;

    fex_getexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);
    fegetexceptflag(&oldinvflag, FE_INVALID);

    fex_set_handling(FEX_DIVBYZERO | FEX_INV_ZDZ | FEX_INV_IDI |
        FEX_INV_ZMI, FEX_NONSTOP, NULL);
    feclearexcept(FE_INVALID);

    f1 = 0.0;
    f = a[N];
    for (j = N - 1; j >= 0; j--) {
        d = x + f;
        d1 = 1.0 + f1;
        q = b[j] / d;
        f1 = (-d1 / d) * q;
        f = a[j] + q;
    }

    if (fetestexcept(FE_INVALID)) {
        /* recompute and test for NaN */
        f1 = pd1 = 0.0;
        f = a[N];
        for (j = N - 1; j >= 0; j--) {
            d = x + f;
            d1 = 1.0 + f1;
            q = b[j] / d;
            f1 = (-d1 / d) * q;
            if (isnan(f1))
                f1 = b[j] * pd1 / b[j+1];
            pd1 = d1;
            f = a[j] + q;
        }
    }

    fesetexceptflag(&oldinvflag, FE_INVALID);
    fex_setexcepthandler(&oldhdl, FEX_DIVBYZERO | FEX_INVALID);

    *pf = f;
    *pf1 = f1;
}
```

In this version, the first loop attempts the computation of `f(x)` and `f'(x)` in the default nonstop mode. If the invalid flag is raised, the second loop recomputes `f(x)` and `f'(x)` explicitly testing for the appearance of a NaN. Usually, no invalid operation exception occurs, so the program only executes the first loop. This loop has no references to `volatile` variables and no extra

arithmetic operations, so it will run as fast as the compiler can make it go. The cost of this efficiency is the need to write a second loop nearly identical to the first to handle the case when an exception occurs. This trade-off is typical of the dilemmas that floating-point exception handling can pose.

A.3.5 Using `libm` Exception Handling With Fortran Programs

The exception handling facilities in `libm` are primarily intended to be used from C/C++ programs, but by using the Sun Fortran language interoperability features, you can call some `libm` functions from Fortran programs as well.

Note - For consistent behavior, do not use both the `libm` exception handling functions and the `ieee_flags` and `ieee_handler` functions in the same program.

The following example shows a Fortran version of the program to evaluate a continued fraction and its derivative using presubstitution (SPARC only):

EXAMPLE 22 Evaluating a Continued Fraction and Its Derivative Using Presubstitution (SPARC)

```
c
c Presubstitution handler
c
 subroutine handler(ex, info)
 structure /fex_numeric_t/
 integer type
 union
 map
 integer i
 end map
 map
 integer*8 l
 end map
 map
 real f
 end map
 map
 real*8 d
 end map
 map
 real*16 q
 end map
 end union
```

```
 end structure

 structure /fex_info_t/
 integer op, flags
 record /fex_numeric_t/ op1, op2, res
 end structure
 integer ex
 record /fex_info_t/ info
 common /presub/ p
 double precision p, d_infinity
 volatile p
c 4 = fex_double; see <fenv.h> for this and other constants
 info.res.type = 4
c x'80' = FEX_INV_ZMI
 if (loc(ex) .eq. x'80') then
 info.res.d = p
 else
 info.res.d = d_infinity()
 endif
 return
 end
c
c Evaluate the continued fraction given by coefficients a(j) and
c b(j) at the point x; return the function value in f and the
c derivative in f1
c
 subroutine continued_fraction(n, a, b, x, f, f1)
 integer n
 double precision a(*), b(*), x, f, f1
 common /presub/ p
 integer j, oldhdl
 dimension oldhdl(24)
 double precision d, d1, q, p, t
 volatile p, t
 data ixff2/x'ff2'/
 data ix2/x'2'/
 data ixb0/x'b0'/

 external fex_getexcepthandler, fex_setexcepthandler
 external fex_set_handling, handler
c$pragma c(fex_getexcepthandler, fex_setexcepthandler)
c$pragma c(fex_set_handling)
c x'ff2' = FEX_DIVBYZERO | FEX_INVALID
 call fex_getexcepthandler(oldhdl, %val(ixff2))
c x'2' = FEX_DIVBYZERO, 0 = FEX_NONSTOP
 call fex_set_handling(%val(ix2), %val(0), %val(0))
c x'b0' = FEX_INV_ZDZ | FEX_INV_IDI | FEX_INV_ZMI, 3 = FEX_CUSTOM
 call fex_set_handling(%val(ixb0), %val(3), handler)
```

```
 f1 = 0.0d0
 f = a(n+1)
 do j = n, 1, -1
 d = x + f
 d1 = 1.0d0 + f1
 q = b(j) / d
 f1 = (-d1 / d) * q
c
c the following assignment to the volatile variable t
c is needed to maintain the correct sequencing between
c assignments to p and evaluation of f1
 t = f1
 p = b(j-1) * d1 / b(j)
 f = a(j) + q
 end do
 call fex_setexcepthandler(oldhdl, %val(ixff2))
 return
 end

c Main program
c
 program cf
 integer i
 double precision a, b, x, f, f1
 dimension a(5), b(4)
 data a /-1.0d0, 2.0d0, -3.0d0, 4.0d0, -5.0d0/
 data b /2.0d0, 4.0d0, 6.0d0, 8.0d0/
 data ixffa/x'ffa'/

 external fex_set_handling
c$pragma c(fex_set_handling)
c x'ffa' = FEX_COMMON, 1 = FEX_ABORT
 call fex_set_handling(%val(ixffa), %val(1), %val(0))
 do i = -5, 5
 x = dble(i)
 call continued_fraction(4, a, b, x, f, f1)
 write (*, 1) i, f, i, f1
 end do
 1  format('f(', I2, ') = ', G12.6, ', f''(', I2, ') = ', G12.6)
 end
```

The output from this program compiled with the `-f77` flag reads as follows:

```
f(-5) = -1.59649    , f'(-5) = -.181800
f(-4) = -1.87302    , f'(-4) = -.428193
f(-3) = -3.00000    , f'(-3) = -3.16667
f(-2) = -.444089E-15, f'(-2) = -3.41667
f(-1) = -1.22222    , f'(-1) = -.444444
```

```
f( 0) = -1.33333    , f'( 0) = 0.203704
f( 1) = -1.00000    , f'( 1) = 0.333333
f( 2) = -.777778    , f'( 2) = 0.120370
f( 3) = -.714286    , f'( 3) = 0.272109E-01
f( 4) = -.666667    , f'( 4) = 0.203704
f( 5) = -.777778    , f'( 5) = 0.185185E-01
 Note: IEEE floating-point exception flags raised:
    Inexact;  Division by Zero;  Underflow; Invalid Operation;
 IEEE floating-point exception traps enabled:
    overflow;  division by zero;  invalid operation;
 See the Numerical Computation Guide, ieee_flags(3M),
ieee_handler(3M)
```

A.4 Miscellaneous

A.4.1 `sigfpe`: Trapping Integer Exceptions

The previous section showed examples of using `ieee_handler`. In general, when there is a choice between using `ieee_handler` or `sigfpe`, the former is recommended.

Note - `sigfpe` is available only in the Oracle Solaris OS.

There are instances, such as trapping integer arithmetic exceptions, when `sigfpe` is the handler to be used. Example 23, “Trapping Integer Exceptions,” on page 147 traps on integer division by zero, on SPARC-based systems.

EXAMPLE 23 Trapping Integer Exceptions

```
/* Generate the integer division by zero exception */
#include <signal.h>
#include <siginfo.h>
#include <ucontext.h>
void int_handler(int sig, siginfo_t *sip, ucontext_t *uap);
int main() {
int a, b, c;
/*
* Use sigfpe(3) to establish "int_handler" as the signal handler
* to use on integer division by zero
*/
/*
* Integer division-by-zero aborts unless a signal
```

```
* handler for integer division by zero is set up
*/
sigfpe(FPE_INTDIV, int_handler);

a = 4;
b = 0;
c = a / b;
printf("%d / %d = %d\n\n", a, b, c);
return 0;
}
void int_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
printf("Signal %d, code %d, at addr %x\n",
sig, sip->si_code, sip->__data.__fault.__addr);
/*
* automatically for floating-point exceptions but not for
* integer division by zero.
*/
uap->uc_mcontext.gregs[REG_PC] =
uap->uc_mcontext.gregs[REG_nPC];
}
```

A.4.2 Calling Fortran From C

The following is a simple example of a C driver calling Fortran subroutines. Refer to *Oracle Developer Studio 12.5: C User's Guide* and *Oracle Developer Studio 12.5: Fortran User's Guide* for more information on working with C and Fortran. The following is the C driver (save it in a file named `driver.c`):

EXAMPLE 24 Calling Fortran From C

```
/*
 * a demo program that shows:
 * 1. how to call f95 subroutine from C, passing an array argument
 * 2. how to call single precision f95 function from C
 * 3. how to call double precision f95 function from C
 */

extern int      demo_one_(double *);
extern float    demo_two_(float *);
extern double   demo_three_(double *);

int main()
{
 double array[3][4];
 float f, g;
```

```
double x, y;
int i, j;

for (i = 0; i < 3; i++)
 for (j = 0; j < 4; j++)
  array[i][j] = i + 2*j;

g = 1.5;
y = g;

/* pass an array to a fortran function (print the array) */
demo_one_(&array[0][0]);
printf(" from the driver\n");
for (i = 0; i < 3; i++) {
 for (j = 0; j < 4; j++)
  printf("    array[%d][%d] = %e\n",
   i, j, array[i][j]);
 printf("\n");
}

/* call a single precision fortran function */
f = demo_two_(&g);
printf(
       " f = sin(g) from a single precision fortran function\n");
printf("    f, g: %8.7e, %8.7e\n", f, g);
printf("\n");

/* call a double precision fortran function */
x = demo_three_(&y);
printf(
 " x = sin(y) from a double precision fortran function\n");
printf("    x, y: %18.17e, %18.17e\n", x, y);

ieee_retrospective_();
return 0;
}
```

Save the Fortran subroutines in a file named `drivee.f`:

```
subroutine demo_one(array)
double precision array(4,3)
print *, 'from the fortran routine:'
do 10 i =1,4
 do 20 j = 1,3
  print *, '   array[', i, '][', j, '] = ', array(i,j)
20 continue
print *
10 continue
return
```

```
end

real function demo_two(number)
real number
demo_two = sin(number)
return
end

double precision function demo_three(number)
double precision number
demo_three = sin(number)
return
end
```

Perform the compilation and linking:

```
cc -c driver.c
f95 -c drivee.f
 demo_one:
 demo_two:
 demo_three:
f95 -o driver driver.o drivee.o
```

The output looks like this:

```
 from the fortran routine:
    array[ 1 ][ 1 ] =  0.0E+0
    array[ 1 ][ 2 ] =  1.0
    array[ 1 ][ 3 ] =  2.0

    array[ 2 ][ 1 ] =  2.0
    array[ 2 ][ 2 ] =  3.0
    array[ 2 ][ 3 ] =  4.0

    array[ 3 ][ 1 ] =  4.0
    array[ 3 ][ 2 ] =  5.0
    array[ 3 ][ 3 ] =  6.0

    array[ 4 ][ 1 ] =  6.0
    array[ 4 ][ 2 ] =  7.0
    array[ 4 ][ 3 ] =  8.0

 from the driver
    array[0][0] = 0.000000e+00
    array[0][1] = 2.000000e+00
    array[0][2] = 4.000000e+00
    array[0][3] = 6.000000e+00

    array[1][0] = 1.000000e+00
```

```
   array[1][1] = 3.000000e+00
   array[1][2] = 5.000000e+00
   array[1][3] = 7.000000e+00

   array[2][0] = 2.000000e+00
   array[2][1] = 4.000000e+00
   array[2][2] = 6.000000e+00
   array[2][3] = 8.000000e+00

f = sin(g) from a single precision fortran function
   f, g: 9.9749500e-01, 1.5000000e+00

x = sin(y) from a double precision fortran function
   x, y: 9.97494986604054446e-01, 1.50000000000000000e+00
```

A.4.3 Useful Debugging Commands

The following table shows examples of debugging commands for the SPARC architecture.

TABLE 35 Some Debugging Commands (SPARC)

Action	dbx	adb
Set breakpoint at function	`stop in myfunct`	`myfunct:b`
Set breakpoint at line number	`stop at 29`	n/a
Set breakpoint at absolute address	n/a	`23a8:b`
Set breakpoint at relative address	n/a	`main+0x40:b`
Run until breakpoint met	`run`	`:r`
Examine source code	`list`	`<pc,10?ia`
Examine an fp register : IEEE single precision	`print $f0`	`<f0=X`
Examine an fp register : decimal equivalent (Hex)	`print -fx $f0`	`<f0=f`
Examine an fp register : IEEE double precision	`print $f0f1`	`<f0=X; <f1=X`
Examine an fp register : decimal equivalent (Hex)	`print -flx $f0f1`	`<f0=F`
Examine all fp registers	`regs -F`	`$x for f0-f15` `$X for f16-f31`
Examine all registers	`regs`	`$r; $x; $X`
Examine fp status register	`print -fllx $fsr`	`<fsr=X`
Put single precision 1.0 in `f0`	`assign $f0=1.0`	`3f800000>f0`
Put double precision 1.0 in `f0/f1`	`assign $f0f1=1.0`	`3ff00000>f0; 0>f1`

Action	dbx	adb
Continue execution	`cont`	`:c`
Single step	`step (or next)`	`:s`
Exit the debugger	`quit`	`$q`

When displaying floating-point numbers, you should keep in mind that the size of registers is 32 bits, a single precision floating-point number occupies 32 bits (hence it fits in one register), and a double precision floating-point number occupies 64 bits (therefore two registers are used to hold a double precision number). In the hexadecimal representation, 32 bits corresponds to 8 hexadecimal digits. In the following snapshot of FPU registers displayed with `adb`, the display is organized as follows:

<name of fpu register> <IEEE hex value> <single precision> <double precision>

The third column holds the single precision decimal interpretation of the hexadecimal pattern shown in the second column. The fourth column interprets pairs of registers. For example, the fourth column of the `f11` line interprets `f10` and `f11` as a 64-bit IEEE double precision number.

Because `f10` and `f11` are used to hold a double precision value, the interpretation (on the `f10` line) of the first 32 bits of that value, `7ff00000`, as `+NaN`, is irrelevant. The interpretation of all 64 bits, `7ff00000 00000000`, as `+Infinity`, happens to be the meaningful translation.

The `adb` command `$x`, that was used to display the first 16 floating-point data registers, also displayed `fsr` (the floating-point status register):

```
$x
fsr    40020
f0 400921fb      +2.1426990e+00
f1 54442d18      +3.3702806e+12      +3.1415926535897931e+00
f2        2      +2.8025969e-45
f3        0      +0.0000000e+00      +4.2439915819305446e-314
f4 40000000      +2.0000000e+00
f5        0      +0.0000000e+00      +2.0000000000000000e+00
f6 3de0b460      +1.0971904e-01
f7        0      +0.0000000e+00      +1.2154188766544394e-10
f8 3de0b460      +1.0971904e-01
f9        0      +0.0000000e+00      +1.2154188766544394e-10
f10 7ff00000      +NaN
f11        0      +0.0000000e+00      +Infinity
f12 ffffffff      -NaN
f13 ffffffff      -NaN                  -NaN
f14 ffffffff      -NaN
f15 ffffffff      -NaN                  -NaN
```

The following table shows examples of debugging commands for the x86 architecture:

TABLE 36 Some Debugging Commands (x86)

Action	dbx	adb
Set breakpoint at function	`stop in myfunct`	`myfunct:b`
Set breakpoint at line number	`stop at 29`	n/a
Set breakpoint at absolute address	n/a	`23a8:b`
Set breakpoint at relative address	n/a	`main+0x40:b`
Run until breakpoint met	`run`	`:r`
Examine source code	`list`	`<pc,10?ia`

Examine fp registers	`print $st0`
	`...`
	`print $st7`
		`$x`

Examine all registers	`refs -F`	`$r`
Examine fp status register	`print -fx $fstat`	`<fstat=X`
Continue execution	`cont`	`:c`
Single step	`step (or next)`	`:s`
Exit the debugger	`quit`	`$q`

The following examples show two ways to set a breakpoint at the beginning of the code corresponding to a routine `myfunction` in `adb`. First you can use:

```
myfunction:b
```

Second, you can determine the absolute address that corresponds to the beginning of the piece of code corresponding to `myfunction`, and then set a break at that absolute address:

```
myfunction=X
   23a8
23a8:b
```

The main subroutine in a Fortran program compiled with `f95` is known as `MAIN_` to `adb`. To set a breakpoint at `MAIN_` in `adb`:

```
   MAIN_:b
```

When examining the contents of floating-point registers, the hex value shown by the `dbx` command `regs -F` is the base-16 representation, not the number's decimal representation. For SPARC-based systems, the `adb` commands `$x` and `$X` display both the hexadecimal representation, and the decimal value. For x86-based systems, the `adb` command `$x` displays only the decimal value. For SPARC-based systems, the double precision values show the decimal value next to the odd-numbered register.

Because the operating system disables the floating-point unit until it is first used by a process, you cannot modify the floating-point registers until they have been accessed by the program being debugged.

The corresponding output on x86 looks like the following:

```
$x
80387 chip is present.
cw      0x137f
sw      0x3920
cssel 0x17  ipoff 0x2d93               datasel 0x1f  dataoff 0x5740

 st[0]  +3.24999988079071044921875 e-1           VALID
 st[1]  +5.65391332434795490344196888 e73        EMPTY
 st[2]  +2.00000000000000008881784197            EMPTY
 st[3]  +1.80732183080704405560160047 e-1        EMPTY
 st[4]  +7.9180300235748291015625 e-1            EMPTY
 st[5]  +4.201639036693904927233234 e-13         EMPTY
 st[6]  +4.201639036693904927233234 e-13         EMPTY
 st[7]  +2.72249992132186946491856360            EMPTY
```

Note - For x86, `cw` is the control word and `sw` is the status word.

♦ ♦ ♦ APPENDIX B

SPARC Behavior and Implementation

This chapter discusses issues related to the floating-point units used in SPARC-based workstations and describes a way to determine which code generation flags are best suited for a particular workstation.

B.1 Floating-Point Hardware

This section lists a number of SPARC processors and describes the instruction sets and exception handling features they support.

The following tables list the hardware floating-point implementations used by recent SPARC systems:

TABLE 37 SPARC Systems Supported in Oracle Solaris 11 and later

Chip	Typical Systems	Best Code Generation Options
T1	T1000, T2000, T6300, CP3060	`-xarch=sparcvis2 -xchip=ultraT1`
T2	T5120, T5220, T6320, CP3260	`-xarch=sparcvis2 -xchip=ultraT2`
T2+	T5140, T5240, T5440	`-xarch=sparcvis2 -xchip=ultraT2plus`
T3	T3-1, T3-2, T3-4	`-xarch=sparcvis3 -xchip=T3`
T4	T4-1, T4-1B, T4-2, T4-4	`-xarch=sparc4 -xchip=T4`
T5	T5-1B, T5-2, T5-4, T5-8	`-xarch=sparc4 -xchip=T5`
M5	M5-32	`-xarch=sparc4 -xchip=M5`
M6	M6-32	`-xarch=sparc4 -xchip=M6`
M7	M7-32	`-xarch=sparc5 -xchip=M7`
SPARC64-VI	M4000, M5000, M8000, M9000	`-xarch=sparcfmaf -xchip=sparc64vi`
SPARC64-VII	M3000, M4000, M5000, M8000, M9000	`-xarch=spracima -xchip=sparc64vii`

Chip	Typical Systems	Best Code Generation Options
SPARC64-VII+	M3000, M4000, M5000, M8000, M9000	`-xarch=sparcima -xchip=sparc64viiplus`
SPARC64-X	M10-1, M10-4, M10-4S	`-xarch=sparcace -xchip=sparc64x`

TABLE 38 UltraSPARC Systems Supported in Oracle Solaris 10 Update 10 but not Oracle Solaris 11

UltraSPARC Chip	Typical Systems	Best Code Generation Options
`I`	Ex000	`-xarch=sparcvis -xchip=ultra`
`II`	Ex000, E10000	`-xarch=sparcvis -xchip=ultra2`
`IIi`	Ultra-5, Ultra-10	`-xarch=sparcvis -xchip=ultra2i`
`IIe`	Sun Blade 100	`-xarch=sparcvis -xchip=ultra2e`
`III`	Sun Blade 1000, 2000	`-xarch=sparcvis2 -xchip=ultra3`
`IIIi`	Sun Blade 1500, 2500	`-xarch=sparcvis2 -xchip=ultra3i`
`IIICu`	Sun Blade 1000, 2000	`-xarch=sparcvis2 -xchip=ultra3cu`
`IV`	V490, V890, Ex900, E20K, E25K	`-xarch=sparcvis2 -xchip=ultra4`
`IV+`	V490, V890, Ex900, E20K, E25K	`-xarch=sparcvis2 -xchip=ultra4plus`

Although it is not supported, programs compiled with Oracle Developer Studio 12.5 on Oracle Solaris 10 Update 10 or earlier Oracle Solaris releases often run on older SPARC systems that support earlier Oracle Developer Studio releases. To create an executable to test on such platforms, try compiling with the following options:

```
-m32 -xarch=generic -xchip=generic
```

For a supported solution, compile on the earliest Oracle Solaris release that you need to use and compile with the latest Oracle Developer Studio version supported on that Oracle Solaris release.

The last column in the preceding table shows the compiler flags to use to obtain the fastest code for each FPU. These flags control two independent attributes of code generation: the `-xarch` flag determines the instruction set the compiler may use, and the `-xchip` flag determines the assumptions the compiler will make about a processor's performance characteristics in scheduling the code. A program compiled with the default `-xarch` or with the explicit `-xarch=sparc` runs on any SPARC-based system listed above, although it might not take full advantage of the features of later processors. Likewise, a program compiled with a particular `-xchip` value runs on any SPARC-based system that supports the instruction set specified with `-xarch`, but it might run more slowly on systems with processors other than the one specified.

The UltraSPARC I, UltraSPARC II, UltraSPARC IIe, UltraSPARC IIi, UltraSPARC III, UltraSPARC IIIi, UltraSPARC IV, and UltraSPARC IV+ floating-point units implement the floating-point instruction set defined in the *SPARC Architecture Manual* Version 9 except for

the quad precision instructions; in particular, they provide 32 double precision floating-point registers. Compiling with `-xarch=sparc` enables the compiler to use all these features. These processors also provide extensions to the standard instruction set. Successive generations of these additional instructions are enabled by these `-xarch` values:

- `sparcvis`
- `sparcvis2`
- `sparcvis2`
- `sparc4`
- `sparc5`

Many of these additional instructions are rarely generated automatically by the compilers, but they can be used in assembly code.

The `-xarch` and `-xchip` options can be specified simultaneously using the `-xtarget` macro option. The `-xtarget` flag simply expands to a suitable combination of `-xarch`, `-xchip`, and `-xcache` flags. The default code generation option is `-xtarget=generic`. See the `cc`(1), `CC`(1), and `f95`(1) man pages and the *Oracle Developer Studio 12.5: Fortran User's Guide*, *Oracle Developer Studio 12.5: C User's Guide*, and *Oracle Developer Studio 12.5: C++ User's Guide* compiler manuals for more information including a complete list of `-xarch`, `-xchip`, and `-xtarget` values.

B.1.1 Floating-Point Status Register and Queue

All SPARC floating-point units, regardless of which version of the SPARC architecture they implement, provide a floating-point status register (FSR) that contains status and control bits associated with the FPU. All SPARC FPUs that implement deferred floating-point traps provide a floating-point queue (FQ) that contains information about currently executing floating-point instructions. The FSR can be accessed by user software to detect floating-point exceptions that have occurred and to control rounding direction, trapping, and nonstandard arithmetic modes. The FQ is used by the operating system kernel to process floating-point traps and is normally invisible to user software.

Software accesses the floating-point status register via `STFSR` and `LDFSR` instructions that store the FSR in memory and load it from memory, respectively. In SPARC assembly language, these instructions are written as follows:

```
st      %fsr, [addr]  ! store FSR at specified address
ld      [addr], %fsr  ! load FSR from specified address
```

The inline template file `libm.il` located in the directory containing the libraries supplied with the Sun Studio compilers contains examples showing the use of `STFSR` and `LDFSR` instructions.

The following figure shows the layout of bit fields in the floating-point status register.

FIGURE 7 SPARC Floating-Point Status Register

In versions 7 and 8 of the SPARC architecture, the FSR occupies 32 bits as shown. In version 9, the FSR is extended to 64 bits, of which the lower 32 match the figure; the upper 32 are largely unused, containing only three additional floating-point condition code fields.

In the figure, `res` refers to bits that are reserved, `ver` is a read-only field that identifies the version of the FPU, and `ftt` and `qne` are used by the system when it processes floating-point traps. The remaining fields are described in the following table

TABLE 39 Floating-Point Status Register Fields

Field	Contains
RM	rounding direction mode
TEM	trap enable modes
NS	nonstandard mode
fcc	floating-point condition code
aexc	accrued exception flags
cexc	current exception flags

The `RM` field holds two bits that specify the rounding direction for floating-point operations. The `NS` bit enables nonstandard arithmetic mode on SPARC FPUs that implement it; on others, this bit is ignored. The `fcc` field holds floating-point condition codes generated by floating-point compare instructions and used by branch and conditional move operations. Finally, the `TEM`, `aexc`, and `cexc` fields contain five bits that control trapping and record accrued and current exception flags for each of the five IEEE 754 floating-point exceptions. These fields are subdivided as shown in the following table.

TABLE 40 Exception Handling Fields

Field	Corresponding bits in register				
`TEM`, trap enable modes	NVM	OFM	UFM	DZM	NXM
	27	26	25	24	23

Field	Corresponding bits in register				
aexc, accrued exception flags	nva	ofa	ufa	dza	nxa
	9	8	7	6	5
cexc, current exception flags	nvc	ofc	ufc	dzc	nxc
	4	3	2	1	0

(The symbols NV, OF, UF, DZ, and NX above stand for the invalid operation, overflow, underflow, division-by-zero, and inexact exceptions respectively.)

B.1.2 Special Cases Requiring Software Support

In most cases, SPARC floating-point units execute instructions completely in hardware without requiring software support. There are four situations, however, when the hardware will not successfully complete a floating-point instruction:

- The floating-point unit is disabled.
- The instruction is not implemented by the hardware, such as quad precision instructions on any SPARC FPU.
- The hardware is unable to deliver the correct result for the instruction's operands.
- The instruction would cause an IEEE 754 floating-point exception and that exception's trap is enabled.

In each situation, the initial response is the same: the process traps to the system kernel, which determines the cause of the trap and takes the appropriate action. The term “trap” refers to an interruption of the normal flow of control. In the first three situations, the kernel emulates the trapping instruction in software. Note that the emulated instruction can also incur an exception whose trap is enabled.

In the first three situations above, if the emulated instruction does not incur an IEEE floating-point exception whose trap is enabled, the kernel completes the instruction. If the instruction is a floating-point compare, the kernel updates the condition codes to reflect the result; if the instruction is an arithmetic operation, it delivers the appropriate result to the destination register. It also updates the current exception flags to reflect any (untrapped) exceptions raised by the instruction, and it “or”s those exceptions into the accrued exception flags. It then arranges to continue execution of the process at the point at which the trap was taken.

When an instruction executed by hardware or emulated by the kernel software incurs an IEEE floating-point exception whose trap is enabled, the instruction is not completed. The destination register, floating-point condition codes, and accrued exception flags are unchanged, the current exception flags are set to reflect the particular exception that caused the trap, and the kernel sends a `SIGFPE` signal to the process.

The following pseudo-code summarizes the handling of floating-point traps. Note that the `aexc` field can normally only be cleared by software.

```
FPop provokes a trap;
if trap type is fp_disabled, unimplemented_FPop, or
  unfinished_FPop then
    emulate FPop;
texc = all IEEE exceptions generated by FPop;
if (texc and TEM) = 0 then
    f[rd]  = fp_result;  // if fpop is an arithmetic op
    fcc = fcc_result;  // if fpop is a compare
    cexc = texc;
    aexc = (aexc or texc);
else
    cexc = trapped IEEE exception generated by FPop;
    throw SIGFPE;
```

A program will encounter severe performance degradation when many floating-point instructions must be emulated by the kernel. The relative frequency with which this happens can depend on several factors including the type of trap.

Under normal circumstances, the `fp_disabled` trap should occur only once per process. The system kernel disables the floating-point unit when a process is first started, so the first floating-point operation executed by the process will cause a trap. After processing the trap, the kernel enables the floating-point unit, and it remains enabled for the duration of the process. (It is possible to disable the floating-point unit for the entire system, but this is not recommended and is done only for kernel or hardware debugging purposes.)

An `unimplemented_FPop` trap occurs any time the floating-point unit encounters an instruction it does not implement. Since most current SPARC floating-point units implement at least the instruction set defined by the *SPARC Architecture Manual* Version 8, except for the quad precision instructions, and the Oracle Developer Studio compilers do not generate quad precision instructions, this type of trap should not occur on most systems compiled with `-xarch=sparc`.

The remaining two trap types, `unfinished_FPop` and trapped IEEE exceptions, are usually associated with special computational situations involving NaNs, infinities, and subnormal numbers.

B.1.2.1 IEEE Floating-Point Exceptions, NaNs, and Infinities

When a floating-point instruction encounters an IEEE floating-point exception whose trap is enabled, the instruction is not completed. Instead the system delivers a `SIGFPE` signal to

the process. If the process has established a `SIGFPE` signal handler, that handler is invoked, and otherwise, the process aborts. Since trapping is most often enabled for the purpose of aborting the program when an exception occurs, either by invoking a signal handler that prints a message and terminates the program or by resorting to the system default behavior when no signal handler is installed, most programs do not incur many trapped IEEE floating-point exceptions. As described in Chapter 4, "Exceptions and Exception Handling", however, it is possible to arrange for a signal handler to supply a result for the trapping instruction and continue execution. Note that severe performance degradation can result if many floating-point exceptions are trapped and handled in this way.

Some SPARC floating-point units will also trap on at least some cases involving infinite or NaN operands or IEEE floating-point exceptions even when trapping is disabled or an instruction would not cause an exception whose trap is enabled. This happens when the hardware does not support such special cases. Instead it generates an `unfinished_FPop` trap and leaves the kernel emulation software to complete the instruction. Different SPARC FPUs vary as to the conditions that result in an `unfinished_FPop` trap. For example, most early SPARC FPUs trap on all IEEE floating-point exceptions regardless of whether trapping is enabled, while UltraSPARC FPUs can trap pessimistically when a floating-point exception's trap is enabled and the hardware is unable to determine whether an instruction would raise that exception. But any recent SPARC processors handle all exceptional cases in hardware and never generate an `unfinished_FPop` traps.

Since most `unfinished_FPop` traps occur in conjunction with floating-point exceptions, a program can avoid incurring an excessive number of these traps by employing exception handling: testing the exception flags, trapping and substituting results, or aborting on exceptions. Take care to balance the cost of handling exceptions with that of allowing exceptions to result in `unfinished_FPop` traps.

B.1.2.2 Subnormal Numbers and Nonstandard Arithmetic

The most common situations in which some SPARC floating-point units will trap with an `unfinished_FPop` involve subnormal numbers. Many older SPARC floating-point units will trap whenever a floating-point operation involves subnormal operands or must generate a nonzero subnormal result, i.e., a result that incurs gradual underflow. Because underflow is somewhat rare but difficult to program around, and because the accuracy of underflowed intermediate results often has little effect on the overall accuracy of the final result of a computation, the SPARC architecture defines a nonstandard arithmetic mode that provides a way for a user to avoid the performance degradation associated with `unfinished_FPop` traps involving subnormal numbers.

The SPARC architecture does not precisely define nonstandard arithmetic mode. It merely states that when this mode is enabled, processors that support it might produce results that do not conform to the IEEE 754 standard. However, all existing SPARC implementations that

support this mode use it to disable gradual underflow, replacing all subnormal operands and results with zero.

Not all SPARC implementations provide a nonstandard mode. SPARC implementations that do not support this mode simply ignore it, so numerical and exception results are the same in nonstandard more. Gradual underflow incurs no performance loss on these processors.

To determine whether gradual underflows are affecting the performance of a program, you should first determine whether underflows are occurring at all and then check how much system time is used by the program. To determine whether underflows are occurring, you can use the math library function `ieee_retrospective()` to see if the underflow exception flag is raised when the program exits. Fortran programs call `ieee_retrospective()` by default. C and C++ programs need to call `ieee_retrospective()` explicitly prior to exit. If any underflows have occurred, `ieee_retrospective()` prints a message similar to the following:

```
Note: IEEE floating-point exception flags raised:
 Inexact; Underflow;
See the Numerical Computation Guide, ieee_flags(3M)
```

If the program encounters underflows, you might want to determine how much system time the program is using by timing the program execution with the `time` command:

```
demo% /bin/time myprog > myprog.output

real 305.3
user 32.4
sys 271.9
```

If the system time, the third figure of the previous output, is unusually high, multiple underflows might be the cause. If so, and if the program does not depend on the accuracy of gradual underflow, you can enable nonstandard mode for better performance.

There are two ways to do this. First, you can compile with the `-fns` flag, which is implied as part of the macros `-fast` and `-fnonstd`, to enable nonstandard mode at program startup. Second, the value-added math library `libsunmath` provides two functions to enable and disable nonstandard mode, respectively: calling `nonstandard_arithmetic()` enables nonstandard mode (if it is supported), while calling `standard_arithmetic()` restores IEEE behavior. The C and Fortran syntax for calling these functions is as follows:

C, C++	`nonstandard_arithmetic();` `standard_arithmetic();`
Fortran	`call nonstandard_arithmetic()`

```
call standard_arithmetic()
```

Caution - Since nonstandard arithmetic mode defeats the accuracy benefits of gradual underflow, you should use it with caution. For more information about gradual underflow, see Chapter 2, “IEEE Arithmetic”.

B.1.2.3 Nonstandard Arithmetic and Kernel Emulation

On SPARC floating-point units that implement nonstandard mode, enabling this mode causes the hardware to treat subnormal operands as zero and flush subnormal results to zero. The kernel software that is used to emulate trapped floating-point instructions, however, does not implement nonstandard mode, in part because the effect of this mode is undefined and implementation-dependent and because the added cost of handling gradual underflow is negligible compared to the cost of emulating a floating-point operation in software.

If a floating-point operation that would be affected by nonstandard mode is interrupted (for example, it has been issued but not completed when a context switch occurs or another floating-point instruction causes a trap), it will be emulated by kernel software using standard IEEE arithmetic. Thus, under unusual circumstances, a program running in nonstandard mode might produce slightly varying results depending on system load. This behavior has not been observed in practice. It would affect only those programs that are very sensitive to whether one particular operation out of millions is executed with gradual underflow or with abrupt underflow.

B.2 *fpversion*(1) Function: Finding Information About the FPU

The *fpversion* utility distributed with the compilers identifies the installed CPU and estimates the processor and system bus clock speeds. `fpversion` determines the CPU and FPU types by interpreting the identification information stored by the CPU and FPU. It estimates their clock speeds by timing a loop that executes simple instructions that run in a predictable amount of time. The loop is executed many times to increase the accuracy of the timing measurements. For this reason, `fpversion` is not instantaneous. It can take several seconds to run.

`fpversion` also reports the best `-xtarget` code generation option to use for the host system.

On a T4-2 server, `fpversion` displays information similar to the following. There might be variations due to differences in timing or machine configuration.

```
demo% fpversion
 A SPARC-based CPU is available.
 Kernel says CPU's clock rate is 1500.0 MHz.
 Kernel says main memory's clock rate is 150.0 MHz.

 Sun-4 floating-point controller version 0 found.
 An UltraSPARC chip is available.

 Use "-xtarget=T4 -xcache=16/32/4/8:128/32/8/8:4096/64/16/64" code-generation option.

Hostid = hardware_host_id
```

See the *fpversion*(1) manual page for more information.

♦ ♦ ♦ APPENDIX C

x86 Behavior and Implementation

This appendix discusses x86/x64 and SPARC compatibility issues related to the floating-point units used in x86/x64 based systems.

C.1 Code Generation for Supported Systems

Oracle Solaris supports many systems from Oracle, Sun, and other system vendors, that contain x86 processors from Intel, AMD, and other chip vendors. A particular Oracle Solaris release supports a number of specific systems containing such chips. For a particular Oracle Solaris release, see its corresponding Hardware Compatibility List.

Oracle Solaris 11 supports x86 processors that support 64-bit addressing. Oracle Solaris 10 Update 10 supports those 64-bit processors and many 32-bit-only x86 processors with hardware floating-point and 120 MHz or faster clock rates.

Compile with the `-m32 -xarch=generic -xchip=generic` flags to generate code that is satisfactory for the largest number of systems. The following table lists some specific code generation options for a few typical Oracle and Sun x86 systems:

System	Code Generation Options
Ultra 20	`-xarch=sse2a -xchip=opteron`
X2200	`-xarch=amdsse4a -xchip=amdfam10`
X6250	`-xarch=sse3 -xchip=core2`
X4170	`-xarch=aes -xchip=westmere`
X2-4	`-xarch=sse4_2 -xchip=nehalem`
X3-2	`-xarch=avx -xchip=sandybridge`
X4-2 X4-4	`-xarch=avx_i -xchip=ivybridge`
?	`-xarch=avx2 -xchip=haswell`

There are hundreds of distinct x86 chips, each with complicated nomenclature.

Using `cc -dryrun -native` is the best way to find out what the compiler would do to optimize a particular system. When generating code intended for a few varied x86 systems, using options for the oldest system is often satisfactory for all.

C.2 Differences from SPARC

Oracle Developer Studio compilers generate code that usually performs similarly on SPARC and x86. However, be aware of the following important differences on x86-based systems:

- The x87 floating-point registers are 80 bits wide. Because intermediate results of arithmetic computations can be in double extended (80-bit) precision when the x87 floating-point register stack is in use, computation results can differ. The `-fstore` flag minimizes these discrepancies. However, using the `-fstore` flag introduces a penalty in performance. While Oracle Developer Studio 12.5 does not use x87 registers by default for single and double precision expression evaluation, they are used if `-xarch=386` is specified, or if the x87 hardware transcendental instructions are used, or if double-extended variables are used.
- Each time a single or double precision floating-point number is loaded onto the x87 floating-point register stack or stored into memory, a conversion to or from double extended (80-bit) precision occurs. Thus loads and stores of floating-point numbers can cause exceptions. With `-m32`, floating-point subroutine operands and results are passed in x87 registers.
- When the x87 floating-point register stack is in use, gradual underflow is implemented in hardware with microcode assist; there is no nonstandard mode.
- There is no `fpversion` utility.
- The double extended (80-bit) format admits certain bit patterns that do not represent any floating-point values (see Table 8, “Values Represented by Bit Patterns (x86),” on page 28). The hardware generally treats these "unsupported formats" as signaling NaNs, but the math libraries are not consistent in their handling of such representations. Since these bit patterns are never generated by the hardware, they can only be created by invalid memory references. such as reading beyond the end of an array, or from explicit coercions of data in memory from one type to another, via C's union construct, for example. Therefore, in most numerical programs, these bit patterns do not arise.

♦ ♦ ♦ APPENDIX D

Addendum to *What Every Computer Scientist Should Know About Floating-Point Arithmetic*

Every reader of this *Numerical Computation Guide* will find helpful the paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys (see http://dl.acm.org/citation.cfm?id=103163).

The following is the abstract:

Floating-point arithmetic is considered as esoteric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems: Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating point standard, and concludes with examples of how computer system builders can better support floating point.

This appendix is not part of the published Goldberg paper. It has been added to clarify certain points and correct possible misconceptions about the IEEE standard that the reader might infer from the paper. This material was not written by David Goldberg, but it appears here with his permission. Standards 754-1985 and 854-1987 have been replaced by 754-2008 which specifies both binary and decimal floating-point arithmetic. This doesn't affect the Goldberg paper.

This appendix specifically discusses “D.1 Differences Among IEEE 754 Implementations” on page 168. This topic covers the following subtopics:

D.1 Differences Among IEEE 754 Implementations

The Goldberg paper has shown that floating-point arithmetic must be implemented carefully, since programmers might depend on its properties for the correctness and accuracy of their programs. In particular, the IEEE standard requires a careful implementation, and it is possible to write useful programs that work correctly and deliver accurate results only on systems that conform to the standard. The reader might be tempted to conclude that such programs should be portable to all IEEE systems. Indeed, portable software would be easier to write if the remark "When a program is moved between two machines and both support IEEE arithmetic, then if any intermediate result differs, it must be because of software bugs, not from differences in arithmetic," were true.

Unfortunately, the IEEE standard does not guarantee that the same program will deliver identical results on all conforming systems. Most programs will actually produce different results on different systems for a variety of reasons. For one, many programs use elementary functions supplied by a system library, and the standard doesn't completely specify these functions. The 1985 standard did not completely specify conversion of numbers between decimal and binary formats, and did not specify transcendental functions at all.

Many programmers might not realize that even a program that uses only the numeric formats and operations prescribed by the IEEE standard can compute different results on different systems. In fact, the authors of the standard intended to allow different implementations to obtain different results. Their intent is evident in the definition of the term *destination* in the IEEE 754 standard: "A destination may be either explicitly designated by the user or implicitly supplied by the system (for example, intermediate results in subexpressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's format and the operands' values." (IEEE 754-1985, p. 7) In other words, the IEEE standard requires that each result be rounded correctly to the precision of the destination into which it will be placed, but the standard does not require that the precision of that destination be determined by a user's program. Thus, different systems might deliver their results to destinations with different precisions, causing the same program to produce different results, sometimes dramatically so, even though those systems all conform to the standard.

Several of the examples in the referenced paper depend on some knowledge of the way floating-point arithmetic is rounded. In order to rely on examples such as these, a programmer must be able to predict how a program will be interpreted, and in particular, on an IEEE system, what the precision of the destination of each arithmetic operation might be. Alas, the loophole in the IEEE standard's definition of destination undermines the programmer's ability to know how a program will be interpreted. Consequently, several of the examples given in the Goldberg paper, when implemented as apparently portable programs in a high-level language, might not work correctly on IEEE systems that normally deliver results to destinations with a different precision than the programmer expects. Other examples might work, but proving that they work might lie beyond the average programmer's ability.

In this appendix, existing implementations of IEEE 754 arithmetic are classified based on the precisions of the destination formats they normally use. Some examples are reviewed from the paper to show that delivering results in a wider precision than a program expects can cause it to compute wrong results even though it is provably correct when the expected precision is used. One of the proofs is revised in the paper to illustrate the intellectual effort required to cope with unexpected precision even when it doesn't invalidate our programs. These examples show that despite all that the IEEE standard prescribes, the differences it allows among different implementations can prevent you from writing portable, efficient numerical software whose behavior we can accurately predict. To develop such software, then, programming languages and environments must first be created that limit the variability the IEEE standard permits to enable you to express the floating-point semantics upon which their programs depend. The 2008 version of the 1985 standard makes such recommendations for programming languages.

D.1.1 Current IEEE 754 Implementations

Current implementations of IEEE 754 arithmetic can be divided into two groups distinguished by the degree to which they support different floating-point formats in hardware. Extended-based systems, exemplified by the Intel x86 family of processors and compiled with the `-xarch=386` option, provide full support for an extended double precision format but only partial support for single and double precision: they provide instructions to load or store data in single and double precision, converting it on-the-fly to or from the extended double format, and they provide special modes (not the default) in which the results of arithmetic operations are rounded to single or double precision even though they are kept in registers in extended double format. Motorola 68000 series processors round results to both the precision and range of the single or double formats in these modes. Intel x86 and compatible processors round results to the precision of the single or double formats but retain the same range as the extended double format. Single/double systems, including most RISC processors, provide full support for single and double precision formats but no support for an IEEE-compliant extended double precision format. The x86 SSE2 extensions provide a single/double system in the SSE2 registers, and still support extended precision in extended-precision registers.

To see how a computation might behave differently on an extended-based system than on a single/double system, consider a C version of the example "Systems Aspects" from the Goldberg paper as follows:

```
int main() {
    double  q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

The constants 3.0 and 7.0 are interpreted as double precision floating-point numbers, and the expression 3.0/7.0 inherits the `double` data type. On a single/double system, the expression will be evaluated in double precision since that is the most efficient format to use. Thus, `q` will be assigned the value 3.0/7.0 rounded correctly to double precision. In the next line, the expression 3.0/7.0 will again be evaluated in double precision, and the result will be equal to the value just assigned to `q`, so the program will print “Equal” as expected.

On an extended-based system, even though the expression 3.0/7.0 has type `double`, the quotient will be computed in a register in extended double format, and thus in the default mode, it will be rounded to extended double precision. When the resulting value is assigned to the variable `q`, however, it might then be stored in memory, and since `q` is declared `double`, the value will be rounded to double precision. In the next line, the expression 3.0/7.0 can again be evaluated in extended precision yielding a result that differs from the double precision value stored in `q`, causing the program to print “Not equal”.

Other outcomes are possible, too: the compiler could decide to store and thus round the value of the expression 3.0/7.0 in the second line before comparing it with `q`, or it could keep `q` in a register in extended precision without storing it. An optimizing compiler might evaluate the expression 3.0/7.0 at compile time, perhaps in double precision or perhaps in extended double precision. With one x86 compiler, the program prints “Equal” when compiled with optimization and “Not Equal” when compiled for debugging. Finally, some compilers for extended-based systems automatically change the rounding precision mode to cause operations producing results in registers to round those results to single or double precision, albeit possibly with a wider range. Thus, on these systems, the behavior of the program cannot be predicted simply by reading its source code and applying a basic understanding of IEEE 754 arithmetic. Neither can the hardware or the compiler be accused of failing to provide an IEEE 754 compliant environment. The hardware has delivered a correctly rounded result to each destination, as it is required to do, and the compiler has assigned some intermediate results to destinations that are beyond the user's control, as it is allowed to do.

The rest of this section describes x86 behavior compiled with `-xarch=386`, which was the default in Oracle Developer Studio 12 and earlier releases. In Oracle Developer Studio 12.5 , the default is `-xarch=sse2`.

D.1.2 Pitfalls in Computations on Extended-Based Systems

Conventional wisdom maintains that extended-based systems must produce results that are at least as accurate, if not more accurate than those delivered on single/double systems, since the former always provide at least as much precision and often more than the latter. Trivial examples as well as more subtle programs based on the examples discussed in the following

section show that this wisdom is naive at best: some apparently portable programs, which are indeed portable across single/double systems, deliver incorrect results on extended-based systems precisely because the compiler and hardware conspire to occasionally provide more precision than the program expects.

Current programming languages make it difficult for a program to specify the precision it expects. As the section “Languages and Compilers” in the Goldberg paper mentions, many programming languages don't specify that each occurrence of an expression like `10.0*x` in the same context should evaluate to the same value. Some languages, such as Ada, were influenced in this respect by variations among different arithmetics prior to the IEEE standard. More recently, languages like ANSI C have been influenced by standard-conforming extended-based systems. In fact, the ANSI C standard explicitly allows a compiler to evaluate a floating-point expression to a precision wider than that normally associated with its type. As a result, the value of the expression `10.0*x` can vary in ways that depend on a variety of factors: whether the expression is immediately assigned to a variable or appears as a subexpression in a larger expression; whether the expression participates in a comparison; whether the expression is passed as an argument to a function, and if so, whether the argument is passed by value or by reference; the current precision mode; the level of optimization at which the program was compiled; the precision mode and expression evaluation method used by the compiler when the program was compiled; and so on.

Language standards are not entirely to blame for the vagaries of expression evaluation. Extended-based systems run most efficiently when expressions are evaluated in extended precision registers whenever possible, yet values that must be stored are stored in the narrowest precision required. Constraining a language to require that `10.0*x` evaluate to the same value everywhere would impose a performance penalty on those systems. Unfortunately, allowing those systems to evaluate `10.0*x` differently in syntactically equivalent contexts imposes a penalty of its own on programmers of accurate numerical software by preventing them from relying on the syntax of their programs to express their intended semantics.

The following example explores whether real programs depend on the assumption that a given expression always evaluates to the same value. Recall the algorithm presented in Theorem 4 of the Goldberg paper for computing ln(1 + x), written in Fortran:

```
real function log1p(x)
real x
if (1.0 + x .eq. 1.0) then
   log1p = x
else
   log1p = log(1.0 + x) * x / ((1.0 + x) - 1.0)
endif
return
```

On an extended-based system, a compiler might evaluate the expression `1.0 + x` in the third line in extended precision and compare the result with `1.0`. When the same expression is

passed to the log function in the sixth line, however, the compiler might store its value in memory, rounding it to single precision. Thus, if `x` is not so small that `1.0 + x` rounds to `1.0` in extended precision but small enough that `1.0 + x` rounds to `1.0` in single precision, then the value returned by `log1p(x)` will be zero instead of `x`, and the relative error will be one—rather larger than 5??. Similarly, suppose the rest of the expression in the sixth line, including the reoccurrence of the subexpression `1.0 + x`, is evaluated in extended precision. In that case, if `x` is small but not quite small enough that `1.0 + x` rounds to `1.0` in single precision, then the value returned by `log1p(x)` can exceed the correct value by nearly as much as `x`, and again the relative error can approach one. For a concrete example, take `x` to be $2^{-24} + 2^{-47}$, so `x` is the smallest single precision number such that `1.0 + x` rounds up to the next larger number, $1 + 2^{-23}$. Then `log(1.0 + x)` is approximately 2^{-23}. Because the denominator in the expression in the sixth line is evaluated in extended precision, it is computed exactly and delivers `x`, so `log1p(x)` returns approximately 2^{-23}, which is nearly twice as large as the exact value.

This actually happens with at least one compiler. When the preceding code is compiled by the Sun WorkShop Compilers 4.2.1 Fortran 77 compiler for x86 systems using the `-O` optimization flag, the generated code computes `1.0 + x` exactly as described. As a result, the function delivers zero for `log1p(1.0e-10)` and `1.19209E-07` for `log1p(5.97e-8)`.

For the algorithm of Theorem 4 to work correctly, the expression `1.0 + x` must be evaluated the same way each time it appears. The algorithm can fail on extended-based systems only when `1.0 + x` is evaluated to extended double precision in one instance and to single or double precision in another. Since `log` is a generic intrinsic function in Fortran, a compiler could evaluate the expression `1.0 + x` in extended precision throughout, computing its logarithm in the same precision, but evidently you cannot assume that the compiler will do so. You can also imagine a similar example involving a user-defined function. In that case, a compiler could still keep the argument in extended precision even though the function returns a single precision result, but few if any existing Fortran compilers do this, either. You might therefore attempt to ensure that `1.0 + x` is evaluated consistently by assigning it to a variable. Unfortunately, if you declare that variable `real`, you might still be foiled by a compiler that substitutes a value kept in a register in extended precision for one appearance of the variable and a value stored in memory in single precision for another. Instead, you would need to declare the variable with a type that corresponds to the extended precision format. Standard FORTRAN 77 does not provide a way to do this, and while Fortran 95 offers the `SELECTED_REAL_KIND` mechanism for describing various formats, it does not explicitly require implementations that evaluate expressions in extended precision to allow variables to be declared with that precision. In short, there is no portable way to write this program in standard Fortran that is guaranteed to prevent the expression `1.0 + x` from being evaluated in a way that invalidates the proof.

There are other examples that can malfunction on extended-based systems even when each subexpression is stored and thus rounded to the same precision. The cause is double-rounding. In the default precision mode, an extended-based system will initially round each result to extended double precision. If that result is then stored to double precision, it is rounded again.

The combination of these two roundings can yield a value that is different than what would have been obtained by rounding the first result correctly to double precision. This can happen when the result as rounded to extended double precision is a “halfway case”, i.e., it lies exactly halfway between two double precision numbers, so the second rounding is determined by the round-ties-to-even rule. If this second rounding rounds in the same direction as the first, the net rounding error will exceed half a unit in the last place. Note, though, that double-rounding only affects double precision computations. You can prove that the sum, difference, product, or quotient of two p-bit numbers, or the square root of a p-bit number, rounded first to q bits and then to p bits gives the same value as if the result were rounded just once to p bits provided $q \geq 2p + 2$. Thus, extended double precision is wide enough that single precision computations don't suffer double-rounding.

Some algorithms that depend on correct rounding can fail with double-rounding. In fact, even some algorithms that don't require correct rounding and work correctly on a variety of machines that don't conform to IEEE 754 can fail with double-rounding. The most useful of these are the portable algorithms for performing simulated multiple precision arithmetic mentioned in Theorem 5, of the Goldberg paper. For example, the procedure described in Theorem 6 for splitting a floating-point number into high and low parts doesn't work correctly in double-rounding arithmetic: try to split the double precision number $2^{52} + 3 \times 2^{26} - 1$ into two parts each with at most 26 bits. When each operation is rounded correctly to double precision, the high order part is $2^{52} + 2^{27}$ and the low order part is $2^{26} - 1$, but when each operation is rounded first to extended double precision and then to double precision, the procedure produces a high order part of $2^{52} + 2^{28}$ and a low order part of $-2^{26} - 1$. The latter number occupies 27 bits, so its square can't be computed exactly in double precision. It would still be possible to compute the square of this number in extended double precision, but the resulting algorithm would no longer be portable to single/double systems. Also, later steps in the multiple precision multiplication algorithm assume that all partial products have been computed in double precision. Handling a mixture of double and extended double variables correctly would make the implementation significantly more expensive.

Likewise, portable algorithms for adding multiple precision numbers represented as arrays of double precision numbers can fail in double-rounding arithmetic. These algorithms typically rely on a technique similar to Kahan's summation formula. As the informal explanation of the summation formula given the section “Errors In Summation” of the Goldberg paper suggests, if `s` and `y` are floating-point variables with $|s| \geq |y|$ and you compute the following:

```
t = s + y;
e = (s - t) + y;
```

then in most arithmetics, `e` recovers exactly the round-off error that occurred in computing `t`. This technique doesn't work in double-rounded arithmetic, however: if `s` = $2^{52} + 1$ and `y` = $1/2 - 2^{-54}$, then `s + y` rounds first to $2^{52} + 3/2$ in extended double precision, and this value rounds to $2^{52} + 2$ in double precision by the round-ties-to-even rule; thus the net rounding error in computing `t` is $1/2 + 2^{-54}$, which is not representable exactly in double precision and

so can't be computed exactly by the expression shown above. Here again, it would be possible to recover the roundoff error by computing the sum in extended double precision, but then a program would have to do extra work to reduce the final outputs back to double precision, and double-rounding could afflict this process, too. For this reason, although portable programs for simulating multiple precision arithmetic by these methods work correctly and efficiently on a wide variety of machines, they do not work as advertised on extended-based systems.

Finally, some algorithms that at first sight appear to depend on correct rounding might in fact work correctly with double-rounding. In these cases, the cost of coping with double-rounding lies not in the implementation but in the verification that the algorithm works as advertised. To illustrate, the following variant of Theorem 7 is proven:

D.1.2.1 Theorem 7'

If m and n are integers representable in IEEE 754 double precision with $|m| < 2^{52}$ and n has the special form $n = 2^i + 2^j$, then $(m \oslash n) \otimes n = m$, provided both floating-point operations are either rounded correctly to double precision or rounded first to extended double precision and then to double precision.

Note - In this theorem, $\oslash$ and $\otimes$ represent computed division and computed multiplication respectively.

D.1.2.2 Proof

Assume without loss that $m > 0$. Let $q = m / n$. Scaling by powers of two, we can consider an equivalent setting in which $2^{52} \le m < 2^{53}$ and likewise for q, so that both m and q are integers whose least significant bits occupy the units place (i.e., $\mathrm{ulp}(m) = \mathrm{ulp}(q) = 1$). Before scaling, we assumed $m < 2^{52}$, so after scaling, m is an even integer. Also, because the scaled values of m and q satisfy $m/2 < q < 2m$, the corresponding value of n must have one of two forms depending on which of m or q is larger: if $q < m$, then evidently $1 < n < 2$, and since n is a sum of two powers of two, $n = 1 + 2^{-k}$ for some k; similarly, if $q > m$, then $1/2 < n < 1$, so $n = 1/2 + 2^{-(k+1)}$. As n is the sum of two powers of two, the closest possible value of n to one is $n = 1 + 2^{-52}$. Because $m/(1 + 2^{-52})$ is no larger than the next smaller double precision number less than m, we can't have $q = m$.)

Let e denote the rounding error in computing q, so that $q = m/n + e$, and the computed value $q \otimes n$ will be the (once or twice) rounded value of $m + ne$. Consider first the case in which each floating-point operation is rounded correctly to double precision. In this case, $|e| < 1/2$. If n has the form $1/2 + 2^{-(k+1)}$, then $ne = nq - m$ is an integer multiple of $2^{-(k+1)}$ and $|ne| < 1/4 + 2^{-(k+1)}$. This implies that $|ne| \le 1/4$. Recall that the difference between m and the next larger representable number is 1 and the difference between m and the next smaller representable

number is either 1 if $m > 2^{52}$ or 1/2 if $m = 2^{52}$. Thus, as $|ne| \le 1/4$, $m + ne$ will round to m. (Even if $m = 2^{52}$ and $ne = -1/4$, the product will round to m by the round-ties-to-even rule.) Similarly, if n has the form $1 + 2^{-k}$, then ne is an integer multiple of 2^{-k} and $|ne| < 1/2 + 2^{-(k+1)}$; this implies $|ne| \le 1/2$. We can't have $m = 2^{52}$ in this case because m is strictly greater than q, so m differs from its nearest representable neighbors by ??1. Thus, as $|ne| \le 1/2$, again $m + ne$ will round to m. (Even if $|ne| = 1/2$, the product will round to m by the round-ties-to-even rule because m is even.) This completes the proof for correctly rounded arithmetic.

In double-rounding arithmetic, it might still happen that q is the correctly rounded quotient (even though it was actually rounded twice), so $|e| < 1/2$ as above. In this case, we can appeal to the arguments of the previous paragraph provided we consider the fact that $q \otimes n$ will be rounded twice. To account for this, note that the IEEE standard requires that an extended double format carry at least 64 significant bits, so that the numbers m ?? 1/2 and m ?? 1/4 are exactly representable in extended double precision. Thus, if n has the form $1/2 + 2^{-(k+1)}$, so that $|ne| \le 1/4$, then rounding $m + ne$ to extended double precision must produce a result that differs from m by at most 1/4, and as noted above, this value will round to m in double precision. Similarly, if n has the form $1 + 2^{-k}$, so that $|ne| \le 1/2$, then rounding $m + ne$ to extended double precision must produce a result that differs from m by at most 1/2, and this value will round to m in double precision. Recall that $m > 2^{52}$ in this case.

Finally, we are left to consider cases in which q is not the correctly rounded quotient due to double-rounding. In these cases, we have $|e| < 1/2 + 2^{-(d+1)}$ in the worst case, where d is the number of extra bits in the extended double format. All existing extended-based systems support an extended double format with exactly 64 significant bits; for this format, $d = 64 - 53 = 11$. Because double-rounding only produces an incorrectly rounded result when the second rounding is determined by the round-ties-to-even rule, q must be an even integer. Thus if n has the form $1/2 + 2^{-(k+1)}$, then $ne = nq - m$ is an integer multiple of 2^{-k}, and $|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}$.

If $k \le d$, this implies $|ne| \le 1/4$. If $k > d$, we have $|ne| \le 1/4 + 2^{-(d+2)}$. In either case, the first rounding of the product will deliver a result that differs from m by at most 1/4, and by previous arguments, the second rounding will round to m. Similarly, if n has the form $1 + 2^{-k}$, then ne is an integer multiple of $2^{-(k-1)}$, and $|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}$.

If $k \le d$, this implies $|ne| \le 1/2$. If $k > d$, we have $|ne| \le 1/4 + 2^{-(d+1)}$. In either case, the first rounding of the product will deliver a result that differs from m by at most 1/2, and again by previous arguments, the second rounding will round to m. ♦

The preceding proof shows that the product can incur double-rounding only if the quotient does, and even then, it rounds to the correct result. The proof also shows that extending our reasoning to include the possibility of double-rounding can be challenging even for a program with only two floating-point operations. For a more complicated program, it might be impossible

to systematically account for the effects of double-rounding, not to mention more general combinations of double and extended double precision computations.

D.1.3 Programming Language Support for Extended Precision

The preceding examples should not be taken to suggest that extended precision in itself is harmful. Many programs can benefit from extended precision when the programmer is able to use it selectively. Unfortunately, current programming languages do not provide sufficient means for a programmer to specify when and how extended precision should be used. To indicate what support is needed, consider the ways in which you might want to manage the use of extended precision.

In a portable program that uses double precision as its nominal working precision, there are five ways we might want to control the use of a wider precision:

1. Compile to produce the fastest code, using extended precision where possible on extended-based systems. Clearly most numerical software does not require more of the arithmetic than that the relative error in each operation is bounded by the “machine epsilon”. When data in memory are stored in double precision, the machine epsilon is usually taken to be the largest relative roundoff error in that precision, since the input data are assumed to have been rounded when they were entered and the results will likewise be rounded when they are stored. Thus, while computing some of the intermediate results in extended precision might yield a more accurate result, extended precision is not essential. In this case, we might prefer that the compiler use extended precision only when it will not appreciably slow the program and use double precision otherwise.
2. Use a format wider than double if it is reasonably fast and wide enough, otherwise resort to something else. Some computations can be performed more easily when extended precision is available, but they can also be carried out in double precision with only somewhat greater effort. Consider computing the Euclidean norm of a vector of double precision numbers. By computing the squares of the elements and accumulating their sum in an IEEE 754 extended double format with its wider exponent range, we can trivially avoid premature underflow or overflow for vectors of practical lengths. On extended-based systems, this is the fastest way to compute the norm. On single/double systems, an extended double format would have to be emulated in software, if one were supported at all, and such emulation would be much slower than simply using double precision, testing the exception flags to determine whether underflow or overflow occurred, and if so, repeating the computation with explicit scaling. Note that to support this use of extended precision, a language must provide both an indication of the widest available format that is reasonably fast, so that a program can choose which method to use, and environmental parameters that indicate the precision and range of each format, so that the program can verify that the widest fast format is wide enough, e.g., that it has wider range than double.

3. Use a format wider than double even if it has to be emulated in software. For more complicated programs than the Euclidean norm example, the programmer might want to avoid the need to write two versions of the program and instead rely on extended precision even if it is slow. Again, the language must provide environmental parameters so that the program can determine the range and precision of the widest available format.
4. Don't use a wider precision; round results correctly to the precision of the double format, albeit possibly with extended range. For programs that are most easily written to depend on correctly rounded double precision arithmetic, including some of the examples mentioned above, a language must provide a way for the programmer to indicate that extended precision must not be used, even though intermediate results can be computed in registers with a wider exponent range than double. Intermediate results computed in this way can still incur double-rounding if they underflow when stored to memory: if the result of an arithmetic operation is rounded first to 53 significant bits, then rounded again to fewer significant bits when it must be denormalized, the final result might differ from what would have been obtained by rounding just once to a denormalized number. Of course, this form of double-rounding is highly unlikely to affect any practical program adversely.
5. Round results correctly to both the precision and range of the double format. This strict enforcement of double precision would be most useful for programs that test either numerical software or the arithmetic itself near the limits of both the range and precision of the double format. Such careful test programs tend to be difficult to write in a portable way; they become even more difficult and error prone when they must employ dummy subroutines and other tricks to force results to be rounded to a particular format. Thus, a programmer using an extended-based system to develop robust software that must be portable to all IEEE 754 implementations would quickly come to appreciate being able to emulate the arithmetic of single/double systems without extraordinary effort.

No current language supports all five of these options. In fact, few languages have attempted to give the programmer the ability to control the use of extended precision at all. One notable exception is the ISO/IEC 9899:1999 Programming Languages - C standard, a major revision to the C language.

The C99 standard allows an implementation to evaluate expressions in a format wider than that normally associated with their type, but the C99 standard recommends using one of only three expression evaluation methods. The three recommended methods are characterized by the extent to which expressions are “promoted” to wider formats, and the implementation is encouraged to identify which method it uses by defining the preprocessor macro `FLT_EVAL_METHOD`: if `FLT_EVAL_METHOD` is 0, each expression is evaluated in a format that corresponds to its type; if `FLT_EVAL_METHOD` is 1, `float` expressions are promoted to the format that corresponds to `double`; and if `FLT_EVAL_METHOD` is 2, `float` and `double` expressions are promoted to the format that corresponds to `long double`. (An implementation is allowed to set `FLT_EVAL_METHOD` to –1 to indicate that the expression evaluation method is indeterminable.) The C99 standard also requires that the `<math.h>` header file define the types `float_t` and `double_t`, which are at least as wide as `float` and `double`, respectively, and are

intended to match the types used to evaluate `float` and `double` expressions. For example, if `FLT_EVAL_METHOD` is 2, both `float_t` and `double_t` are `long double`. Finally, the C99 standard requires that the `<float.h>` header file define preprocessor macros that specify the range and precision of the formats corresponding to each floating-point type.

The combination of features required or recommended by the C99 standard supports some of the five options listed above but not all. For example, if an implementation maps the `long double` type to an extended double format and defines `FLT_EVAL_METHOD` to be 2, the programmer can reasonably assume that extended precision is relatively fast, so programs like the Euclidean norm example can simply use intermediate variables of type `long double` (or `double_t`). On the other hand, the same implementation must keep anonymous expressions in extended precision even when they are stored in memory (e.g., when the compiler must spill floating-point registers), and it must store the results of expressions assigned to variables declared `double` to convert them to double precision even if they could have been kept in registers. Thus, neither the `double` nor the `double_t` type can be compiled to produce the fastest code on current extended-based hardware.

Likewise, the C99 standard provides solutions to some of the problems illustrated by the examples in this section but not all. A C99 standard version of the `log1p` function is guaranteed to work correctly if the expression `1.0 + x` is assigned to a variable (of any type) and that variable used throughout. A portable, efficient C99 standard program for splitting a double precision number into high and low parts, however, is more difficult: how can we split at the correct position and avoid double-rounding if we cannot guarantee that `double` expressions are rounded correctly to double precision? One solution is to use the `double_t` type to perform the splitting in double precision on single/double systems and in extended precision on extended-based systems, so that in either case the arithmetic will be correctly rounded. Theorem 14 says that we can split at any bit position provided we know the precision of the underlying arithmetic, and the `FLT_EVAL_METHOD` and environmental parameter macros should give us this information.

The following fragment shows one possible implementation:

```
#include <math.h>
#include <float.h>

#if (FLT_EVAL_METHOD==2)
#define PWR2  LDBL_MANT_DIG - (DBL_MANT_DIG/2)
#elif ((FLT_EVAL_METHOD==1) || (FLT_EVAL_METHOD==0))
#define PWR2  DBL_MANT_DIG - (DBL_MANT_DIG/2)
#else
#error FLT_EVAL_METHOD unknown!
#endif

...
    double   x, xh, xl;
```

```
double_t m;

m = scalbn(1.0, PWR2) + 1.0;  // 2**PWR2 + 1
xh = (m * x) - ((m * x) - x);
xl = x - xh;
```

To find this solution, you must know that `double` expressions can be evaluated in extended precision, that the ensuing double-rounding problem can cause the algorithm to malfunction, and that extended precision can be used instead according to Theorem 14. A more obvious solution is simply to specify that each expression be rounded correctly to double precision. On extended-based systems, this merely requires changing the rounding precision mode, but unfortunately, the C99 standard does not provide a portable way to do this. Early drafts of the Floating-Point C Edits, the working document that specified the changes to be made to the C90 standard to support floating-point, recommended that implementations on systems with rounding precision modes provide `fegetprec` and `fesetprec` functions to get and set the rounding precision, analogous to the `fegetround` and `fesetround` functions that get and set the rounding direction. This recommendation was removed before the changes were made to the C99 standard.

Coincidentally, the C99 standard's approach to supporting portability among systems with different integer arithmetic capabilities suggests a better way to support different floating-point architectures. Each C99 standard implementation supplies an `<stdint.h>` header file that defines those integer types the implementation supports, named according to their sizes and efficiency: for example, `int32_t` is an integer type exactly 32 bits wide, `int_fast16_t` is the implementation's fastest integer type at least 16 bits wide, and `intmax_t` is the widest integer type supported. One can imagine a similar scheme for floating-point types: for example, `float53_t` could name a floating-point type with exactly 53 bit precision but possibly wider range, `float_fast24_t` could name the implementation's fastest type with at least 24 bit precision, and `floatmax_t` could name the widest reasonably fast type supported. The fast types could allow compilers on extended-based systems to generate the fastest possible code subject only to the constraint that the values of named variables must not appear to change as a result of register spilling. The exact width types would cause compilers on extended-based systems to set the rounding precision mode to round to the specified precision, allowing wider range subject to the same constraint. Finally, `double_t` could name a type with both the precision and range of the IEEE 754 double format, providing strict double evaluation. Together with environmental parameter macros named accordingly, such a scheme would readily support all five options described above and allow programmers to indicate easily and unambiguously the floating-point semantics their programs require.

Must language support for extended precision be so complicated? On single/double systems, four of the five options listed above coincide, and there is no need to differentiate fast and exact width types. Extended-based systems, however, pose difficult choices: they support neither pure double precision nor pure extended precision computation as efficiently as a mixture of the two, and different programs call for different mixtures. Moreover, the choice of when to use

extended precision should not be left to compiler writers, who are often tempted by benchmarks and sometimes told outright by numerical analysts to regard floating-point arithmetic as “inherently inexact” and therefore neither deserving nor capable of the predictability of integer arithmetic. Instead, the choice must be presented to programmers, and they will require languages capable of expressing their selection.

D.1.4 Conclusion

The foregoing remarks are not intended to disparage extended-based systems but to expose several fallacies, the first being that all IEEE 754 systems must deliver identical results for the same program. We have focused on differences between extended-based systems and single/double systems, but there are further differences among systems within each of these families. For example, some single/double systems provide a single instruction to multiply two numbers and add a third with just one final rounding. This operation, called a fused multiply-add, can cause the same program to produce different results across different single/double systems, and, like extended precision, it can even cause the same program to produce different results on the same system depending on whether and when it is used. A fused multiply-add can also foil the splitting process of Theorem 6, although it can be used in a non-portable way to perform multiple precision multiplication without the need for splitting. Even though the IEEE standard didn't anticipate such an operation, it nevertheless conforms: the intermediate product is delivered to a “destination” beyond the user's control that is wide enough to hold it exactly, and the final sum is rounded correctly to fit its single or double precision destination.

The idea that IEEE 754 prescribes precisely the result a given program must deliver is nonetheless appealing. Many programmers like to believe that they can understand the behavior of a program and prove that it will work correctly without reference to the compiler that compiles it or the computer that runs it. In many ways, supporting this belief is a worthwhile goal for the designers of computer systems and programming languages. Unfortunately, when it comes to floating-point arithmetic, the goal is virtually impossible to achieve. The authors of the IEEE standards knew that, and they didn't attempt to achieve it. As a result, despite nearly universal conformance to most of the IEEE 754 standard throughout the computer industry, programmers of portable software must continue to cope with unpredictable floating-point arithmetic.

If programmers are to exploit the features of IEEE 754, they will need programming languages that make floating-point arithmetic predictable. The C99 standard improves predictability to some degree at the expense of requiring programmers to write multiple versions of their programs, one for each `FLT_EVAL_METHOD`. Whether future languages will choose instead to allow programmers to write a single program with syntax that unambiguously expresses the extent to which it depends on IEEE 754 semantics remains to be seen. Existing extended-based systems threaten that prospect by tempting us to assume that the compiler and the hardware can know better than the programmer how a computation should be performed on a given system.

That assumption is the second fallacy: the accuracy required in a computed result depends not on the machine that produces it but only on the conclusions that will be drawn from it, and of the programmer, the compiler, and the hardware, at best only the programmer can know what those conclusions may be.

♦ ♦ ♦ APPENDIX E

Standards Compliance

The Oracle Developer Studio compiler products together with the header files and libraries in the Solaris 10 operating environment support multiple standards including: System V Interface Definition Edition 3 (SVID), X/Open, ANSI C (C90), POSIX.1-2001 (SUSv3) and ISO C (C99). (See *standards*(5) for a more complete discussion.) Some of these standards allow implementations to vary in certain respects. In some cases, the specifications of these standards conflict. For the math libraries, the variances and conflicts are primarily related to special cases and exceptions. This appendix documents the behavior of the functions in`libm` in these cases and discusses conditions under which a C program can expect behavior that conforms to each standard. The final section of this appendix documents the conformance of the Sun Studio C and Fortran language products to LIA-1.

E.1 `libm` Special Cases

Table 41, “Special Cases and `libm` Functions,” on page 184 lists all of the cases in which two or more of the standards mentioned above specify conflicting behavior for functions in `libm`. Which behavior a C program will observe depends on the compiler flags that are used when the program is compiled and linked. Possible behaviors include raising floating-point exceptions, calling the user-supplied function `matherr` with information about the special case that occurred and the value to be returned (see *matherr*(3M)), printing a message on the standard error file, and setting the global variable `errno` (see *intro*(2) and *perror*(3C)).

The first column in Table 41, “Special Cases and `libm` Functions,” on page 184 defines the special case. The second column shows the value to which `errno` will be set if it is set at all. The possible values for `errno` are defined in `<errno.h>`; the only two values used by the math library are `EDOM` for domain errors and `ERANGE` for range errors. When the second column shows both `EDOM` and `ERANGE`, the value to which `errno` is set is determined by the relevant standard as described below and shown in the fourth or fifth column. The third column shows the error code that will be indicated in any error message that is printed. The fourth, fifth, and sixth columns show the function values that will nominally be returned as defined by various

standards. In some cases, a user-supplied matherr routine can override these values and supply another return value.

The specific responses to these special cases are determined by the compiler flags specified when a program is linked as follows. If either -xlibmieee or -xc99=lib is specified, then when any of the special cases in Table 41, "Special Cases and libm Functions," on page 184 occurs, any appropriate floating-point exceptions are raised, and the function value listed in the sixth column of the table is returned.

If neither -xlibmieee nor -xc99=lib is used, then the behavior depends on the language conformance flag specified when the program is linked.

Specifying the -Xa flag selects X/Open conformance. When any of the special cases in the table occurs, any appropriate floating-point exceptions are raised, errno is set, and the function value listed in the fifth column of the table is returned. If a user-defined matherr routine is supplied, the behavior is undefined. Note that -Xa is the default when no other language conformance flag is given.

Specifying the -Xc flag selects strict C90 conformance. When a special case occurs, any appropriate floating-point exceptions are raised, errno is set, and the function value listed in the fifth column of the table is returned. matherr is not invoked in this case.

Finally, specifying either the -Xs or the -Xt flag selects SVID conformance. When a special case occurs, any appropriate floating-point exceptions are raised, matherr is called, and if matherr returns zero, then errno is set and an error message is printed. The function value listed in the fourth column of the table is returned unless it is overridden by matherr.

See the *cc*(1) manual page and *Oracle Developer Studio 12.5: C User's Guide* for more information about the -xc99, -Xa, -Xc, -Xs, and -Xt flags.

TABLE 41 Special Cases and libm Functions

Function	errno	error message	SVID	X/Open, C90	IEEE, C99, SUSv3
acos(|x|>1)	EDOM	DOMAIN	0.0	0.0	NaN
acosh(x<1)	EDOM	DOMAIN	NaN	NaN	NaN
asin(|x|>1)	EDOM	DOMAIN	0.0	0.0	NaN
atan2(+/-0,+/-0)	EDOM	DOMAIN	0.0	0.0	+/- 0.0,+/- pi
atanh(|x|>1)	EDOM	DOMAIN	NaN	NaN	NaN
atanh(+/-1)	EDOM/ERANGE	SING	+/- HUGE1 (EDOM)	+/- HUGE_VAL2 (ERANGE)	+/- infinity
cosh overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
exp overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
exp underflow	ERANGE	-	0.0	0.0	0.0

Function	errno	error message	SVID	X/Open, C90	IEEE, C99, SUSv3
fmod(x,0)	EDOM	DOMAIN	x	NaN	NaN
gamma(0 or -integer)	EDOM	SING	HUGE	HUGE_VAL	infinity
gamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
hypot overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
j0(X_TLOSS3<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
j1(X_TLOSS<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
jn(n,X_TLOSS<|x|< inf)	ERANGE	TLOSS	0.0	0.0	computed answer
ldexp overflow	ERANGE	-	+-infinity	+/-infinity	+/-infinity
ldexp underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
lgamma(0 or -integer)	EDOM	SING	HUGE	HUGE_VAL	infinity
lgamma overflow	ERANGE	-	HUGE	HUGE_VAL	infinity
log(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log10(0)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log10(x<0)	EDOM	DOMAIN	-HUGE	-HUGE_VAL	NaN
log1p(-1)	EDOM/ERANGE	SING	-HUGE (EDOM)	-HUGE_VAL (ERANGE)	-infinity
log1p(x<-1)	EDOM	DOMAIN	NaN	NaN	NaN
logb(0)	EDOM	-	-HUGE_VAL	-HUGE_VAL	-infinity
nextafter overflow	ERANGE	-	+-HUGE_VAL	+/-HUGE_VAL	+/-infinity
pow(0,0)	EDOM	DOMAIN	0.0	1.0 (no error)	1.0 (no error)
pow(NaN,0)	EDOM	DOMAIN	NaN	NaN	1.0 (no error)
pow(0,x<0)	EDOM	DOMAIN	0.0	-HUGE_VAL	+/-infinity
pow(x<0, non-integer)	EDOM	DOMAIN	0.0	NaN	NaN
pow overflow	ERANGE	-	+/-HUGE	+/-HUGE_VAL	+/-infinity
pow underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
remainder(x,0) or remainder(inf,y)	EDOM	DOMAIN	NaN	NaN	NaN
scalb overflow	ERANGE	-	+-HUGE_VAL	+/-HUGE_VAL	+/-infinity
scalb underflow	ERANGE	-	+/-0.0	+/-0.0	+/-0.0
scalb(0,+inf) or scalb (inf,-inf)	EDOM/ERANGE	-	NaN (ERANGE)	NaN (EDOM)	NaN
scalb(|x|>0,+inf)	ERANGE	-	+-infinity	+/-infinity (no error)	+/-infinity (no error)
scalb(|x|<inf, -inf)	ERANGE	-	+/-0.0	+/-0.0 (no error)	+/-0.0 (no error)

Function	errno	error message	SVID	X/Open, C90	IEEE, C99, SUSv3
sinh overflow	ERANGE	-	+/- HUGE	+/- HUGE_VAL	+/- infinity
sqrt(x<0)	EDOM	DOMAIN	0.0	NaN	NaN
y0(0)	EDOM	DOMAIN	- HUGE	- HUGE_VAL	- infinity
y0(x<0)	EDOM	DOMAIN	- HUGE	- HUGE_VAL	NaN
y0(X_TLOSS<x<inf)	ERANGE	TLOSS	0.0	0.0	correct answer
y1(0)	EDOM	DOMAIN	- HUGE	- HUGE_VAL	- infinity
y1(x<0)	EDOM	DOMAIN	- HUGE	- HUGE_VAL	NaN
y1(X_TLOSS<x<inf)	ERANGE	TLOSS	0.0	0.0	correct answer
yn(n,0)	EDOM	DOMAIN	- HUGE	- HUGE_VAL	- infinity
yn(n,x<0)	EDOM	DOMAIN	- HUGE	- HUGE_VAL	NaN
yn(n,X_TLOSS<x< inf)	ERANGE	TLOSS	0.0	0.0	correct answer

Notes:

1. HUGE is defined in <math.h>. SVID requires that HUGE be equal to MAXFLOAT, which is approximately 3.4e+38.
2. HUGE_VAL is defined in <iso/math_iso.h>, which is included in <math.h>. HUGE_VAL evaluates to infinity.
3. X_TLOSS is defined in <values.h>.

E.1.1 Other Compiler Flags Affecting Standard Conformance

The compiler flags listed above directly select which of several standards will be followed in handling the special cases listed in Table 41, “Special Cases and libm Functions,” on page 184. Other compiler flags can indirectly affect whether a program observes the behavior described above.

First, both the -xlibmil and -xlibmopt flags substitute faster implementations of some of the functions in libm. These faster implementations do not conform to SVID, X/Open, or C90. Neither do they set errno or call matherr. They do, however, raise floating-point exceptions as appropriate and deliver the results specified by IEEE 754 and/or C99. Similar comments apply to the -xvector flag, since it can cause the compiler to transform calls to standard math functions into calls to vector math functions.

Second, the -xbuiltin flag allows the compiler to treat the standard math functions defined in <math.h> as intrinsic and substitute inline code for better performance. The substitute code

might not conform to SVID, X/Open, C90, or C99. It need not set errno, call matherr, or raise floating-point exceptions.

Third, when the C preprocessor token __MATHERR_ERRNO_DONTCARE is defined, a number of #pragma directives in <math.h> are compiled. These directives tell the compiler to assume that the standard math functions have no side effects. Under this assumption, the compiler can reorder calls to the math functions and references to global data such as errno or data that might be modified by a user-supplied matherr routine so as to violate the expected behavior described above. For example, consider the code fragment:

```
#include <errno.h>
#include <math.h>

...
errno = 0;
x = acos(2.0);
if (errno) {
    printf("error\n");
}
```

If this code is compiled with __MATHERR_ERRNO_DONTCARE defined, the compiler might assume that errno is not modified by the call to acos and transform the code accordingly, removing the call to printf entirely.

Note that the -fast macro flag includes the flags -xbuiltin, -xlibmil, -xlibmopt, and -D__MATHERR_ERRNO_DONTCARE.

Finally, since all of the math functions in libm raise floating-point exceptions as needed, running a program with trapping on those exceptions enabled will generally result in behavior other than that specified by the standards listed above. Thus, the -ftrap compiler flag can also affect standard conformance.

E.1.2 Additional Notes on C99 Conformance

C99 specifies two possible methods by which an implementation can handle special cases such as those in Table 41, "Special Cases and libm Functions," on page 184. An implementation indicates which of the two methods it supports by defining the identifier math_errhandling to evaluate to an integer expression having the value MATH_ERRNO (1) or MATH_ERREXCEPT (2) or the bitwise "or" of these. (These values are defined in <math.h>.) If the expression (math_errhandling & MATH_ERRNO) is nonzero, then the implementation handles cases in which the argument of a function lies outside its mathematical domain by setting errno to EDOM and handles cases in which the result value of a function would underflow, overflow,

or equal infinity exactly by setting `errno` to `ERANGE`. If the expression `(math_errhandling & MATH_ERREXCEPT)` is nonzero, then the implementation handles cases in which the argument of a function lies outside its mathematical domain by raising the invalid operation exception and handles cases in which the result value of a function would underflow, overflow or equal infinity exactly by raising the underflow, overflow, or division-by-zero exception, respectively.

On Oracle Solaris, `<math.h>` defines `math_errhandling` to be `MATH_ERREXCEPT`. Although the functions listed in Table 41, “Special Cases and `libm` Functions,” on page 184 may perform other actions for the special cases shown there, all `libm` functions — including the `float` and `long double` functions, complex functions, and additional functions specified by C99 — respond to special cases by raising floating-point exceptions. This is the only method for handling special cases that is supported uniformly for all C99 functions.

Finally, note that there are three functions for which either C99 or SUSv3 requires different behavior from the Oracle Solaris default. The differences are summarized in the following table. The table lists only the `double` version of each function, but the differences apply to the `float` and `long double` versions as well. In each case, the SUSv3 specification is followed when a program is linked with `-xc99=lib` and the Solaris default is followed otherwise.

TABLE 42 Solaris and C99/SUSv3 Differences

Function	Solaris behavior	C99/SUSv3 behavior
`pow`	`pow(1.0, +/-inf)` returns NaN	`pow(1.0, +/-inf)` returns 1
	`pow(-1.0, +/-inf)` returns NaN	`pow(-1.0, +/-inf)` returns 1
	`pow(1.0, NaN)` returns NaN	`pow(1.0, NaN)` returns 1
`logb`	`logb(subnormal)` returns Emin	`logb(x)` = `ilogb(x)` when x is subnormal
`ilogb`	`ilogb(+/-0)`, `ilogb(+/-inf)`,	`ilogb(+/-0)`, `ilogb(+/-inf)`,
	`ilogb(NaN)` raise no exceptions	`ilogb(NaN)` raise invalid operation

E.2 LIA-1 Conformance

In this section, LIA-1 refers to ISO/IEC 10967-1:1994 Information Technology - Language Independent Arithmetic - Part 1: Integer and floating-point arithmetic.

The C and Fortran 95 compilers (`cc` and `f95`) contained in the Sun Studio compilers release conform to LIA-1 in the following senses (paragraph letters correspond to those in LIA‑1 section 8):

E.2.1 a. TYPES (LIA 5.1):

The LIA-1 conformant types are C `int` and Fortran `INTEGER`. Other types can conform as well, but they are not specified here. Further specifications for specific languages await language bindings to LIA-1 from the cognizant language standards organizations.

E.2.2 b. PARAMETERS (LIA 5.1):

```
#include <values.h> /* defines MAXINT */
#define TRUE 1
#define FALSE 0
#define BOUNDED TRUE
#define MODULO TRUE
#define MAXINT 2147483647
#define MININT  ‾2147483648
 logical bounded, modulo
 integer maxint, minint
 parameter (bounded = .TRUE.)
 parameter (modulo = .TRUE.)
 parameter (maxint = 2147483647)
 parameter (minint =  ‾2147483648)
```

E.2.3 d. DIV/REM/MOD (LIA 5.1.3):

C / and %, and Fortran / and mod(), provide `DIVtI(x,y)` and `REMtI(x,y)`. Also, `modaI(x,y)` is available with the following code:

```
int modaI(int x, int y) {
 int t = x % y;
 if (y < 0 && t > 0)
 t  ‾= y;
 else if (y > 0 && t < 0)
 t += y;
  return t;
 }
```

It is also available with the following code:

```
 integer function modaI(x, y)
 integer x, y, t
 t = mod(x, y)
 if (y .lt. 0 .and. t .gt. 0) t = t  ‾  y
```

```
if (y .gt. 0 .and. t .lt. 0) t = t + y
modaI = t
return
end
```

E.2.4 i. NOTATION (LIA 5.1.3):

The following table shows the notation by which the LIA integer operations can be realized.

TABLE 43 LIA-1 Conformance - Notation

LIA	C	Fortran if different
addI(x,y)	x+y	n/a
subI(x,y)	x - y	n/a
mulI(x,y)	x*y	n/a
divtI(x,y)	x/y	n/a
remtI(x,y)	x%y	mod(x,y)
modaI(x,y)	see above	n/a
negI(x)	- x	n/a

absI(x)	#include <stdlib.h>
	abs(x)
		abs(x)

signI(x)	#define signI(x) (x > 0 ? 1 : (x < 0 ? - 1 : 0))	see below
eqI(x,y)	x==y	x.eq.y
neqI(x,y)	x!=y	x.ne.y
lssI(x,y)	x<y	x.lt.y
leqI(x,y)	x<=y	x.le.y
gtrI(x,y)	x>y	x.gt.y
geqI(x,y)	x>=y	x.ge.y

The following code shows the Fortran notation for `signI(x)`.

```
integer function signi(x)
integer x, t
if (x .gt. 0) t=1
if (x .lt. 0) t= - 1
if (x .eq. 0) t=0
return
end
```

E.2.5 j. EXPRESSION EVALUATION:

By default, when no optimization is specified, expressions are evaluated in `int` (C) or `INTEGER` (Fortran) precision. Parentheses are respected. The order of evaluation of associative unparenthesized expressions such as a + b + c or a * b * c is not specified.

E.2.6 k. METHOD OF OBTAINING PARAMETERS:

Include the definitions in "E.2.2 b. PARAMETERS (LIA 5.1):" on page 189 in your source code.

E.2.7 n. NOTIFICATION:

Integer exceptions are x/0 and x%0 or mod(x,0). By default, these exceptions generate `SIGFPE`. When no signal handler is specified for `SIGFPE`, the process terminates and dumps memory.

E.2.8 o. SELECTION MECHANISM:

`signal`(3) or `signal`(3F) can be used to enable user exception handling for `SIGFPE`.

♦ ♦ ♦ APPENDIX F

References

The following manuals provide more information about SPARC® floating-point hardware:

- UltraSPARC Architecture 2005 (base ISA for T1) (http://www.oracle.com/technetwork/systems/opensparc/1537734)
- UltraSPARC Architecture 2007 (base ISA for T2, T2+, T3) (http://www.oracle.com/technetwork/systems/hardware/usparcarchdoc2007-329425.pdf)
- Oracle SPARC Architecture 2011 (base ISA for T4, T5, M5, M6) (http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/140521-ua2011-d096-p-ext-2306580.pdf)

The remaining references are organized by chapter. Information on obtaining Standards documents and test programs is included at the end.

F.1 Chapter 2: "IEEE Arithmetic"

Cody et al., "A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic," IEEE Computer, August 1984.

Coonen, J.T., "An Implementation Guide to a Proposed Standard for Floating Point Arithmetic", Computer, Vol. 13, No. 1, Jan. 1980, pp 68-79.

Demmel, J., "Underflow and the Reliability of Numerical Software", SIAM J. *Scientific Statistical Computing*, Volume 5 (1984), 887-919.

Hough, D., "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic", Computer, Vol. 13, No. 1, Jan. 1980, pp 70-74.

Kahan, W., and Coonen, J.T., "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments", published in The Relationship between Numerical Computation and Programming Languages, Reid, J.K., (editor), North-Holland Publishing Company, 1982.

Kahan, W., “Implementation of Algorithms”, Computer Science Technical Report No. 20, University of California, Berkeley CA, 1973. Available from National Technical Information Service, NTIS Document No. AD–769 124 (339 pages), 1-703-487-4650 (ordinary orders) or 1-800-336-4700 (rush orders.)

Karpinski, R., “Paranoia: a Floating-Point Benchmark”, Byte, February 1985.

Knuth, D.E., The Art of Computer Programming, Vol.2: Semi-Numerical Algorithms, Addison-Wesley, Reading, Mass, 1969, p 195.

Linnainmaa, S., “Combatting the effects of Underflow and Overflow in Determining Real Roots of Polynomials”, SIGNUM Newsletter 16, (1981), 11-16.

Rump, S.M., “How Reliable are Results of Computers?”, translation of “Wie zuverlassig sind die Ergebnisse unserer Rechenanlagen?”, Jahrbuch Uberblicke Mathematik 1983, pp 163-168, C Bibliographisches Institut AG 1984.

Sterbenz, P, Floating-Point Computation, Prentice-Hall, Englewood Cliffs, NJ, 1974. (Out of print; most university libraries have copies.)

Stevenson, D. et al., Cody, W., Hough, D. Coonen, J., various papers proposing and analyzing a draft standard for binary floating-point arithmetic, IEEE Computer, March 1981.

The Proposed IEEE Floating-Point Standard, special issue of the ACM SIGNUM Newsletter, October 1979.

F.2 Chapter 3: “The Math Libraries”

Cody, William J. and Waite, William, Software Manual for the Elementary Functions, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1980.

Coonen, J.T., Contributions to a Proposed Standard for Binary Floating-Point Arithmetic, PhD Dissertation, University of California, Berkeley, 1984.

Tang, Peter Ping Tak, Some Software Implementations of the Functions Sin and Cos, Technical Report ANL-90/3, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1990.

Tang, Peter Ping Tak, Table-driven Implementations of the Exponential Function EXPM1 in IEEE Floating-Point Arithmetic, Preprint MCS-P125-0290, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February, 1990.

Tang, Peter Ping Tak, Table-driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic, ACM Transactions on Mathematical Software, Vol. 15, No. 2, June 1989, pp 144-157 communication, July 18, 1988.

Tang, Peter Ping Tak, Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic, preprint MCS-P55-0289, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, February 1989 (to appear in ACM Trans. on Math. Soft.)

Park, Stephen K. and Miller, Keith W., “Random Number Generators: Good Ones Are Hard To Find”, Communications of the ACM, Vol. 31, No. 10, October 1988, pp 1192 - 1201.

F.3 Chapter 4: “Exceptions and Exception Handling”

Coonen, J.T, “Underflow and the Denormalized Numbers”, Computer, 14, No. 3, March 1981, pp 75-87.

Demmel, J., and X. Li, “Faster Numerical Algorithms via Exception Handling”, IEEE Trans. Comput. Vol. 48, No. 8, August 1994, pp 983-992.

Kahan, W., “A Survey of Error Analysis”, Information Processing 71, North-Holland, Amsterdam, 1972, pp 1214-1239.

F.4 Standards

American National Standard for Information Systems ISO/IEC 9899:1999 Programming Languages - C (C99), American National Standards Institute, 1430 Broadway, New York, NY 10018.

IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Std 754-2008, published by the Institute of Electrical and Electronics Engineers, Inc, 3 Park Avenue, New York, NY 10016, 2008.

IEEE Standard Glossary of Mathematics of Computing Terminology, ANSI/IEEE Std 1084-1986, published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1986.

IEEE Standard Portable Operating System Interface for Computer Environments (POSIX®), IEEE Std 1003.1-1988, The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017.

System V Application Binary Interface (ABI), AT&T (1-800-432-6600), 1989.

SPARC System V ABI Supplement (SPARC ABI), AT&T (1-800-432-6600), 1990.

System V Interface Definition, 3rd edition, (SVID89, or SVID Issue 3), Volumes I–IV, Part number 320-135, AT&T (1-800-432-6600), 1989.

X/OPEN Portability Guide, Set of 7 Volumes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1989.

F.5 Test Programs

A number of test programs for floating-point arithmetic and math libraries are available from Netlib in the ucbtest package. These programs include versions of `Paranoia`, Z. Alex Liu's Berkeley Elementary Function test program, the IEEE test vectors, and programs based on number-theoretic methods developed by Prof. W. Kahan that generate hard test cases for correctly rounded multiplication, division, and square root.

ucbtest is located at `http://www.netlib.org/fp/ucbtest.tgz`.

Glossary

A-E

abrupt underflow When a floating-point operation underflows, always return zero, even if the result would be in the range of subnormal numbers.

accuracy A measure of how well one number approximates another. For example, the accuracy of a computed result often reflects the extent to which errors in the computation cause it to differ from the mathematically exact result. Accuracy can be expressed in terms of significant digits (e.g., “The result is accurate to six digits”) or more generally in terms of the preservation of relevant mathematical properties (e.g., “The result has the correct algebraic sign”).

biased exponent The sum of the base-2 exponent and a constant (bias) chosen to make the stored exponent's range non-negative. For example, the exponent of 2^{-100} is stored in IEEE single precision format as (-100) + (single precision bias of 127) = 27.

binade The interval between any two consecutive powers of two.

chaining A hardware feature of some pipeline architectures that allows the result of an operation to be used immediately as an operand for a second operation, simultaneously with the writing of the result to its destination register. The total cycle time of two chained operations is less than the sum of the stand-alone cycle times for the instructions. For example, the TI 8847 supports chaining of consecutive `fadd`, `fsub`, and `fmul` (of the same precision). Chained `faddd`/`fmuld` requires 12 cycles, while consecutive unchained `faddd`/`fmuld` requires 17 cycles.

common exceptions The three floating point exceptions overflow, invalid, and division-by-zero are collectively referred to as the common exceptions for the purposes of `ieee_flags`(3m) and `ieee_handler`(3m). They are called common exceptions because they are commonly trapped as errors.

context switch In multitasking operating systems, such as the Oracle Solaris operating system, processes run for a fixed time quantum. At the end of the time quantum, the CPU receives a signal from the timer, interrupts the currently running process, and prepares to run a new process. The CPU saves the registers for the old process, and then loads the registers for the new process. Switching from the old process state to the new is known as a context switch. Time spent

switching contexts is system overhead; the time required depends on the number of registers, and on whether there are special instructions to save the registers associated with a process.

default result The value that is delivered as the result of a floating-point operation that caused an exception when no other handling has been specified for that exception.

denormalized number Older nomenclature for subnormal number.

double precision Using two words to represent a number in order to keep or increase precision. On SPARC® workstations, double precision is the 64-bit IEEE double precision.

exception An arithmetic exception arises when an attempted atomic arithmetic operation has no result that is acceptable universally. The meanings of atomic and acceptable vary with time and place.

exponent The component of a floating-point number that signifies the integer power to which the base is raised in determining the value of the represented number.

F-I

floating-point number system A system for representing a subset of real numbers in which the spacing between representable numbers is not a fixed, absolute constant. Such a system is characterized by a base, a sign, a significand, and an exponent (usually biased). The value of the number is the signed product of its significand and the base raised to the power of the unbiased exponent.

gradual underflow When a floating-point operation underflows into the range of subnormal numbers, return a subnormal number instead of zero. This method of handling underflow minimizes the loss of accuracy in floating-point calculations on small numbers.

hidden bits Extra bits used by hardware to ensure correct rounding, not accessible by software. For example, IEEE double precision operations use three hidden bits to compute a 56-bit result that is then rounded to 53 bits.

IEEE Standard 754 The standard for binary floating-point arithmetic developed by the Institute of Electrical and Electronics Engineers, published in 1985, revised in 2008.

in-line template A fragment of assembly language code that is substituted for the function call it defines, during the inlining pass of Oracle Developer Studio compilers. Used (for example) by the math library in in-line template files (`libm.il`) in order to access hardware implementations of trigonometric functions and other elementary functions from C programs.

L-P

NaN Stands for Not a Number. A symbolic entity that is encoded in floating-point format.

normal number In IEEE arithmetic, a number with a biased exponent that is neither zero nor maximal (all ones), representing a subset of the normal range of real numbers with a bounded small relative error.

pipelining A hardware feature where operations are reduced to multiple stages, each of which takes (typically) one cycle to complete. The pipeline is filled when new operations can be issued each cycle. If there are no dependencies among instructions in the pipe, new results can be delivered each cycle. Chaining implies pipelining of dependent instructions. If dependent instructions cannot be chained, when the hardware does not support chaining of those particular instructions, then the pipeline stalls.

precision A quantitative measure of the density of representable numbers. For example, in a binary floating point format that has a precision of 53 significant bits, there are 253 representable numbers between any two adjacent powers of two (within the range of normal numbers). Do not confuse precision with accuracy, which expresses how closely one number approximates another.

Q-R

quiet NaN A NaN (not a number) that propagates through almost every arithmetic operation without raising new exceptions.

radix The base number of any system of numbers. For example, 2 is the radix of a binary system, and 10 is the radix of the decimal system of numeration. SPARC workstations use radix-2 arithmetic; IEEE Std 754 is a radix-2 arithmetic standard.

round Inexact results must be rounded up or down to obtain representable values. When a result is rounded up, it is increased to the next representable value. When rounded down, it is reduced to the preceding representable value.

roundoff error The error introduced when a real number is rounded to a machine-representable number. Most floating-point calculations incur roundoff error. For any one floating-point operation, IEEE Std 754 specifies that the result shall not incur more than one rounding error.

S-T

signaling NaN A NaN (not a number) that raises the invalid operation exception whenever it appears as an operand.

significand The component of a floating-point number that is multiplied by a signed power of the base to determine the value of the number. In a normalized number, the significand consists of a single nonzero digit to the left of the radix point and a fraction to the right.

single precision Using one computer word to represent a number.

stderr Standard Error is the Unix file pointer to standard error output. This file is opened when a program is started.

store 0 Same as abrupt underflow. See *abrupt underflow*.

subnormal number In IEEE arithmetic, a nonzero floating point number with a biased exponent of zero. The subnormal numbers are those between zero and the smallest normal number.

two's complement The radix complement of a binary numeral, formed by subtracting each digit from 1, then adding 1 to the least significant digit and executing any required carries. For example, the two's complement of 1101 is 0011.

U-Z

ulp Stands for unit in last place. In binary formats, the least significant bit of the significand, bit 0, is the unit in the last place.

ulp(x) Stands for `ulp` of x truncated in working format.

underflow A condition that occurs when the result of a floating-point arithmetic operation is so small that it cannot be represented as a normal number in the destination floating-point format with only normal roundoff.

word An ordered set of characters that are stored, addressed, transmitted and operated on as a single entity within a given computer. In the context of SPARC workstations, a word is 32 bits.

wrapped number In IEEE arithmetic, a number created from a value that otherwise overflows or underflows by adding a fixed offset to its exponent to position the wrapped value in the normal number range. Wrapped results are not currently produced on SPARC workstations.

Index

G

I

L

N

O

P