HP-UX Floating-Point Guide

HP 9000 Computers

Edition 4

B3906-90005

May 1997

Printed in: United States

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Contents

Contents

Contents

Contents

Contents

Figures

Tables

Tables

Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. the manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

Edition	Date	Part No.	HP-UX Release	Reason for New Edition
First	August 1992	B2355-90024	9.0	First publication of *HP-UX Floating-Point Guide*.
Second	January 1995	B3906-90003	10.0	New file layout for 10.0 to support compatibility with UNIX SVR4.
Third	July 1996	B3906-90004	10.20	Support for PA2.0 architecture. Last OS release supported on PA1.0 machines.
Fourth	May 1997	B3906-90005	10.30	PA1.1 and PA2.0 are the only supported architectures at this release. Changes to library file structure.

Preface

The *HP-UX Floating-Point Guide* describes how floating-point arithmetic is implemented on HP 9000 systems and discusses how floating-point behavior affects the programmer. This book provides information useful to any programmer writing or porting floating-point-intensive programs.

We recommend that you start with Chapter 1, which not only provides an overview of floating-point principles but also provides important information about HP-UX math libraries.

If you are unfamiliar with floating-point issues, you should go next to Chapter 2 and Chapter 3. Chapter 2 provides an overview of the IEEE Standard for Binary Floating-Point Arithmetic, which constitutes the foundation for many floating-point architectures, including Hewlett-Packard's. For more specific information about the standard, contact the IEEE Standards Board. Chapter 3 describes the many factors that can cause floating-point computations to produce unexpected results.

If you already understand floating-point issues, you may want to go directly to Chapter 4 and the subsequent chapters and appendixes, which provide specific information about floating-point behavior on HP 9000 systems and about the libraries and facilities available on these systems.

If you have the HP FORTRAN/9000 (FORTRAN 77) compiler, all of the sample programs in this manual are online, in the directory `/opt/fortran/lib/demos/FPGuide`. In the manual, a comment in the text of each sample program indicates the path name of the program.

Comments

We welcome your comments on this manual. Please send electronic mail to `editor@ch.hp.com`, or send regular mail to

MLL Learning Products
Hewlett-Packard Company
Mailstop: CHR 02 DC
300 Apollo Drive
Chelmsford, MA 01824

Audience

This manual is written for application developers who write programs that perform mathematical operations. It assumes a basic knowledge of the HP-UX operating system and of a high-level programming language such as C or Fortran.

Summary of Technical Changes

This edition of the *HP-UX Floating-Point Guide* describes the following changes to the HP-UX math libraries at Release 10.30.

The PA1.0 math libraries are no longer supported at HP-UX Release 10.30. The math libraries provided in the `/usr/lib` directory support both PA1.1 and PA2.0 systems.

A version of the BLAS library tuned for optimal performance on PA2.0 systems is provided in the directory `/opt/fortran90/lib/pa2.0/libblas.a` (for HP Fortran 90) or `/opt/fortran/lib/pa2.0/libblas.a` (for HP FORTRAN/9000, the Fortran 77 product).

The vector library, `libvec.a`, is obsolete. It is supplied for backward compatibility but has been moved to the directory `/opt/fortran/old/lib`.

At Release 10.30, the C math library implements several new functions approved by the ISO/ANSI C committee for inclusion in the C9X draft standard, and eliminates support for several functions supported neither by the C9X standard nor by the XPG4.2 standard.

The following C library functions are no longer supported:

- `cabs` (`hypot` has equivalent functionality)
- `drem` (`remainder` has equivalent functionality)
- `finite` (replaced by the `isfinite` macro)
- The *fpgetround*(3M) suite of functions (replaced by the *fenv*(5) suite)
- `matherr`

The C library supports the following new functions and macros:

- The *fenv*(5) suite of functions (see Chapter 5)
- `isfinite` macro
- `isnormal` macro
- `signbit` macro

The following C library functions are now implemented as macros:

- `fpclassify` macro (supersedes `fpclassify` and `fpclassifyf` functions)
- `isinf` macro (supersedes `isinf` and `isinff` functions)
- `isnan` macro (supersedes `isnan` and `isnanf` functions)

The value of `HUGE_VAL` has changed from "The maximum non-infinity value of a double-precision floating-point number" to positive infinity.

Related Documentation

The *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985) is the essential reference for developers of floating-point applications. To obtain a copy, write to the IEEE Standards Board, 345 East 47th Street, New York, NY 10017, USA.

The document ANSI/IEEE Std 754-1985, entitled "IEEE Standard for Binary Floating-Point Arithmetic," was also published in *ACM SIGPLAN Notices* 22(2), pp. 9 - 25, Feb. 1987.

The international version of the standard is *Binary floating-point arithmetic for microprocessor systems*, second edition (IEC 559:1989).

Information on HP-UX programming languages and tools is available in both online and hardcopy format. The following online guides are available through the HP Help System if you have the relevant compilers or tools:

- HP FORTRAN/9000 Online Reference
- HP C Online Reference
- HP C++ Online Programmer's Guide
- HP-UX Linker and Libraries Online User Guide

You may also refer to the following hardcopy documents:

- *Programming on HP-UX* (B2355-90653) provides an overview of programming on HP-UX. It includes information about linking programs, creating and managing user libraries, and optimizing programs.
- The *HP Fortran 90 Programmer's Reference* (B3908-90001) describes the Fortran 90 programming language on HP-UX systems.

- The *HP FORTRAN/9000 Programmer's Reference* (B3906-90002) and *HP FORTRAN/9000 Programmer's Guide* (B3906-90001) describe the FORTRAN 77 programming language on HP-UX systems.
- The *HP C/HP-UX Reference Manual* (92453-90024) and *HP C Programmer's Guide* (92434-90002) describe the C programming language on HP-UX systems.
- The *HP Pascal/HP-UX Reference Manual* (92431-90005) and *HP Pascal/HP-UX Programmer's Guide* (92431-90006) describe the HP Pascal programming language on HP-UX systems.
- The *HP PA-RISC Compiler Optimization Technology White Paper* (5964-9846E) provides detailed information about compiler optimization levels, optimization types, and specific optimizations.
- The *HP/DDE Debugger User's Guide* (B3476-90015) describes the HP DDE debugger. The *HP PAK Performance Analysis Tools User's Guide* (B3476-90017) describes the HP-UX performance analysis tools.
- The *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* (09740-90039) describes the PA-RISC 1.1 architecture. *PA-RISC 2.0 Architecture*, by Gerry Kane (Prentice-Hall, ISBN 0-13-182734-0), describes the PA-RISC 2.0 architecture.
- The *Assembly Language Reference Manual* (92432-90001) describes assembly language programming on HP-UX systems. The *ADB Tutorial* (92432-90005) introduces the assembly language debugger.

To order manuals, call HP DIRECT at 1-800-637-7740. Outside the USA, please contact your local sales office.

Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

`literals`	This font indicates commands, keywords, options, literals, source code, system output, and path names. In syntax formats, this font indicates commands, keywords, and punctuation that you must enter exactly as shown.
`user input`	In examples, this font represents user input.
variables, titles	In syntax formats, words or characters in this font represent values that you must supply. This font is also used for book titles and for emphasis.
terms	This font indicates the first use of a new term.
name(N)	A word in title font followed by a number in parentheses indicates an online reference page (man page). For example, *cc*(1) refers to the *cc* page in Section 1 of the online man pages.
Vertical ellipsis	A vertical ellipsis means that irrelevant parts of a figure or example have been omitted.
Revision bars	Revision bars in the margin show where significant changes have been made to this manual since the last edition.

In This Book

This manual is organized as follows:

Chapter 1	Provides an overview of the basic principles involved in writing floating-point programs and of the HP-UX math libraries used by most floating-point applications.
Chapter 2	Provides an overview of the IEEE Standard for Binary Floating-Point Arithmetic, the standard on which HP floating-point behavior is based.
Chapter 3	Describes the factors that can cause the results of floating-point computations to vary from one system to another or even from one execution to another.
Chapter 4	Describes the math libraries available with the HP-UX operating system on HP 9000 platforms.
Chapter 5	Describes the methods you can use to manipulate the floating-point status register.
Chapter 6	Describes how to enable and handle floating-point traps.
Chapter 7	Describes techniques for obtaining the best possible floating-point performance.
Appendix A	Describes the C math library.
Appendix B	Describes the Fortran math library.
Appendix C	Provides a checklist to help you locate possible causes of a floating-point programming problem.
Glossary	Defines the terms used in this manual.

1 Introduction

This chapter introduces some of the basic principles involved in writing floating-point programs and introduces the HP-UX math libraries used by most floating-point applications.

Overview of Floating-Point Principles

In the context of computer programming, the term **floating-point** refers to the ways in which modern computer systems represent real numbers and perform real arithmetic. Computers use special representations for floating-point numbers. They also have special rules for performing floating-point arithmetic that differ from the rules for performing integer arithmetic. Usually, a computer has special hardware for performing floating-point calculations at a higher speed than would be possible using the computer's integer-oriented hardware.

In all modern computer systems, representations of real numbers are inherently inexact. There are an infinite number of real numbers, and a digital computer can represent only a finite subset of them.

When you write a program that attempts to generate an unrepresentable value, the computer approximates the value by choosing a representable value close to the one you attempted to generate. Data that is input into a computer in floating-point format is almost always approximate, and the calculations performed by the computer are usually approximations of the intended mathematical operations; therefore, the results you receive from a mathematical computation are also usually approximations.

The approximate nature of floating-point arithmetic has several important ramifications:

- Results from floating-point calculations are almost never exactly equal to the corresponding mathematical value.
- Results from a particular calculation may vary from one computer system to another, and all may be valid. However, when the computer systems conform to the same standard, the amount of variation is drastically reduced.
- Incorrect results are not necessarily caused by programming errors in the traditional sense. Correcting the problems may require an understanding of the floating-point approximation techniques used by the computer system executing the program.

The types of incorrect results and unexpected errors that floating-point applications sometimes generate can be very difficult to interpret if you do not understand how your computer performs floating-point

arithmetic. The purpose of this book is to help you avoid or fix these types of problems on HP 9000 computer systems and to help you increase the performance of your floating-point-intensive applications.

Overview of HP-UX Math Libraries

Basic operations such as addition and multiplication are specified by the IEEE standard. More complex mathematical operations such as logarithmic and trigonometric functions are provided by math library routines. The high-level operations of math library routines are specified not by the IEEE standard but by individual language standards (such as ISO/ANSI C) and by programming environment specifications (such as X/Open and SVID).

C math library functions are located in the `libm` math library. The `libm` functions operate according to the C standard and the latest versions of the *System V Interface Definition* (currently SVID3) and of the *X/Open Portability Guide* (currently XPG4.2). The XPG4.2 specification is a superset of the POSIX.1 standard (IEEE Std 1003.1-1990). The SVID3 and XPG4.2 specifications are compatible. The `libm` library also supports some functions and macros approved by the ISO/ANSI C committee for inclusion in the C9X draft standard.

NOTE

The `libM` library, which formerly supported XPG and POSIX while the `libm` library supported SVID, is obsolete now that these standards are compatible. The various versions of `libM` now exist only as soft links to the corresponding versions of `libm`. (See “Locations of the Math Libraries at Release 10.30” on page 27 for details.)

Fortran and Pascal intrinsic functions are located in the `libcl` library. In addition, Basic Linear Algebra Subroutine (BLAS) library routines are provided in the `libblas` library (provided with the HP Fortran 90 and HP FORTRAN/9000 products only).

Table 1-1 lists the math libraries available on HP-UX systems and shows the option you specify to the compiler or linker in order to link in each library.

Table 1-1 HP-UX Math Libraries

Library Name	Description	Linker Option
`libm`	C math library; ANSI C, POSIX, XPG4.2, and SVID specifications	`-lm`
`libcl`	Fortran and Pascal library	Linked in automatically by `f90`, `f77`, and `pc` commands; use `-lcl` with other compiler commands
`libblas`	Basic Linear Algebra Subroutine (BLAS) library (provided with the HP Fortran 90 and HP FORTRAN/9000 products only)	`-lblas`

Math Libraries and System Architecture

The `libm` and `libcl` math libraries on HP-UX operating system at Release 10.30 and later are targeted primarily to the PA-RISC 1.1 architecture (PA1.1). They also run well on PA-RISC 2.0 (PA2.0) systems.

The libraries will execute on all HP 9000 systems that run HP-UX Release 10.30 and later.

The HP Fortran 90 and HP FORTRAN/9000 products supply two versions of the BLAS library, one specially tuned for PA1.1 systems and the other specially tuned for PA2.0 systems.

NOTE The PA-RISC 1.0 architecture is no longer supported, and the PA1.0 libraries are no longer provided on HP-UX systems.

All HP 9000 systems except the oldest Series 800 systems are PA1.1-based or PA2.0-based. If you do not know your system's architecture type, see "Determining Your System's Architecture Type" on page 26.

For complete information about the math libraries, see Chapter 4.

Determining Your System's Architecture Type

There are two main ways to find the architecture type of your system. To do it from the command line:

1. Issue the command `uname -m` to learn the model number of your system. For example:

   ```
   $ uname -m
   9000/879
   ```

2. Look up the second part of the model number in the file `/opt/langtools/lib/sched.models` to find its architecture type. For example:

   ```
   $ grep 879 /opt/langtools/lib/sched.models
   879     2.0     PA8000
   ```

   The `2.0` indicates that a Model 879 is PA2.0-based.

You can also learn the system architecture type at run time at HP-UX Release 10.*x*. A simple program that gives you useful information follows.

Sample Program: get_arch.c

```
#include <stdio.h>
#include <sys/utsname.h>

extern int _SYSTEM_ID;
extern int _CPU_REVISION;

struct utsname uts;

int main(void)
{
    uname(&uts);
    printf("Release = %s\n", uts.release);
    printf("_SYSTEM_ID = %x\n", _SYSTEM_ID);
    printf("_CPU_REVISION = %x\n", _CPU_REVISION);
}
```

The `uts.release` is the release of HP-UX on the system where you run the program. The `_SYSTEM_ID` is the kind of code the compiler generated. The `_CPU_REVISION` is the architecture type.

If you compile this program on a PA1.1 system, then run it on a PA2.0 system running HP-UX Release 10.30, you get results like the following:

```
Release = B.10.30
_SYSTEM_ID = 210
_CPU_REVISION = 214
```

The release, 10.30, is easy to decipher. To decode the other results, search the file /usr/include/sys/unistd.h:

```
$ grep 210 /usr/include/sys/unistd.h
#  define CPU_PA_RISC1_1  0x210 /* HP PA-RISC1.1 */
$ grep 214 /usr/include/sys/unistd.h
/#  define CPU_PA_RISC2_0  0x214 /* HP PA-RISC2.0 */
```

The compiler generated PA1.1 code, which is running on a PA2.0 system.

Selecting Different Versions of the Math Libraries

If you use a compilation command (f90, cc, and so on) to invoke the link editor (ld), the selection of math libraries is driven by the +DA compiler option, which allows you to generate PA1.1 code (+DA1.1) or PA2.0 code (+DA2.0). By default, the compiler generates code for the kind of system on which you are running the compiler. This ensures the best possible performance on that system.

If your application must run on both PA1.1 and PA2.0 systems, compile with +DA1.1. Code compiled with +DA2.0 will run only on PA2.0 systems.

When you select +DA1.1 or +DA2.0, the compilation command invokes ld with a library search path that begins with the PA1.1 library directories. Again, the search path may vary from compiler to compiler. For example, if you invoke the HP Fortran 90 compiler with +DA2.0, it contains the following:

```
/opt/fortran90/lib/pa2.0
/opt/fortran90/lib
/usr/lib
/opt/langtools/lib
```

Locations of the Math Libraries at Release 10.30

At Release 10.30, the main HP-UX math libraries are in the directory /usr/lib. The BLAS library is in both /opt/fortran90/lib and /opt/fortran/lib. The obsolete vector library exists only in /opt/fortran/old/lib.

NOTE The PA1.0 libraries are no longer provided.

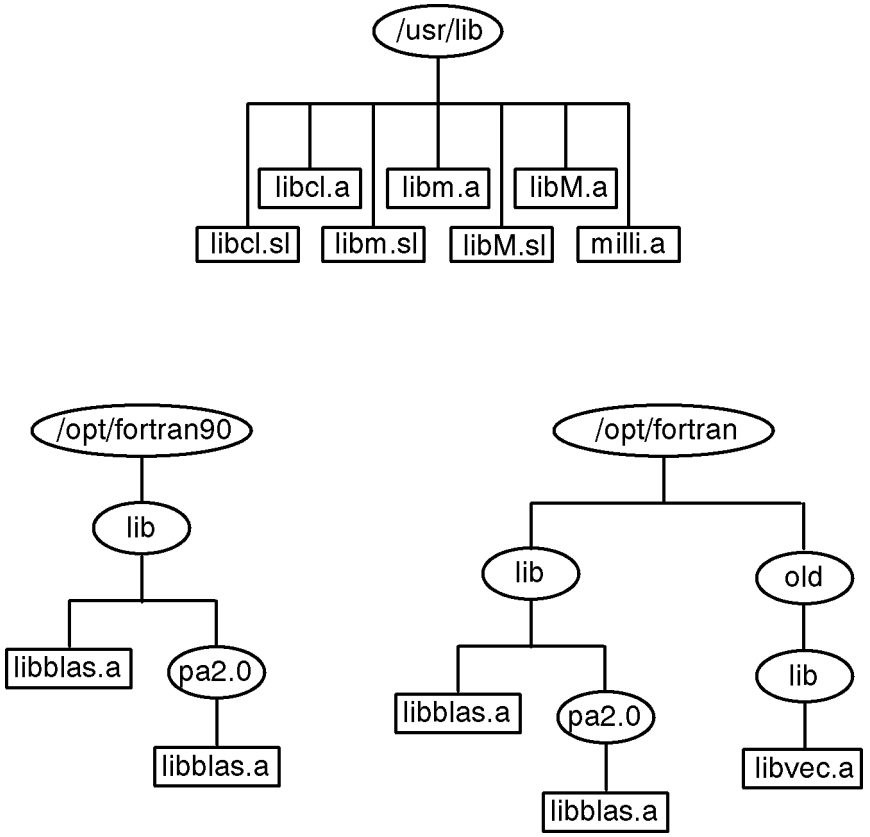
Table 1-2 shows the math library path names.

Table 1-2 Math Library Path Names

Library Path Name	Description
/usr/lib/libm.a	C math library, archive version
/usr/lib/libm.sl	C math library, shared version
/usr/lib/libcl.a	Compiler library (includes Fortran math library), archive version
/usr/lib/libcl.sl	Compiler library (includes Fortran math library), shared version
/usr/lib/milli.a	Millicode versions of several math library routines, available only if you compile with +Olibcalls. For details, see "Millicode Versions of Math Library Functions" on page 112.
/opt/fortran90/lib/libblas.a	Basic Linear Algebra Subroutine (BLAS) library, archive version (provided with the HP Fortran 90 product)
/opt/fortran90/lib/pa2.0/libblas.a	Basic Linear Algebra Subroutine (BLAS) library, PA2.0 archive version (provided with the HP Fortran 90 product)
/opt/fortran/lib/libblas.a	Basic Linear Algebra Subroutine (BLAS) library, archive version (provided with the HP FORTRAN/9000 product)
/opt/fortran/lib/pa2.0/libblas.a	Basic Linear Algebra Subroutine (BLAS) library, PA2.0 archive version (provided with the HP FORTRAN/9000 product)
/opt/fortran/old/lib/libvec.a	Vector library, archive version (provided with the HP FORTRAN/9000 product only) (obsolete)

Figure 1-1 illustrates the directory hierarchy for the math libraries.

Figure 1-1 Math Library Directory Hierarchy at Release 10.30



2 Floating-Point Principles and the IEEE Standard for Binary Floating-Point Arithmetic

This chapter introduces the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985). Throughout this chapter and the remainder of the book, we refer to the IEEE Standard for Binary Floating-Point Arithmetic as “the IEEE standard” or simply “the standard.” Programmers who intend to write floating-point-intensive code should become familiar with the IEEE standard.

What Is the IEEE Standard?

The **IEEE standard** was approved in 1985. Its main purpose is to define specifications for representing and manipulating floating-point values so that programs written on one IEEE-conforming machine can be moved to another conforming machine with predictable results. In addition, the standard is specifically designed to make it easier for programmers to write useful and robust floating-point programs. The standard defines the following:

- Formats for representing floating-point numbers
- Representations of special values (for example, infinity, very small values, and non-numbers)
- Five types of exception conditions, when they occur, and what happens when they do occur
- Four rounding modes (different algorithms for rounding values)
- A minimum set of operations that can be performed on floating-point values, precisely defined so that they yield the same results for the same operands on any standard-conforming system

All HP 9000 systems comply with the IEEE standard. A complete understanding of the IEEE standard and your system's implementation of the standard is helpful for writing robust floating-point programs.

Floating-Point Formats

The IEEE standard specifies four formats for representing floating-point values:

- Single-precision
- Double-precision (optional, though a `double` type wider than IEEE single-precision is required by standard C)
- Single-extended precision (optional)
- Double-extended precision (optional)

The IEEE standard does not require an implementation to support single-extended precision and double-extended precision in order to be standard-conforming.

HP 9000 systems support the **single-precision**, **double-precision**, and **double-extended precision** formats. Double-extended precision format on these systems is also known as quadruple-precision or **quad-precision** format.

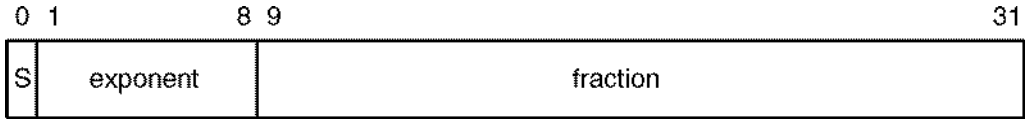
Single-Precision, Double-Precision, and Quad-Precision Formats

Single-precision, double-precision, and quad-precision values consist of three fields: sign bit, exponent, and fraction. The **sign bit** reflects the algebraic sign of the value. A 1 indicates a negative value; a 0 indicates a positive value. The **exponent** represents an integer value that is a power to which 2 is raised. The **fraction**, also called the **significand**, represents a value between 1.0 and 2.0 (for normalized values). The result of the exponent expression is multiplied by the fraction to yield the actual numerical value.

The only difference among the single-precision, double-precision, and quad-precision formats is the number of bits allocated for the exponent and fraction. Figure 2-1, Figure 2-2, and Figure 2-3 show the number of bits allocated in each format.

The single-precision format is 32 bits long: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fraction.

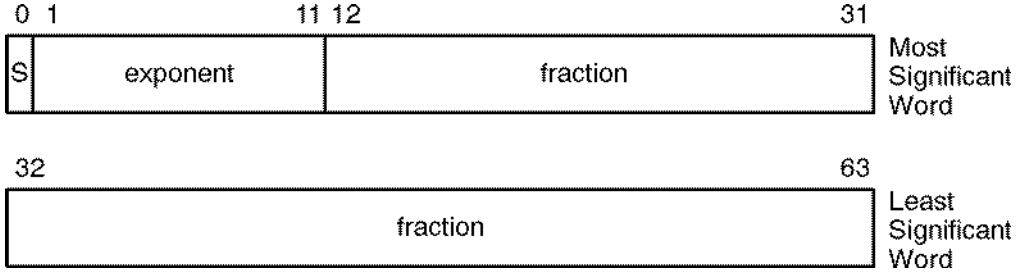
Figure 2-1 IEEE Single-Precision Format



The double-precision format is 64 bits long: 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fraction.

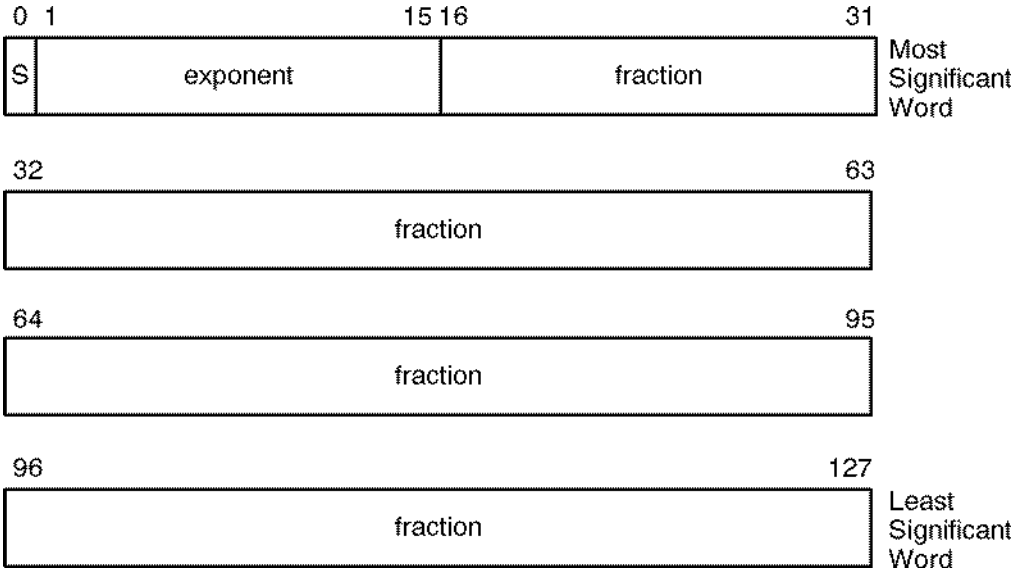
The double-precision format is sometimes divided conceptually into two 32-bit words. The word containing the sign bit, the exponent field, and the first portion of the fraction field is referred to as the **most significant word**. The other word, containing the last portion of the fraction, is called the **least significant word**.

Figure 2-2 IEEE Double-Precision Format



The quad-precision format is 128 bits long: 1 bit for the sign, 15 bits for the exponent, and 112 bits for the fraction. This format is divided conceptually into four 32-bit words: the most significant word, two middle words, and the least significant word.

Figure 2-3 IEEE Quad-Precision Format



NOTE On HP 9000 systems, the most significant word is stored at a lower memory address than the least significant word. If, for example, a double-precision value is stored at address 0x1000, the least significant word is stored at address 0x1004. If a quad-precision value is stored at address 0x1000, the least significant word is at address 0x100C. This ordering is often referred to as "big-endian."

The Fraction Field

For **normalized** values, the fraction represents a value greater than or equal to 1.0 and less than 2.0. Each bit in the fraction represents the value 2 raised to a negative power. For example, the first bit represents the value 2^{-1} (0.5), the second bit is 2^{-2} (0.25), and so on. The sum of 1.0 and the values represented by all these bits is the value of the fraction. The 1.0 in the sum corresponds to the zeroth fraction bit, 2^{0}. Since this bit would always be set for a normalized value, it is not included in the actual format, but it is implied. It is sometimes referred to as the **fraction implicit bit** or the **hidden bit**.

For example, if the 23 bits in the fraction field of a single-precision number are

011 0100 0000 0000 0000 0000

and the exponent field is not all 1's or all 0's, the fraction value is

$1.0 + 2^{-2} + 2^{-3} + 2^{-5} = 1.0 + 0.25 + 0.125 + .03125 = 1.40625$

The 1.0 represents the fraction implicit bit, and the exponents of –2, –3, and –5 indicate that the second, third, and fifth bits of the fraction field are set.

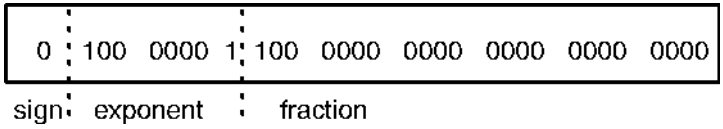
The Exponent Field

The exponent field uses a **biased representation**. This means that the value represented by the exponent field is the value in the exponent field interpreted as an unsigned integer minus a constant value (the **bias**). The purpose of the bias is to allow all exponent calculations to be performed using unsigned arithmetic. For single-precision formats, the bias is 127; for double-precision formats, it is 1023; for quad-precision formats, it is 16383.

Floating-Point Format: Examples

The value 6.0 would be represented in single-precision format as shown in Figure 2-4.

Figure 2-4 IEEE Single-Precision Format: Example



The first bit is the sign bit. Because the sign bit is 0, the floating-point value is positive. The next eight bits make up the exponent. 1000 0001 equals 129, but the true value of the exponent is derived by subtracting the bias constant 127 from this value. So the true exponent value is 2. The fraction bits are 100 0000 0000 0000 0000 0000, which, when added to the implicit bit, equal 1 + 0.5, or 1.5.

In algebraic terms, a floating-point value is

$(-1.0)^S * M * 2^{E-B}$

where S is the value of the sign bit, M is the fraction (with implicit bit), E is the exponent, and B is the bias.

In our example, this would be

$(-1)^0 * 1.5 * 2^2 = 1.5 * 4.0 = 6.0$

Table 2-1 shows some additional examples.

Table 2-1 IEEE Representations of Floating-Point Values

Hexadecimal Representation	Sign	Exponent	Fraction	Value
SP: 40C0 0000 DP: 4018 0000 0000 0000 QP: 4001 8000 0000 0000 0000 0000 0000 0000	+	129 – 127 = 2 1025 – 1023 = 2 16385 – 16383 = 2	1.0 + 0.5 = 1.5	$+1.5 * 2^2 = 6.0$
SP: BF00 0000 DP: BFE0 0000 0000 0000 QP: BFFE 0000 0000 0000 0000 0000 0000 0000	–	126 – 127 = –1 1022 – 1023 = –1 16382 – 16383 = –1	1.0 + 0.0 = 1.0	$–1.0 * 2^{–1} = –0.5$
SP: 7F00 0001 DP: 7FE0 0000 0000 0001 QP: 7FFE 0000 0000 0000 0000 0000 0000 0001	+	254 – 127 = 127 2046 – 1023 = 1023 32766 – 16383 = 16383	$1.0 + 2^{–23}$ $1.0 + 2^{–52}$ $1.0 + 2^{–112}$	$+1.00000019209 * 2^{127}$ $+1.000…001$ (51 zeros) $* 2^{1023}$ $+1.000…001$ (111 zeros) $* 2^{16383}$

Floating-Point Formats and the Limits of IEEE Representation

Because floating-point numbers have a finite number of bits in the fraction, only a finite subset of the continuum of real numbers can be represented exactly in IEEE format. The unit of granularity of the representable numbers is the **ULP (Unit in the Last Place)**. ULPs measure the distance between two numbers in terms of their representation in binary. One ULP is the distance from one value to the next representable value in the direction away from 0.

One ULP is about 1 part in 17 million for single-precision values, 1 part in 10^{16} for double-precision values, and 1 part in 10^{34} for quad-precision values. For this reason, there is a general rule of thumb that single-precision arithmetic represents about 7 or 8 decimal places, double-precision about 16, and quad-precision about 34. If you try to read or write a value with a greater number of decimal digits, the last digits will probably not contain useful information.

Because of this granularity in floating-point representation, most real numbers cannot be represented exactly. The result of an arithmetic operation (including the operation of converting from a decimal string into IEEE format) usually must be rounded to a nearby representable number. (For information on rounding, see “Inexact Result (Rounding)” on page 51.)

Even some simple fractions cannot be represented exactly. Consider the fraction 1/3. The exact value of this expression would require an infinite number of bits, because the value is an infinitely repeating fraction (0.33333…in decimal, 0.010101…in binary). Many values that can be represented exactly in a few decimal digits cannot be represented exactly in binary: for example, 1/10, which in decimal is 0.1, is in binary 0.000110011001100… Because simple numbers like 1/3 and 1/10 cannot be represented exactly, no floating-point operation can ever yield these exact values.

Although most real numbers cannot be represented exactly in floating-point arithmetic, a great many can. Any integer with a magnitude less than 16 million can be represented exactly in any format, and any 32-bit integer can be represented exactly in double-precision or quad-precision. Also, all numbers representable as some number over a power of 2, such as 0.1875 (3/16) or 27.375 (219/8), can be represented exactly if they have no more decimal digits than the chosen precision can faithfully represent.

Normalized and Denormalized Values

Values that are represented by a sign bit, a fraction, and an exponent whose bits are not all zeros and not all ones are called **normalized value**s (also called **normal** values). Because the value in the exponent field of a normalized value cannot be 0, the size of the exponent field determines the smallest value that can be represented in normalized format. For single-precision numbers, the largest-magnitude negative exponent is –126 (that is, 1 – 127); for double-precision numbers, it is –1022 (that is, 1 – 1023); for quad-precision numbers, it is –16382 (that is, 1 – 16383).

Denormalized values (also called **subnormal** values) fill in the gap on the number line between the smallest-magnitude normalized value and zero. They also allow floating-point values to satisfy the arithmetic rule that x is equal to y if and only if $x - y$ is equal to 0.

A denormalized value is represented by a zero exponent field and a nonzero fraction (if the fraction were also zero, the floating-point value would be zero). You can compute the value of a denormalized number by interpreting the fraction as an integer and then multiplying this integer by 2^{-149} for single-precision numbers, by 2^{-1074} for double-precision numbers, and by 2^{-16494} for quad-precision numbers. The maximum fraction is always $2^k - 1$, where k is the number of bits in the fraction. (Alternatively, you can compute the value by regarding the implicit bit as 0 and the exponent as 1 minus the bias.)

The purpose of denormalized values is to allow the space between the smallest normalized values to be divided up, so that as values become smaller they **underflow** with a gradually increasing loss of accuracy.

In the range of representable values, normalized values flow smoothly into denormalized values, but there is an increasing loss of accuracy as denormalized values become smaller and smaller. Table 2-2 shows the range of positive denormalized values. (The hexadecimal representation of the equivalent negative values begins with the digit 8; for example, the minimum negative denormalized value in single-precision is 8000 0001.)

When used as operands, denormalized values are treated exactly like normalized values in most instances. When a denormalized value is the result of an arithmetic operation, however, an underflow exception condition may occur. See “Underflow Conditions” on page 55 for more information about underflow exceptions. Also, you should be aware that denormalized values can significantly degrade performance. This issue is addressed in “Denormalized Operands” on page 180.

Table 2-2 Minimum and Maximum Positive Denormalized Values

Precision	Values	Hexadecimal Representation	Value

Single	Minimum denormalized
	Maximum denormalized
	Minimum normalized
		`0000 0001`
		`007F FFFF`
		`0080 0000`
			2^{-149}
			$2^{-149} * (2^{23} - 1)$
			2^{-126}

Double	Minimum denormalized
	Maximum denormalized
	Minimum normalized
		`0000 0000 0000 0001`
		`000F FFFF FFFF FFFF`
		`0010 0000 0000 0000`
			2^{-1074}
			$2^{-1074} * (2^{52} - 1)$
			2^{-1022}

Quad	Minimum denormalized Maximum denormalized Minimum normalized	(24 zeros)...`0000 0001` `0000 FFFF`...(24 more F's) `0001 0000`...(24 more zeros)	2^{-16494} $2^{-16494} * (2^{112}-1)$ 2^{-16382}

Infinity

Values that are larger in magnitude than the maximum-magnitude normalized values are approximated by special bit patterns that represent positive and negative **infinity**.

According to the IEEE standard, infinities are represented by setting all the bits in the exponent field to 1 (value 255 for single-precision, 2047 for double-precision, 32767 for quad-precision) and setting the fraction bits to 0. There are actually two infinity values, negative infinity if the sign bit is 1 and positive infinity if the sign bit is 0.

The IEEE standard defines the properties of infinities. For example, it defines what happens when you add a number to an infinity or subtract one infinity from another. Table 2-3 shows some of these properties. The term **finite value** in the table refers to any floating-point value other than infinity or NaN (see "Not-a-Number (NaN)" on page 43 for information about NaN values). For the multiplication and division operators, the sign of the result is determined by the usual arithmetic rules.

Table 2-3 Arithmetic Properties of Infinity

Operand	Operator	Operand	Result
+Infinity	+	Finite Value	+Infinity
–Infinity	+	Finite Value	–Infinity
+Infinity	+	+Infinity	+Infinity
–Infinity	+	–Infinity	–Infinity
+Infinity	+	–Infinity	NaN (invalid operation)
+Infinity	–	Finite Value	+Infinity
–Infinity	–	Finite Value	–Infinity
Finite Value	–	+Infinity	–Infinity
Finite Value	–	–Infinity	+Infinity
+Infinity	–	–Infinity	+Infinity
–Infinity	–	+Infinity	–Infinity
+Infinity	–	+Infinity	NaN (invalid operation)
–Infinity	–	–Infinity	NaN (invalid operation)
Infinity	*	Finite Value (except 0)	Infinity
Infinity	*	0	NaN (invalid operation)
Infinity	*	Infinity	Infinity
Infinity	/	Finite Value	Infinity
Finite Value	/	Infinity	0
Infinity	/	Infinity	NaN (invalid operation)
+Infinity	sqrt()		+Infinity
–Infinity	sqrt()		NaN (invalid operation)

NOTE In multiplication and division operations with infinity operands, the sign is determined by the usual arithmetic rules.

Not-a-Number (NaN)

A **NaN (Not-a-Number)** is a special IEEE representation for error values. A NaN can be

- The result of an invalid operation
- The result returned by a library function when it would be incorrect to return a numeric value
- An undetermined value

NaNs are represented by setting all of the bits in the exponent to 1 and setting at least one of the bits in the fraction field to 1.

There are two types of NaNs—a **signaling NaN (SNaN)** and a **quiet NaN (QNaN)**. When an SNaN is used, it generates an invalid operation exception and, if a **trap** for this exception is enabled, it produces a trap. A QNaN does not generate an exception; instead, it silently propagates through an operation. Floating-point operations produce only QNaNs.

NOTE

The IEEE standard does not fully define the bit patterns used by the two types of NaNs. HP 9000 systems use the most significant bit of the fraction to differentiate between the two types. If the bit is set to 1, it is an SNaN; if the bit is 0, it is a QNaN.

Table 2-4 shows some of the properties of NaNs.

Table 2-4 Properties of NaNs

Operand	Operator	Operand	Result
SNaN QNaN SNaN1 QNaN1	+ + + +	Finite Value Finite Value SNaN2 QNaN2	QNaN (invalid operation) QNaN QNaN (invalid operation) QNaN1 or QNaN2 (implementation-dependent)
SNaN QNaN	float_to_int() float_to_int()		Largest-magnitude integer (invalid operation) Largest-magnitude integer (invalid operation)
SNaN QNaN	sqrt() sqrt()		QNaN (invalid operation) QNaN

Zeros

The IEEE standard defines both a positive **zero** and a negative zero. In both cases, the value is represented by setting all bits in the exponent and fraction to zero. The only difference, therefore, is that the sign bit is set for a negative zero. Table 2-5 shows some of the properties of floating-point zeros.

Table 2-5 Operations With Zero

Operand	Operator	Operand	Result
+Zero	.EQ.	–Zero	True
+Zero –Zero +Zero –Zero	+ + + +	+Zero –Zero –Zero +Zero	+Zero –Zero +Zero (in round-to-nearest mode) +Zero (in round-to-nearest mode)
+Zero –Zero +Zero –Zero	– – – –	+Zero –Zero –Zero +Zero	+Zero (in round-to-nearest mode) –Zero (in round-to-nearest mode) +Zero –Zero
+Zero	*	–Zero	–Zero
Infinity Finite Value	/ /	Zero Zero	Infinity Infinity
–Zero	sqrt()		–Zero

NOTE

In multiplication and division operations with positive and negative zero operands, the sign is determined by the usual arithmetic rules.

The result of some operations is dependent on the rounding mode. The table assumes that the rounding is set to the default round-to-nearest mode. See “Inexact Result (Rounding)” on page 51.

Complex Data Types

The IEEE standard does not address the topic of complex arithmetic, so it does not define complex data type formats. HP Fortran 90 and HP FORTRAN/9000 implement two complex data types, single-precision complex (`COMPLEX`, `COMPLEX(KIND=4)`) and double-precision complex (`COMPLEX(KIND=8)`). The `COMPLEX` type consists of a real and an imaginary component, each of which is a single-precision IEEE operand. The `COMPLEX(KIND=8)` data type is analogous to `COMPLEX`, except that

each component is a double-precision IEEE operand type. HP Fortran 90 and HP FORTRAN/9000 support both complex data types and a full range of complex arithmetic operations.

NOTE

HP Fortran 90 and HP FORTRAN/9000 also support the nonstandard data type names `DOUBLE COMPLEX` and `COMPLEX*16` (equivalent to `COMPLEX(KIND=8)`) and `COMPLEX*8` (equivalent to `COMPLEX`).

IEEE Representation Summary

Table 2-6, Table 2-7, Table 2-8, and Table 2-9 summarize how IEEE values are represented in binary. To determine the class (normalized, infinity, NaN, and so on) of a floating-point value at run time, you can

- Use one of the following macros:
  - `fpclassify` to determine the value class
  - `isnan` to determine if the value is a NaN
  - `isinf` to determine if the value is an infinity
  - `isfinite` to determine if the value is finite (that is, neither infinity nor NaN)
  - `isnormal` to determine if the value is normalized

  See “Floating-Point Classification Macros” on page 116 for information about `fpclassify`. See the online man pages for information about all of these macros.
- Provide code in your program that writes out the value in hexadecimal (see “Displaying Floating-Point Values in Binary” on page 75 for an example)
- Examine the value in hexadecimal using the debugger

Table 2-6 IEEE Single-Precision Value Summary (Hexadecimal Values)

Value	Exponent	Fraction	Hexadecimal Values (Single-Precision)	
			Positive	Negative
Zero	All zeros	All zeros	0000 0000	8000 0000
Denormalized	All zeros	Nonzero	0000 0001 to 007F FFFF	8000 0001 to 807F FFFF
Normalized	Neither all zeros nor all ones	Anything	0080 0000 to 7F7F FFFF	8080 0000 to FF7F FFFF
Infinity	All ones	All zeros	7F80 0000	FF80 0000
Quiet NaN	All ones	Most significant bit 0	7F80 0001 to 7FBF FFFF	FF80 0001 to FFBF FFFF
Signaling NaN	All ones	Most significant bit 1	7FC0 0000 to 7FFF FFFF	FFC0 0000 to FFFF FFFF

Table 2-7 IEEE Single-Precision Value Summary (Decimal Values)

Value	Decimal Values (Single-Precision)	
	Positive	**Negative**
Zero	0.0	0.0
Denormalized	1.4012985E-45 to 1.1754942E-38	-1.4012985E-45 to -1.1754942E-38
Normalized	1.1754944E-38 to 3.4028235E+38	-1.1754944E-38 to -3.4028235E+38
Infinity	Not applicable	Not applicable
Quiet NaN	Not applicable	Not applicable
Signaling NaN	Not applicable	Not applicable

Table 2-8 IEEE Double-Precision Value Summary (Hexadecimal Values)

Value	Exponent	Fraction	Hexadecimal Values (Double-Precision)	
			Positive	Negative
Zero	All zeros	All zeros	0000 0000 0000 0000	8000 0000 0000 0000
Denormalized	All zeros	Nonzero	0000 0000 0000 0001 to 000F FFFF FFFF FFFF	8000 0000 0000 0001 to 800F FFFF FFFF FFFF
Normalized	Neither all zeros nor all ones	Anything	0010 0000 0000 0000 to 7FEF FFFF FFFF FFFF	8010 0000 0000 0000 to FFEF FFFF FFFF FFFF
Infinity	All ones	All zeros	7FF0 0000 0000 0000	FFF0 0000 0000 0000
Quiet NaN	All ones	Most significant bit 0	7FF0 0000 0000 0001 to 7FF7 FFFF FFFF FFFF	FFF0 0000 0000 0001 to FFF7 FFFF FFFF FFFF
Signaling NaN	All ones	Most significant bit 1	7FF8 0000 0000 0000 to 7FFF FFFF FFFF FFFF	FFF8 0000 0000 0000 to FFFF FFFF FFFF FFFF

Table 2-9 IEEE Double-Precision Value Summary (Decimal Values)

Value	Decimal Values (Double-Precision)	
	Positive	**Negative**
Zero	0.0	0.0
Denormalized	4.94065E–324 to 2.22507E–308	–4.94065E–324 to –2.22507E–308
Normalized	2.22507E–308 to 1.79769E+308	–2.22507E–308 to –1.79769E+308
Infinity	Not applicable	Not applicable
Quiet NaN	Not applicable	Not applicable
Signaling NaN	Not applicable	Not applicable

Exception Conditions

The IEEE standard defines five **exception condition**s, also called **exception**s:

- Inexact result
- Overflow
- Underflow
- Invalid operation
- Division by zero

The following sections describe the exceptions.

On HP-UX systems, traps for all of these exceptions are initially disabled by default. You can enable traps for some or all of these exceptions by using the `fesettrapenable` function, the `+T` option (`f77` only), the `+fp_exception` option (`f90` only), or the `+FP` compiler option. For more information, see "Enabling Traps" on page 151.

The standard requires that a conforming implementation provide **exception flags** (see "The PA-RISC Floating-Point Status Register" on page 126). If an exception occurs and a trap for the exception is not enabled, the corresponding exception flag is set.

NOTE

The IEEE standard defines the conditions under which exceptions occur only for the basic operations (see "Floating-Point Operations" on page 59). Exceptions for more complicated operations, such as trigonometric and transcendental functions, can vary depending on what compiler options you specified when compiling the program. For more information on exception handling, see "Math Library Basics" on page 99 and Chapter 6.

Inexact Result (Rounding)

As we explained in "Floating-Point Formats and the Limits of IEEE Representation" on page 38, all computer floating-point systems are inherently inexact because they cannot represent all values. When a computer system cannot represent a number exactly, it must choose a nearby representable value. This is called **rounding**, and it always

produces an **inexact result condition**. Because most floating-point operations produce rounded (that is, inexact) results most of the time, the inexact result exception is not usually considered to be an error.

The IEEE standard requires that for the basic operations, the result must always be rounded to the nearest representable value (unless the rounding mode has been changed; see "IEEE Rounding Modes" on page 52). So, while the result of dividing 1 by 3 is not precisely 1/3, it is precisely repeatable, portable, and standard.

An inexact result condition is always raised along with an overflow condition, and is also raised with an underflow condition if the result is inexact. The overflow or underflow is always raised first. These are the only situations where more than one exception is raised by the same operation.

Conversions Between Decimal and Binary Floating-Point Format

Inexactness can occur when the system attempts to convert between decimal representations and binary floating-point representations. This can occur, for example, during a C language `scanf` or `printf` call.

An inexact conversion from binary to decimal can occur if the format of the decimal representation does not contain enough room to represent the floating-point value—for example, if the format specification in a `printf` call is `7.5f`, but the value being printed requires more than 5 decimal places.

An inexact conversion from decimal to binary can occur because the IEEE floating-point format cannot represent all decimal values. For instance, the IEEE format cannot represent the decimal value 0.1 exactly, because 0.1 cannot be represented by a finite sum of powers of 2.

For more information and examples, see "Conversions Between Binary and Decimal" on page 74.

IEEE Rounding Modes

Choosing the most appropriate representable value through rounding is not always straightforward. Whether the system rounds to the lower or higher of two representable values depends upon the **rounding mode** (algorithm for rounding values) the user selects. The available choices include an IEEE standard default rounding mode as well as three alternate modes.

NOTE Most applications do not need the alternate rounding modes.

The default rounding mode is **round to nearest**. The four rounding modes are:

Round To Nearest Round to the representable value closest to the true value. If two representable values are equally close to the true value, choose the one whose least significant bit is 0.

Round Toward +Infinity Always round toward positive infinity (that is, choose the algebraically greater value).

Round Toward –Infinity Always round toward negative infinity (that is, choose the algebraically lesser value).

Round Toward Zero Always round toward zero (that is, choose the representation with the smaller magnitude).

For all but specifically designed numerical analysis applications, **round to nearest** is the best rounding mode. In round-to-nearest mode, the nearest representable value is never more than 1/2 ULP away from the exact result being rounded, so the error introduced from one operation by rounding is never more than 1/2 ULP. For the other rounding modes, the error is less than 1 ULP.

As an example of the size of a 1/2 ULP rounding error, suppose you tried to measure precisely the distance to the sun (about 93 million miles). An error of 1/2 ULP in single-precision would put your measurement off by about 2.5 miles; an error of 1/2 ULP in double-precision would put it off by about 8 microns; and an error of 1/2 ULP in quad-precision would put it off by about 10^{-17} microns, which is about one millionth of the diameter of a proton.

You can modify the rounding mode on HP 9000 systems by using library routines in the `fenv` suite of routines (see Chapter 5).

NOTE We recommend that you use the default rounding mode if your application calls library routines. Most math library routines (for example, `log`, `cos`, and `pow`) consist of many IEEE arithmetic operations, each of which is affected by the current rounding mode.

These routines are designed to produce the best possible results in the default rounding mode, round to nearest. Changing the rounding mode may cause library routines to yield results with more rounding errors in unpredictable directions.

Be careful if you change the rounding mode in the middle of your progra when you are optimizing your code. Some optimizations change the order of operations in a program, and the compiler may place the change in rounding mode after the operations it is intended to affect.

Overflow Conditions

An **overflow condition** occurs whenever a floating-point operation attempts to produce a result whose magnitude is greater than the maximum representable value. Table 2-10 shows approximate maximum values for floating-point numbers on HP 9000 systems. For example, an attempt to represent the value 10^{400} would produce an overflow condition in single-precision and double-precision, but not in quad-precision.

NOTE Out-of-range conditions that occur during conversions from floating-point to integer format are discussed in "Invalid Operation Conditions" on page 56.

Table 2-10 Approximate Maximum Representable Floating-Point Values

	Largest Negative Value	Largest Positive Value
Single-Precision	–3.4E38	3.4E38
Double-Precision	–1.7E308	1.7E308
Quad-Precision	–1.2E4932	1.2E4932

Several actions are possible when an overflow occurs, depending on whether overflow traps are enabled or disabled. (By default, traps are disabled.) If overflow traps are enabled, the system signals a floating-point exception (`SIGFPE`). Then, if the program provides a **trap handler**, the system takes whatever action is dictated by the trap handler. While the IEEE standard does not define trap handler operations, it does define what type of information should be stored in

the result of an operation that overflows. If the program does not provide a trap handler, the `SIGFPE` exception will cause the program to terminate.

If overflow traps are disabled, the result of a floating-point operation that overflows is assigned either an infinity code or the closest representable number (this will be either the largest positive value or the largest negative value). The choice of whether to use infinity or the nearest representable value depends on the rounding mode, as shown in Table 2-11. If overflow traps are disabled, the system generates an inexact result condition in addition to the overflow condition.

Table 2-11 Overflow Results Based on Rounding Mode

	Rounding Mode			
	Round To Nearest	**Round Toward +Infinity**	**Round Toward –Infinity**	**Round Toward Zero**
Positive Result	+Infinity	+Infinity	Maximum positive value	Maximum positive value
Negative Result	–Infinity	Maximum negative value	–Infinity	Maximum negative value

Underflow Conditions

An **underflow condition** may occur when a floating-point operation attempts to produce a result that is smaller in magnitude than the smallest normalized value. The standard allows the vendor of the floating-point system to choose whether an underflow condition is detected before or after rounding. On HP 9000 systems, underflow conditions always occur before rounding. Consequently, an operation that underflows can produce a minimum-magnitude normalized value, a denormalized value, or zero.

According to the standard, an underflow condition may be signaled only if it produces an inexact result, because it is possible that the result will be exact even though it is denormalized (for example, 2^{-1040}). In this case, there is no reason to signal an exception.

When an underflow condition does produce an inexact result, it is often difficult to determine whether the inaccuracy occurs because the value is denormalized or whether the loss of accuracy is inherent in the operation

being performed. For the sake of efficiency, the standard allows the implementor to decide how to define loss of accuracy for underflow conditions. On HP 9000 systems, the definition of loss of accuracy in underflow conditions includes *all* inaccuracies, whether they originate from denormalization or are inherent in the operation.

NOTE

In many properly functioning applications, underflows may occur in the normal course of execution—for example, in convergence algorithms. Many mathematically intensive applications encounter underflow conditions occasionally.

Underflow conditions can slow down floating-point operations considerably on HP 9000 systems. See "Denormalized Operands" on page 180 for information on what to do about performance problems caused by underflows.

Invalid Operation Conditions

An **invalid operation condition** occurs whenever the system attempts to perform an operation that has no numerically meaningful interpretation. The following are invalid operations (also called **operation errors**, **operand errors**, or **domain errors**):

- Any operation on a signaling NaN
- Magnitude subtraction of infinities (see Table 2-3 on page 42)
- Multiplication of zero by infinity
- Division of zero by zero or division of infinity by infinity
- Taking `remainder(`*x*`, `*y*`)` when *x* is infinity or *y* is zero (see "The Remainder Operation" on page 64)
- Taking the square root of a negative value (except for negative zero)
- Conversion of a floating-point value to an integer format when the floating-point value is an infinity or NaN or when the conversion results in a value outside the range of the integer format
- Comparison involving a <, <=, >, or >= operator when at least one operand is a NaN (see "Comparison" on page 60)

Out-of-range results that occur while converting from floating-point to integer trigger invalid operation conditions, but all floating-point overflows produce overflow conditions.

If an invalid operation condition occurs when invalid operation traps are disabled, the system by default returns a quiet NaN as the result of the operation. If traps are enabled, the system signals a floating-point exception and, if a trap handler is provided, takes whatever action the trap handler dictates.

Division by Zero Conditions

A **division by zero condition** occurs whenever the system attempts to divide a nonzero, finite value by zero. More generally, this condition occurs whenever an exact infinity is produced from finite operands. If divide-by-zero traps are disabled, the result is infinity: positive infinity if the two operands have the same sign, negative infinity if they have different signs. If traps are enabled, the system signals a floating-point exception and, if a trap handler is provided, takes whatever action the trap handler dictates.

Exception Processing

Exception processing refers to the sequence of events that takes place when any of the IEEE exception conditions occur. The standard states that a programmer should be able to enable or disable the trapping of any of the exception conditions. The standard also defines default results for all disabled exceptions. For example, Table 2-11 on page 55 shows how a default result is formulated when an overflow occurs and overflow traps are disabled.

The standard also states that a programmer should be able to define a trap handler for each exception condition. The **trap handler** (also called a **signal handler**) is a routine that is invoked whenever the particular exception condition is detected, assuming that trapping for the exception is enabled. If a program enables trapping but provides no trap handler, a default handler will be invoked that prints an error message and causes the process to terminate.

As we noted in "Exception Conditions" on page 51, all traps are disabled by default on HP-UX systems. You can use the `fesettrapenable` function, described in "Run-Time Mode Control: The fenv(5) Suite" on page 125, to enable any traps you want to handle. You can also enable traps through compiler options, which we discuss in "Command-Line Mode Control: The +FP Compiler Option" on page 146.

On HP-UX systems, the methods for establishing trap handlers for the IEEE-754 exceptions are the *sigaction*(2) routine for C programs and the `ON` statement for Fortran programs. See "Math Library Basics" on page 99 and "Handling Traps" on page 155 for more information.

Floating-Point Operations

The IEEE standard requires a complying system to support the following floating-point operations:

Addition Algebraic addition.

Subtraction Algebraic subtraction.

Multiplication Algebraic multiplication.

Division Algebraic division.

Comparison There are four possible relations between any two floating-point values: less than, equal, greater than, and **unordered**. The unordered relation occurs when one or both of the operands is a Not-a-Number (NaN). See "Comparison" on page 60 for details.

Square Root The square root operation never overflows or underflows.

Conversion The following conversions must be supported by a conforming implementation, if the implementation supports single-precision, double-precision, and quad-precision formats:

- Single-precision to double-precision
- Single-precision to quad-precision
- Double-precision to single-precision
- Double-precision to quad-precision
- Quad-precision to single-precision
- Quad-precision to double-precision
- Floating-point to integer
- Integer to floating-point
- Binary floating-point to decimal
- Decimal to binary floating-point

See "Conversion Between Operand Formats" on page 62 for more information about these conversions.

Round to Nearest Integral Value Rounds an argument to the nearest integral value (in floating-point format) based on the current rounding mode. Rounding modes are described in "Inexact Result (Rounding)" on page 51.

Remainder The remainder operation takes two arguments, *x* and *y*, and is defined as $x - y * n$, where *n* is the integer nearest the exact value x/y. See "The Remainder Operation" on page 64 for more information.

To understand the properties of each operation, you need a full understanding of denormalized numbers, infinities, and NaNs (see "Normalized and Denormalized Values" on page 39, "Infinity" on page 41, and "Not-a-Number (NaN)" on page 43). HP 9000 systems conform to the IEEE standard for all of these operations.

The standard requires that the result of each operation be rounded from its mathematically exact value into an IEEE representation in accordance with the rounding mode. In round-to-nearest mode (the default), the result is within 1/2 ULP. (There is one exception to this rule; conversions between binary and decimal need not be exact at the extremes of their ranges.)

Comparison

The comparison operation determines the truth of an assertion about the relationship of two floating-point values. The four basic assertions are

operand1 < operand2 The first operand is less than the second.

operand1 = operand2 The first operand is equal to the second.

operand1 > operand2 The first operand is greater than the second.

operand1 ? operand2 Unordered. This assertion is true if either operand is a NaN.

The basic assertions can be combined with each other. For example, "a >= b" asserts that a is greater than or equal to b. Similarly, "a <> b" asserts that a is either greater than or less than b; for operands that are not NaNs, this assertion is the opposite of "a = b".

NOTE The assertion operators should not be confused with actual programming language operators. Languages, for example, do not support the ? operator.

An assertion may also be negated; for non-NaN operands, negation of an assertion is the same as asserting the opposite assertion.

The IEEE standard defines two versions of every possible assertion: the aware and the non-aware version. The **aware** version of an assertion treats a NaN as a special value that compares as neither less than nor greater than any numeric value, and as unequal to any value, including any other NaN and even itself. This definition yields the interesting fact that the assertion "x = x" will evaluate to FALSE if x is a NaN. In fact, applications sometimes use this comparison operation specifically to detect NaNs, although it is a dangerous practice because some vendors' optimizers remove this operation from the code.

The **non-aware** version of an assertion behaves the same as the aware version, except that if either or both operands is a NaN, it also raises an invalid operation exception for the <, <=, >, and >= assertions. The =, !=, and ? assertions are the only ones that are valid with NaN operands.

Signaling NaNs cause an invalid operation exception for both aware and non-aware assertions.

The behavior of the comparison operation for each of the possible operand kinds is as follows:

Normalized and Denormalized Values	The operands are algebraically compared.
Zero	Zeros are greater than any nonzero negative value and less than any nonzero positive value. The sign of a zero is ignored, so that two zeros always compare as equal even if they have opposite signs.
Infinity	To the comparison operators, infinity is just another signed numeric value whose magnitude is greater than the largest normalized magnitude. Infinities with the same sign compare as equal to each other.
NaN	A NaN compares as unequal to all other operands, including other NaNs and itself. The rules above are used to evaluate assertions involving NaNs as TRUE

or FALSE. If the assertion is non-aware, an invalid operation exception is also signaled for any comparison involving a <, <=, >, or >= assertion.

Conversion Between Operand Formats

The standard requires that it be possible to convert between decimal and binary floating-point, and between binary floating-point and integer formats. This section describes some of the properties of various conversions. The operand type **integer** refers to either signed or unsigned integers.

Single-Precision to Double-Precision or Quad-Precision

These conversions can never overflow, underflow, or be inexact. The only possible type of exception is an invalid operation if the operand is an SNaN.

Double-Precision to Quad-Precision

These conversions can never overflow, underflow, or be inexact. The only possible type of exception is an invalid operation if the operand is an SNaN.

Quad-Precision or Double-Precision to Single-Precision

These conversions can overflow or underflow and are usually inexact.

Quad-Precision to Double-Precision

These conversions can overflow or underflow and are usually inexact.

Decimal to Single-Precision, Double-Precision, or Quad-Precision

These conversions can overflow or underflow and are usually inexact. See “Conversions Between Binary and Decimal” on page 74 for more information about these conversions.

Single-Precision, Double-Precision, or Quad-Precision to Decimal	These conversions can overflow or underflow and are usually inexact. See “Conversions Between Binary and Decimal” on page 74 for more information about these conversions.
Single-Precision, Double-Precision, or Quad-Precision to Integer	These conversions are usually inexact. Out-of-range finite values, infinities, and NaNs cause an invalid operation exception. The underflow exception does not apply to these conversions; results that are too small to round up to one round down to zero. Signed zeros become integer zeros. HP 9000 systems round these conversions in accordance with IEEE rounding rules. However, some programming languages, such as C, require that these conversions be performed with truncation. See “Truncation to an Integer Value” on page 90 for information about problems that can result when floating-point values are truncated to integer.
Integer to Quad-Precision	These conversions are always exact and never generate an exception.
Integer to Double-Precision or Single-Precision	These conversions are exact except for conversions of large 32-bit integer values to single-precision, or of large 64-bit integer values to double-precision, which may generate an inexact result exception.

The Remainder Operation

The remainder operation is an exact modulo function. When *y* is not equal to zero, the remainder *r* = `remainder(`*x*`, `*y*`)` is defined as

r = *x* - *y* * *n*

where *n* is the integer nearest the exact value *x*/*y*. When $|n - x/y| = 1/2$, *n* is even. If *r* is zero, its sign is that of *x*.

Two examples:

- The integer closest to the exact value 1.6/2.0 is 1. So the remainder of 1.6 and 2.0 is 1.6 – (2.0 * 1), or –0.4.
- The integer closest to the exact value 5.0/2.0 is 2 (the exact value is halfway between 2 and 3, so *n* is even). So the remainder of 5.0 and 2.0 is 5.0 – (2.0 * 2), or 1.

The result of the remainder operation is not affected by the rounding mode. (The result is always exact, so rounding is not a factor.)

The C math library `remainder` function implements the IEEE remainder operation.

Recommended Functions

In an appendix, the IEEE standard lists several useful floating-point functions that an implementor may support but is not required to support. Table 2-12 describes how HP-UX systems support these functions. The supported functions and macros are provided in the C library only. Appendix A describes these functions briefly; see the online man pages for more information.

Table 2-12 HP-UX Support for IEEE Recommended Functions

Function Name	HP Implementation
`class`	The `fpclassify` and `signbit` macros provide identical functionality. See "Floating-Point Classification Macros" on page 116.
`copysign`	Supported. Float version (`copysignf`) also supported.
`finite`	The `isfinite` macro provides identical functionality.
`logb`	Supported.
`nextafter`	Supported.
`isnan`	Supported.
`scalb`	Supported.
`unordered`	Not supported.

3 Factors that Affect the Results of Floating-Point Computations

When a floating-point application executes, the results it yields may be different from those of previous executions, or the results may be inaccurate. A great many factors can contribute to such differences or inaccuracies. This chapter describes these factors.

We've divided the factors that can affect application results into the following general categories:

- Basic operations
- Mathematical library functions
- Exception conditions and library errors
- Other system-related factors, usually indirect results of the three preceding factors
- Floating-point coding practices that lead to inaccurate results

How Basic Operations Affect Application Results

To understand why floating-point calculations can yield different results, you need to know how a system performs the most basic operations: add, subtract, multiply, divide. Recall from "Inexact Result (Rounding)" on page 51 that these operations always round their results to the nearest representable floating-point value using an algorithm specified by the rounding mode, and that this rounding introduces a **rounding error** into the result of the operation.

For example, suppose you run the following program. Two operations that are mathematically equivalent may produce different results:

Sample Program: rounderr.f

```
      PROGRAM ROUNDERR
      REAL A, B, C, D, E, F

      PRINT *, 'Enter 4 reals:'
      READ *, A, B, C, D

      E = (A + B) * (C + D)
      F = (A * C) + (A * D) + (B * C) + (B * D)

      IF (E .EQ. F) THEN
        WRITE (*, 20) E, ' equals ', F
      ELSE
        WRITE (*, 20) E, ' not equal to ', F
        WRITE (*, *) 'Math error!'
      ENDIF

 20   FORMAT(F, A, F)
      END
```

Depending on the values read for A, B, C, and D, the program will indeed print "Math error!", although, of course, no error has actually occurred—only legitimate rounding errors:

```
$ f90 rounderr.f
rounderr.f
 program ROUNDERR

21 Lines Compiled
$ ./a.out
 Enter 4 reals:
1.1 2.2 3.3 4.4
     25.4100018 not equal to      25.4099998
 Math error!
```

This example also demonstrates a guiding principle of floating-point programming:

Be very careful about testing two floating-point values or expressions for exact equality or inequality.

This principle derives from the fact that two given floating-point values are almost never equal, even when the programmer might expect them to be equal from a purely mathematical standpoint. The inequality occurs because of the rounding errors incurred during the calculation of the two values being compared.

For further discussion of this issue, see “Testing Floating-Point Values for Equality” on page 82.

How Mathematical Library Functions Affect Application Results

Mathematical library functions do not always yield identical results from one system to another, from one language to another, or even from one software release to another. The reason is that computer vendors often modify these functions for any or all of the following purposes:

- To make them faster
- To make them more accurate
- To take advantage of the capabilities of a particular hardware system

The art in algorithm design for math library routines is in finding the best tradeoff between accuracy and performance. Some algorithms are better than others at getting the most accurate results obtainable. For a given function, it is usually possible to design an algorithm that produces results that are accurate to within the 1/2 ULP mandated for the results of the basic operations. The most precise algorithm, however, is often unacceptably slow. Because correct rounding is not required for most mathematical functions (though it is required for basic operations and for some functions, such as `sqrt`), algorithm designers are free to sacrifice a small amount of accuracy in favor of better performance. As advances in algorithm design improve the speed and/or accuracy of the library functions, the results may change.

Because HP modifies math functions from time to time, you should not count on always getting identical results from a given function. Instead, it is wise to assume that a library call may have an error of 1 ULP or thereabouts. As we explain in “Testing Floating-Point Values for Equality” on page 82, you should test for results within a certain range, not for an exact value.

How Exceptions and Library Errors Affect Application Results

When an application performs a floating-point operation that causes an exception condition or a library error, and the application is not coded to detect and deal with the exception or error, the default exception-handling response of the system may introduce a dramatic amount of error into the continuing computation. The types of exception conditions and errors that can have this effect are

- Overflow conditions
- Underflow conditions
- Invalid operation conditions
- Division by zero conditions
- Library domain and range errors (`EDOM` and `ERANGE`)

"Exception Conditions" on page 51 describes the exception conditions.

Math library routines generate **domain errors** when they encounter invalid arguments; they generate **range errors** when they overflow or when they generate exact infinities from finite arguments.

Table 3-1 shows the effects of these exceptions and errors on typical applications. In many cases, the effect of an exception or a library error on a program is catastrophic: that is, the program continues, but the results it produces are meaningless. A common exception to this rule is an underflow condition. An underflow means that the value returned by the operation is extremely small. In many cases—for example, if the program is evaluating a function at the point where the function crosses the X axis—a value that is very near zero is exactly what you want, so that the occurrence of an underflow is actually a successful result. In other cases an overflow may be an expected result. If, in the expression `1 + 1/f(x)`, `f(x)` overflows, then `1/f(x)` is 0, and the expression evaluates to 1, which may be a perfectly acceptable result.

Table 3-1 Effects of Floating-Point Exceptions and Library Errors

Type of Error	Default System Behavior	Effect on Application
Overflow (ERANGE)	Substitute the largest normalized value or an infinity as the result	May be catastrophic unless specifically handled
Underflow	Substitute either a denormalized or zero value as the result: if the result after rounding would be smaller in magnitude than MINDOUBLE (the smallest denormalized value), substitute zero	Application usually continues successfully; however, performance may suffer
Invalid operation (invalid argument) (EDOM)	Substitute a NaN as the result	Catastrophic unless specifically handled
Division by zero	Substitute an infinity as the result	Catastrophic unless specifically handled

The actual nature of the inaccuracies introduced by exceptions and library errors depends on three main factors:

- The programming environment selected (C, Fortran)
- The default system error-response behavior, which is implementation-dependent
- The additional error-response behavior defined by the programmer

Other System-Related Factors that Affect Application Results

In the previous sections we discussed the three most fundamental causes of inaccuracy in floating-point computations:

- Rounding in basic operations
- Math library functions
- Exceptions and library errors

This section lists the factors that can contribute to inaccuracy or to changes in the results of an application. We also show how each factor is derived from the previously described three fundamental factors.

These factors are

- Conversions between binary and decimal
- Compiler behavior and compiler version
- Compiler options
- Hardware version of build-time system
- Operating system release of build-time system
- Operating system release of run-time system
- Values of certain modifiable hardware status register fields

Conversions Between Binary and Decimal

A conversion from decimal to binary or from binary to decimal may take place either at compile time or at run time. A compile-time decimal-to-binary conversion takes place when a statement like the following is compiled:

```
Y = 1.25E2
```

A run-time decimal-to-binary conversion takes place when the C library routine `atof` is called, or when a statement like the following is executed:

```
READ (*, '(G5.2)') X
```

The conversion between a decimal value and a binary floating-point value may cause a loss of accuracy for any of three reasons:

- The algorithm may not be accurate, probably because speed/accuracy tradeoffs have been made in favor of speed.
- Not all decimal values are exactly representable in binary—for example, 0.1.
- The conversion may be specified so as to deliberately limit precision. For example, the statements

  ```
  REAL X
  READ (*, '(G5.2)') X
  ```

  deliver only three decimal digits of precision (5 minus 1 for the sign and 1 for the decimal point). If the user enters –1.23456, the last three digits are lost. The same loss of precision on output can be particularly confusing if it causes very small values to be printed as zero, as in the following example:

  ```
  REAL X
  X = 1.0E-10
  WRITE(*, '(F5.2)') X
  ```

Displaying Floating-Point Values in Binary

It is possible in each language to read or print floating-point variables directly in binary (actually hexadecimal), where no conversion inaccuracies occur. Floating-point programmers should familiarize themselves with these techniques and should also examine floating-point variables in hexadecimal in the symbolic debugger.

The following Fortran program shows how you can display floating-point values in hexadecimal:

Sample Program: flophex.f

```
      PROGRAM FLOPHEX
      DOUBLE PRECISION X, Y

      X = 1.234D0
      Y = DCOS(X)
      WRITE(*, *) 'Y = ', Y
      WRITE(*, 10) Y
10    FORMAT(' Y = ', Z16.16)
      END
```

The Fortran program displays results similar to the following:

```
 Y =  .3304651080717298
 Y = 3FD526571FE8C7A5
```

The following C program shows how you can use a union to display floating-point values in hexadecimal:

Sample Program: flophex.c

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

union {
    double y;
    struct {
         unsigned int ym, yl;
    } i;
} di;

int main(void)
{
    double x;

    x = 1.234e0;
    di.y = cos(x);
    printf("di.y = %18.16g\n", di.y);
    printf("di.y = %08x%08x\n", di.i.ym, di.i.yl);
}
```

The C program displays results similar to the following:

```
di.y = 0.3304651080717299
di.y = 3fd526571fe8c7a5
```

You can see that, although the last digits of the floating-point values displayed by the Fortran and C programs are not identical, the hexadecimal values are exactly the same.

Compiler Behavior and Compiler Version

The compiler can contribute to variations in floating-point results in several ways, including

- Conversion of constants
- Decisions about constant folding and algorithms used in constant folding
- Rearrangement of operations

Like the math library functions, the parsing routines (both the run-time routines such as READ and scanf and the internal routines the compiler uses) attempt to come as close as possible to the ideal accuracy of less than 1/2 ULP and to be bit-for-bit compatible from one release to another. However, these routines may change slightly from time to time. Because

of this, the parsing of a floating-point constant might change if the exact value of the constant lies extremely close to the halfway point between two representable values.

The compiler usually performs compile-time expression evaluation, which is commonly referred to as **constant folding**. A statement like

```
X = 1.1/10.0E1 + 5.0E-1
```

will probably be compiled as

```
X = 0.511
```

but, because an extra division and addition took place, each contributing some rounding error, the result may not be bit-for-bit identical in all cases.

NOTE Floating-point constants without suffixes (1.1, for example) are evaluated differently in Fortran 90 and FORTRAN 77. FORTRAN 77 evaluates them in double precision, while Fortran 90 evaluates them in single precision. This can lead to slight differences in the resulting constant value.

The zeal with which the compiler pursues opportunities for constant folding may vary from one compiler release to another and may depend on the optimization level and other compiler options. In general, lower optimization levels produce more repeatable results.

New versions of compilers often incorporate improved constant folders and optimizers that compile applications into more efficient sequences of operations. However, as we have shown, different sequences of operations produce different results, even though the new sequence is mathematically equivalent to the old.

Compiler Options

A compiler typically has several options that affect the sophistication of the optimization algorithms used. As these compiler options change, the final sequence of operations produced by the compiler changes.

HP-UX compilers by default do not reorder floating-point operations, even at high optimization levels. The order of operations is the same as the order your program specifies. If, however, you want to allow the compiler to reorder floating-point operations so as to improve performance, even at the expense of possible small differences in results, specify the `+Onofltacc` option.

For example, by default the compiler orders the expression

```
a + b * c + d
```

as

```
(a + (b * c)) + d
```

But if you use `+Onofltacc`, it may change the ordering to

```
a + d + (b * c)
```

As we showed in "How Basic Operations Affect Application Results" on page 69, this kind of reordering has an effect on rounding errors and consequently on the final result.

The `+Ovectorize` option may also reorder operations on arrays and thus cause small differences in results. See "Optimizing Your Program" on page 169 for details.

NOTE

Optimization affects the ordering of operations around calls to the *fenv*(5) suite of functions. See "Run-Time Mode Control: The fenv(5) Suite" on page 125.

Architecture Type of Run-Time System

PA2.0 systems use two new instructions, known collectively as **FMA** (fused multiply-add) instructions. These instructions, `FMPYFADD` and `FMPYNFADD`, combine a multiplication and an addition (or subtraction) into a single operation. Use the `+DA2.0` compiler option (the default on PA2.0 systems) to generate code that uses these instructions.

For example, in the statement

```
d = a * c + b
```

the multiplication of `a` and `c`, and the addition of the product to `b`, may all be accomplished by a single `FMPYFADD` instruction. When the instruction is executed, the product of `a` and `c` is computed to infinite precision and added to `b`. The result is then rounded according to the current rounding mode.

This means, for example, that a code sequence that might be expected to incur two rounding errors may instead incur only one. Therefore, the use of FMA instructions may change the results of your application slightly.

FMA instructions are generated by default at optimization levels of 2 and higher on PA2.0 systems. If you want your optimized code to preserve exactly the expression semantics of your source code, specify the `+Ofltacc` option to suppress the generation of FMA instructions.

Operating System Release of Build-Time System

When the compiler performs binary-decimal conversions or constant folding, it calls system math libraries to perform certain operations. For example, the statement

```
X = LOG(10.1E2) + 1.2E0
```

causes the constant-folding phase of the compiler to invoke a system logarithmic function.

New operating system releases from time to time include improved math function libraries. The changes may be for improved performance, accuracy, or both. In any case, if the libraries on the build-time system change, the compiler's constant folding may yield a different result, and the application, regardless of which system it is run on, may also yield a different result.

Operating System Release of Run-Time System

As stated in "Operating System Release of Build-Time System", different operating system releases may have different libraries. If the libraries on the run-time system change, an application run on that system may yield different results from those yielded on previous releases.

If your application must produce the same results on every run-time system, you should build your application using archive libraries instead of shared libraries. Archive libraries cause the math routines of the build-time system to be permanently bound to the application, making it immune to future library changes on the run-time system. If you use archive libraries, however, you cannot take advantage of improvements in accuracy or performance in future library releases unless you rebuild your application. For more information about archive libraries, see "HP-UX Library Basics" on page 97 and "Shared Libraries versus Archive Libraries" on page 179.

Values of Certain Modifiable Hardware Status Register Fields

All HP 9000 systems have a modifiable **floating-point status register**. Figure 5-1 on page 126 illustrates this register. Two of the fields in the status register can be modified in ways that may change the results of an application:

- The rounding-mode field
- The D bit, which affects underflow mode

The rounding-mode field is a two-bit field that specifies which of the four rounding modes defined by the IEEE-754 standard is in force. The default setting for this field is round to nearest. Changing this field changes the rounding errors developed during computations and thus changes the results yielded by an application.

Many HP 9000 systems have a D bit in the floating-point status register. When this bit is set to 0 (the default), the system is fully IEEE-754 conforming. When this bit is set to 1, the system operates on denormalized operands as if they were zeros, and results that underflow are flushed to 0 instead of denormalized. The purpose of this bit is to improve the performance of applications that encounter denormalized values often, since operations on denormalized values degrade system performance. Changes to the D bit, by virtue of its effect, can obviously have the effect of altering application results.

You may modify the floating-point status register by using either a compiler option or any of several library functions. For more information, see Chapter 5.

Floating-Point Coding Practices that Affect Application Results

The most common types of floating-point "bugs" reported to Hewlett-Packard are not bugs at all, but rather a class of programming mistakes. These types of mistakes usually stem from one of the following invalid assumptions:

- It is invalid to assume that an arithmetic expression in a computer language produces an exactly representable result and that all of the digits in a floating-point value are always meaningful. The fact that an application prints out 25 decimal digits of a result does not mean that the value printed actually has 25 significant digits. Many of the rightmost digits printed may be meaningless. Values may lose precision in the course of computation, and you must be alert for the kinds of operations that can cause precision loss.

  The significance limitations of the system are immutable. Entering a datum of 3.14159265358979323846 for `pi` is no better than entering 3.1415926535897932 (in double-precision). In fact, the former may be worse, because it might beguile a programmer into thinking that the system accepted all 21 digits, when in reality it accepted only 17 (9 in single-precision).

- It is invalid to assume that an arithmetic expression in a computer language will abide by all algebraic rules, including the associative and distributive laws. You cannot make this assumption unless you have made a thorough analysis of the code to determine that rounding errors and other sources of inaccuracy will not invalidate the rules.

Sometimes the source of an erroneous result is related to a particular optimization generated by the compiler. This kind of problem can be particularly hard to solve, because it may disappear when you compile the program with a debugging option in order to debug it. See "Compiler Behavior and Compiler Version" on page 76 for a discussion of the effects of compiler optimization on floating-point results.

The following sections describe common floating-point programming mistakes that can lead to incorrect application results:

- Testing floating-point values for equality
- Taking the difference of similar values
- Adding values with very different magnitudes
- Unintentional underflow
- Truncation to an integer value
- Ill-conditioned computations

The first mistake produces results that are simply wrong. The others are more insidious: they usually cause the result of an operation to lose a substantial amount of precision.

NOTE

The illustrations in these sections use decimal representations, rather than binary or hexadecimal ones, so as to correspond more closely to the way most users think about floating-point values. These examples illustrate general principles; they cannot necessarily be reproduced using IEEE-754 arithmetic.

Testing Floating-Point Values for Equality

Think very carefully before you test two floating-point values for equality. Because of the inherent inexactness of floating-point representations and because of the many sources of rounding inaccuracies in a floating-point computation, values that should be equal from a purely algebraic perspective in fact rarely will be.

For example, on many computers 1.2 – 0.1 is not exactly equal to 1.1. Consider the following example.

Sample Program: fpeq.c

```
#include <stdio.h>

int main(void)
{
  union {
    double x;
    int a[2];
  } u1, u2;

  u1.x = 1.2 - 0.1;
  u2.x = 1.1;

  if (u1.x == u2.x)
    printf("1.2 - 0.1 equals 1.1\n");
  else {
    printf("1.2 - 0.1 is NOT equal to 1.1.\n");
    printf("1.2 - 0.1 = %x%x\n1.1 =       %x%x\n",
                          u1.a[0], u1.a[1], u2.a[0], u2.a[1]);
  }
}
```

From an algebraic viewpoint, this routine should print that 1.2 – 0.1 equals 1.1. In fact, though, when executed on a Series 700 machine, the output is

```
1.2 - 0.1 is NOT equal to 1.1.
1.2 - 0.1 = 3ff1999999999999
1.1 =       3ff199999999999a
```

This anomaly occurs because the numbers 0.1 and 1.1 cannot be represented exactly in IEEE-754 double-precision format. Both values, 1.1 and (1.2 – 0.1), are very close to the real number 1.1, but neither is exact.

A better technique in most circumstances is to test that two values lie within a relative proximity to each other. For example, the Fortran test

```
IF( X .EQ. Y )
```

could be replaced by

```
IF( ABS(X - Y) .LE. EPSILON )
```

where `EPSILON` is a sufficiently small floating-point value. This test establishes whether `X` is within `+/-EPSILON` of `Y`.

Consider again the example, `rounderr.f`, in "How Basic Operations Affect Application Results" on page 69. A better way to code that example is to test whether the two values are sufficiently close to each other, rather than exactly the same:

Sample Program: roundeps.f

```
      PROGRAM ROUNDEPS
      REAL A, B, C, D, E, F, EPSILON

      PRINT *, 'Enter epsilon:'
      READ *, EPSILON
      PRINT *, 'Enter 4 reals:'
      READ *, A, B, C, D

      E = (A + B) * (C + D)
      F = (A * C) + (A * D) + (B * C) + (B * D)

      IF (ABS(E - F) .LE. EPSILON) THEN
        WRITE (*, 20) E, ' equals ', F
      ELSE
        WRITE (*, 20) E, ' not equal to ', F
        WRITE (*, *) 'Math error!'
      ENDIF

 20   FORMAT(F, A, F)
      END
```

You must choose a value for the error bound `EPSILON` that is appropriate to the magnitudes of the values being computed. If computations are yielding results with magnitudes around `m`, then for single-precision computations, a reasonable value for `EPSILON` might be `m/1.0e5`; for double-precision computations, a reasonable value might be `m/1.0e14`; for quad-precision computations, a reasonable value would be `m/1.0e26`. Choosing an appropriate value for the error bound, however, requires a thorough knowledge of the mathematical nature of your application. For example, a value of `1.0e-5` for `EPSILON` yields the following result, which may or may not be acceptable in your application:

```
$ f90 roundeps.f
roundeps.f
   program ROUNDEPS

23 Lines Compiled
$ ./a.out
 Enter epsilon:
1.0e-5
Enter 4 reals:
1.1 2.2 3.3 4.4
     25.4100018 equals      25.4099998
```

The actual difference between the values is one bit (in hexadecimal, 41CB47AF versus 41CB47AE).

Taking the Difference of Similar Values

Calculations can lose precision when a program attempts to take the difference between two values that are similar in magnitude and also have some degree of inaccuracy to begin with. If the operands have *M* bits of insignificance, and the fractions are the same for the first *N* significant bits, then the difference between these two values will have *M+N* bits of insignificance because of the cancellation of significant bits during the subtraction.

For example, suppose that a single-precision application performs a computation using two different algorithms and then takes the difference between them to check the similarity of the results. Assume that the true result should be 1.0 and that both actual results have up to four bits of insignificance. If the actual results are

```
0x3f80001f
0x3f80003e
```

the magnitude of the difference is

```
0x36780000
```

However, the two values were the same for the first eighteen bits of their fractions, which were canceled during the subtraction. This leaves six bits in the difference, four of which are insignificant. So only two bits are significant, and the remaining 22 bits are insignificant. (Although the fraction field has only 23 bits, we include the fraction implicit bit in our count.)

Figure 3-1 shows how this problem commonly manifests itself in practice. Suppose you have two very large values:

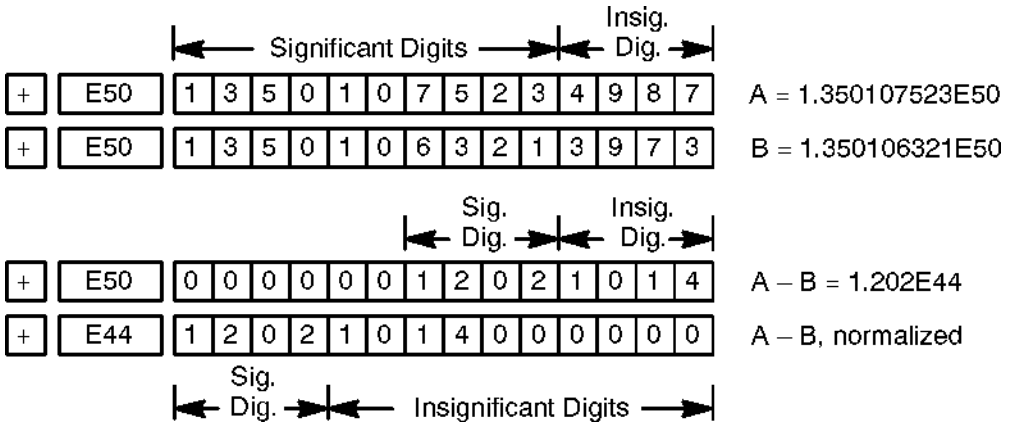
```
A = 1.350107523E50
B = 1.350106321E50
```

If you subtract B from A, the result is

```
C = 0.0000001202E50
```

which is normalized to 1.202E44. Suppose there are already 4 digits of insignificance in A and B. Normalizing the result from 0.0000001202E50 to 1.202E44 adds another 6 digits of insignificance. So the result has 10 digits of insignificance in all, though the operands had only 4.

Figure 3-1 Taking the Difference of Similar Values



The modulo operation (`mod(`*x*`, `*y*`)` in Fortran) is an instance of this type of problem when *x* is much greater than *y*; remember that the modulo formula is

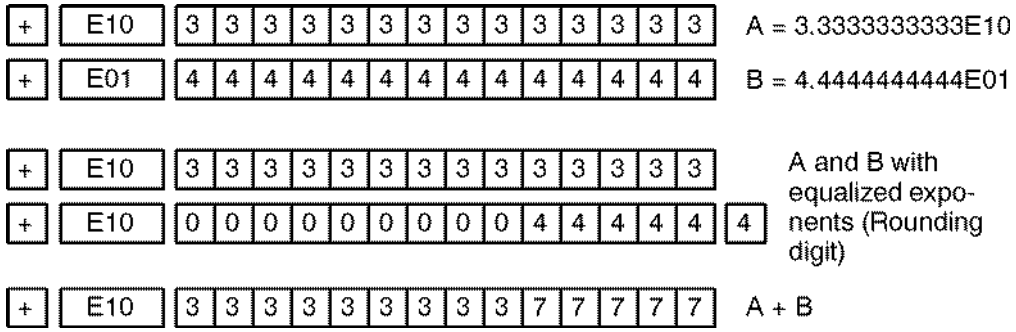
```
mod(x,y) = x - int(x/y) * y
```

(See "The Remainder Operation" on page 64 for details.) Trigonometric and transcendental functions use an enhanced version of `mod(`*x*`, pi/2)` during argument reduction. Therefore, although HP-UX math libraries perform extremely careful and accurate argument reduction, trigonometric functions like `cos(`*x*`)` can lose significance when *x* is large.

Adding Values with Very Different Magnitudes

When the system adds two floating-point values, it equalizes the operands' exponents before performing the calculation. It does so by right-shifting the smaller value so as to give it the same exponent as the larger. If the two values are very different in magnitude, this right shift causes a major loss of precision in the smaller value, as Figure 3-2 illustrates.

Figure 3-2 Adding Values with Very Different Magnitudes



In fact, if the difference between the exponents of two single-precision values is 25 or greater, the smaller value is right-shifted out of existence, and adding the two values results in no change at all in the larger value.

For example, consider a criminally minded American bank employee who arranges to have a penny transferred to his or her own account every so often. If the bank's computer uses single-precision floating-point arithmetic, and if the account starts with a balance of zero, then after almost twenty-three million transfers, when the bank account reaches $262,144.00, it will suddenly stop growing. Well before then, the employee may notice irregularities due to rounding errors. After 5 million transfers, the account will have more than 5 million pennies in it; but after 15 million transfers, it will have fewer than 14 million pennies. The larcenous employee may become frustrated with the bank's computer system, but it is behaving perfectly properly. (This irregularity is the reason why bank computers use COBOL's fixed-point arithmetic, not floating-point.)

The following example makes the same point more simply. It accumulates a sum by adding 0.01 to the starting value a thousand times, in single-precision, and then subtracting it.

Sample Program: diffmag1.f

```
      PROGRAM DIFFMAG1
      REAL X
      INTEGER I

      X = 0.01
      DO 10 I = 1, 1000
        X = X + 0.01
10    CONTINUE
      PRINT *, 'X is', X
      DO 20 I = 1, 1000
        X = X - 0.01
20    CONTINUE
      PRINT *, 'X is', X
      END
```

The result is not exact. Instead of 10.01 and 0.01, you are likely to get results similar to the following:

```
 X is 10.01013
 X is 9.99994E-03
```

The following example is even simpler. At higher magnitudes, adding 1 to a value has no effect at all.

Sample Program: diffmag2.f

```
      REAL BIG, SMALL, SUM, DIFF
      INTEGER I

      SMALL = 1.0

      DO I = 1, 10
        BIG = 10.0**I
        SUM = BIG + SMALL
        DIFF = ABS(BIG - SUM)

        PRINT 100, SUM, DIFF
      END DO

100   FORMAT(F14.1, F8.3)
      END
```

The addition of 1 stops affecting the result when the value reaches about 10^8:

```
          11.0   1.000
         101.0   1.000
        1001.0   1.000
       10001.0   1.000
      100001.0   1.000
     1000001.0   1.000
    10000001.0   1.000
   100000000.0    .000
  1000000000.0    .000
 10000000000.0    .000
```

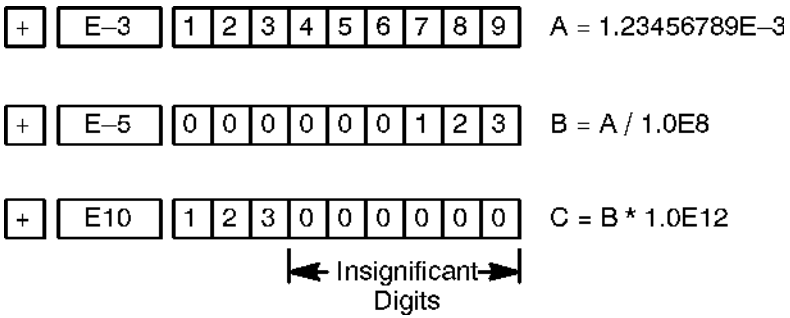
One way to minimize this kind of precision loss, if you can tolerate the added execution time, is to sort the elements of an array in ascending order before you add them together.

Unintentional Underflow

An underflow may occur when a calculation produces a result that is smaller in magnitude than the smallest normalized value. Sometimes a calculation can lose virtually all significance when it underflows, yielding a denormalized value that is then used in subsequent operations.

Suppose that the minimum decimal exponent of a system is –5, and that A is assigned the value 1.23456789E–3. Then the expression `A/1.0E8` would underflow, as Figure 3-3 shows.

Figure 3-3 Unintentional Underflow



Six significant digits are lost during the division, and they cannot be recovered by scaling the quotient back into the normalized range.

As another example, suppose the familiar form of the Pythagorean theorem is coded as

```
Z = SQRT(X**2 + Y**2)
```

Suppose further that this expression is executed using the following values for the single-precision variables X and Y:

```
X = 0.95830116E-20
Y = 0.25553963E-20
```

These values, when squared, produce the values 9.1834095E–41 and 6.5300508E–42 respectively, whose single-precision representations in hexadecimal are

```
0000ffff
00001234
```

Both of these values have only 16 bits of significance. The final result, 9.9178697E−21, is a reasonable-looking normalized number. However, because it is produced by a calculation that once lost all but 16 bits of significance, it can have at most 16 bits of significance itself. In fact, it actually has considerably less.

You can find out whether your application has underflowed by using the `fetestexcept` function. Alternatively, you can use the `fesettrapenable` function or the `+FP` compiler option to run the application with the underflow exception trap enabled. See "Exception Bits" on page 130 and "Command-Line Mode Control: The +FP Compiler Option" on page 146 for more information.

If you have a single-precision application that underflows frequently, you can solve the problem by changing to double-precision. If a double-precision application underflows frequently, you could migrate it to quad-precision, but at a considerable loss of efficiency; you may want to restructure your application instead so as to avoid underflows, if possible.

Truncation to an Integer Value

The **floor** of a value is the greatest whole number less than the value. The **ceiling** of a value is the smallest whole number greater than the value.

Rounding, precision mode problems, and compiler optimizations can all contribute to inaccurate results, but under most circumstances the inaccuracy is very small and not noticeable. However, some operations can magnify the inaccuracy of a calculation to the point where the result is meaningless. This can occur in algorithms that truncate a floating-point value to make it an integer. Truncating a positive floating-point value, for instance, reduces its magnitude to the floor integer, regardless of how close to the ceiling value it may be. An expression may yield 1.999999 on one system and 2.00001 on another. Both results may be acceptable in terms of expected rounding errors. However, if the result is truncated to an integer, these two systems will produce 1 and 2, respectively, which can be an unacceptable difference.

The following program provides a simple example of this situation.

Sample Program: trunc.c

```
#include <stdio.h>

int main(void)
{
    double x;
    int i, n;

    x = 1.5;
    for (i = 0; i < 10; i++) {
         n = x;
         printf("x is %g, n is %d\n", x, n);
         x += 0.1;
    }
}
```

No matter how close the value of `x` gets to 2.0, C conversion rules require the fractional part to be truncated. Therefore, the output of the program is as follows:

```
x is 1.5, n is 1
x is 1.6, n is 1
x is 1.7, n is 1
x is 1.8, n is 1
x is 1.9, n is 1
x is 2, n is 2
x is 2.1, n is 2
x is 2.2, n is 2
x is 2.3, n is 2
x is 2.4, n is 2
```

Many algorithms legitimately require truncation of results to integral values. One way to avoid the kind of problem illustrated by the preceding example is to add 0.5 to the result of a floating-point value that must be assigned to an integer:

```
n = x + 0.5;
```

Doing this will effectively round the result to the nearest integer (if `x` is greater than or equal to 0). Another solution is to call the function `rint`, which rounds a `double` value to the nearest integer:

```
n = rint(x);
```

If you use either of these solutions, the program output is

```
x is 1.5, n is 2
x is 1.6, n is 2
x is 1.7, n is 2
x is 1.8, n is 2
x is 1.9, n is 2
x is 2, n is 2
x is 2.1, n is 2
x is 2.2, n is 2
x is 2.3, n is 2
x is 2.4, n is 2
```

Ill-Conditioned Computations

If relatively small changes to the input of a program or to the intermediate results generated by a program cause relatively large changes in the final output, the program is said to be **ill-conditioned** or **numerically unstable**.

The following example illustrates an ill-conditioned program:

Sample Program: sloppy_tangent.f

```
C The following program is an example of how small
C perturbations in the argument to a function near the
C function's singularity can cause large variations in the
C function's result. A program may be ill-conditioned because
C of a case like this, where minor inaccuracies created during
C preliminary calculations become major inaccuracies when they
C are passed through a function at a point where the function
C has a very steep slope.

      PROGRAM SLOPPY_TANGENT
      DOUBLE PRECISION X

      X = 1.570796D0
      WRITE(*,*) 'TAN( X-1.0D-5 ):', TAN( X-1.0D-5 )
      WRITE(*,*) 'TAN( X ):       ', TAN( X )
      WRITE(*,*) 'TAN( X+1.0D-5 ):', TAN( X+1.0D-5 )
      END
```

The output from this program shows how small changes in the argument to the TAN function lead to wildly varying results:

```
 TAN( X-1.0D-5 ): 96835.46637430933
 TAN( X ):        3060023.306952844
 TAN( X+1.0D-5 ): -103378.351773411
```

Ill-conditioned computations cause trouble not because their input values may change, but because the seemingly innocuous rounding errors and loss of significance can have large effects on the final results.

One way to establish that rounding errors and loss of significance are causing a program to produce incorrect results is to run the program in various rounding modes. (See "Rounding Mode: fegetround and fesetround" on page 128 for information on how to change rounding modes using the `fesetround` function.) However, this technique does not always work. In the preceding example, for instance, changing the rounding mode has little effect on the results. However, if you do observe that simply changing the rounding mode causes large changes in the application results, then your application is most likely ill-conditioned.

A better technique, which does not require you to use additional functions or even to modify your code, is to make very small changes in the input data and to observe the amount by which the result changes. Wild swings in output magnitude may indicate an ill-conditioned application.

Fixing an ill-conditioned application requires a thorough understanding of the computations executed by the application. Chances are that the instability is caused by very few intermediate computations—possibly by just one. You must try to identify the location of the instability using your knowledge of the application, of the characteristics of the math functions called, and of the data being processed.

4 HP-UX Math Libraries on HP 9000 Systems

This chapter describes what libraries are and how to use them. It also provides detailed information about the math libraries on HP 9000 systems. It covers the following topics:

- HP-UX library basics
- Math library basics, including math library error handling
- HP-UX math library contents
- Calling C library functions from Fortran

For basic information on HP 9000 math libraries, see “Overview of HP-UX Math Libraries” on page 24.

HP-UX Library Basics

A **library** is a collection of commonly used functions, precompiled in object format and ready to be linked to an application. Because different programming languages have different calling conventions, there are separate libraries for various languages. On HP-UX systems, the C and C++ languages use one set of libraries, while the Fortran and Pascal languages use another.

In the HP-UX environment there are two types of libraries: archive libraries and shared libraries.

An **archive library** is a collection of object modules. When an application is linked with an archive library, the linker scans the contents of the archive library and extracts the object modules that satisfy any unresolved references in the application. The linker copies the archive library modules into the application's code section.

A **shared library** is also a collection of object modules. However, when the linker scans a shared library, it does not copy modules into the application's code section. Instead, the linker preserves information in the application's code section about which unresolved references were resolved in each shared library. At run time, the loader copies the referenced modules from the shared library into memory. If multiple applications linked with a common shared library execute simultaneously, they will all share (or be attached to) the same physical copy of the library in memory (hence the name **shared library**). The shared library improves the efficiency of memory use and allows smaller application binaries.

The name of an archive library is `lib`*`name`*`.a`, and the name of a shared library is `lib`*`name`*`.sl`. Thus the library named `m` (for math) can have versions named `libm.a` and `libm.sl`. The HP-UX system libraries are in the directory `/usr/lib`.

By default, the HP-UX linker selects a shared version of a library, if one is available. Although shared libraries save space in memory, using shared libraries makes a program run more slowly. If your application makes heavy use of math library functions, you may want to use archive libraries instead of shared libraries. For more information about performance issues related to shared and archive libraries, see "Shared Libraries versus Archive Libraries" on page 179.

For detailed information about archive libraries and shared libraries, see the HP-UX Linker and Libraries Online User Guide.

Math Library Basics

Math libraries in most computer systems, including HP-UX, are collections of frequently used mathematical functions. The functions take one or more arguments and return one or more results. When an application source file contains a use of a math function, the compiler automatically generates a call to the appropriate routine name in the appropriate math library. For example, suppose your Fortran program contains the following declaration and statement:

```
DOUBLE PRECISION A, B, Y
   .
   .
   .
Y = A**B
```

The statement raises `A` to the power `B` and assigns the result to `Y`. The Fortran compiler emits a call to `FTN_DTOD`, which is one of the functions in `libcl.a` and `libcl.sl`. `libcl` is the Fortran and Pascal library.

Conceptually, the definition of a math library function is simple; in this example, `FTN_DTOD` merely raises a `DOUBLE PRECISION` value to the power of another `DOUBLE PRECISION` value. However, there are practical questions about math library functions that the programmer should consider:

1. How accurate is the result returned?
2. How efficient is the function (that is, how fast does it run)?
3. What happens when an error occurs (for example, overflow or invalid arguments)?
4. What are the calling conventions for the function?
5. What functions are available?

The answers to questions 1 and 2 are implementation-dependent. The answers to questions 3, 4, and 5 are determined by a variety of programming environment specifications, such as XPG4.2.

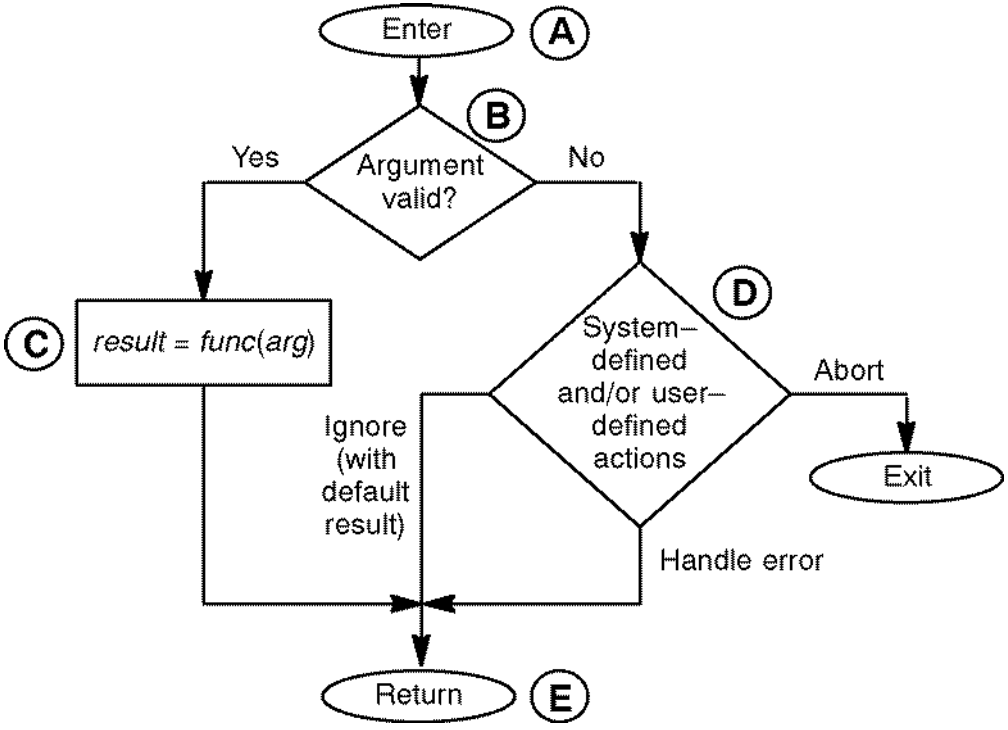
Appendix A partially answers question 5 by listing the functions in the HP-UX C math library. The Fortran equivalents are the intrinsic functions; see the *HP Fortran 90 Programmer's Reference* or the *HP FORTRAN/9000 Programmer's Reference* for a list of these functions.

The following section, which describes what happens when a program calls a math library function, provides some answers to questions 1 through 4.

Anatomy of a Math Library Function Call

Figure 4-1 shows a generalized flowchart of a math library function call, applicable to all languages and standards. The figure and the discussion that follows assume that the function takes one argument, but they also apply to functions that take more than one argument.

Figure 4-1 Anatomy of a Math Library Call



Steps A through E of this process are described below.

Step A

The compiler-generated call to a math library function includes code that

- Converts the argument to the required format, if necessary
- Places the argument in the appropriate place (that is, where the function expects it)
- Calls the function

Step B

The function determines whether the argument is valid. All functions check for NaN arguments and make additional checks specific to the function. For example, a logarithmic function checks for negative or zero arguments; an exponential function checks to see if the argument is so large that the result would overflow. If the argument is valid, control passes to Step C, execution of the function. If the argument is not valid, control passes to Step D.

Step C

The execution of the function involves performing the appropriate transformation on the argument(s). For most math functions, the result has the same data type as the argument, which is usually double-precision.

The execution of the function actually consists of one of many possible paths through the function code. The nature of the argument usually determines the path taken. For example, there may be special paths for particular argument ranges, or there may be an argument reduction phase. Math library implementors tune the most frequently executed paths for optimal efficiency. Consequently, arguments that are unusual (for example, very large or very small) can cause noticeable performance degradation.

Step D

If an argument proves to be invalid, control passes to the error-handling function of the library. When this happens, performance slows considerably; usually it is 2 to 3 orders of magnitude slower than for a valid argument. This follows from the basic assumption in math library design that errors are rare events.

If you do not supply an error-handling function in your program, a math library call that encounters an illegal argument does some or all of the following, depending on which programming language you are using and which standards are being enforced:

1. Supplies a system-defined default result: NaN for invalid operations, a huge value for overflows, and so on
2. Sets the globally accessible error code variable `errno`
3. Sets some state in the hardware floating-point status register
4. Prints an error message to `stderr`
5. Returns to the calling application, returning the default result

Your program may cause any of the above steps to be modified or stopped and may also provide an error-handling subroutine or function to be invoked. “Math Library Error Handling for C” on page 102 describes how these steps are performed for the C programmer in the XPG4.2 and SVID environments; “Math Library Error Handling for Fortran” on page 104 describes how they are performed for the Fortran programmer. Users of other languages should refer to the appropriate language manual for details.

Step E

When the function exits, the compiler-generated call to the function passes the result from where the function left it to where it is needed, converting it to the required format if necessary.

Math Library Error Handling for C

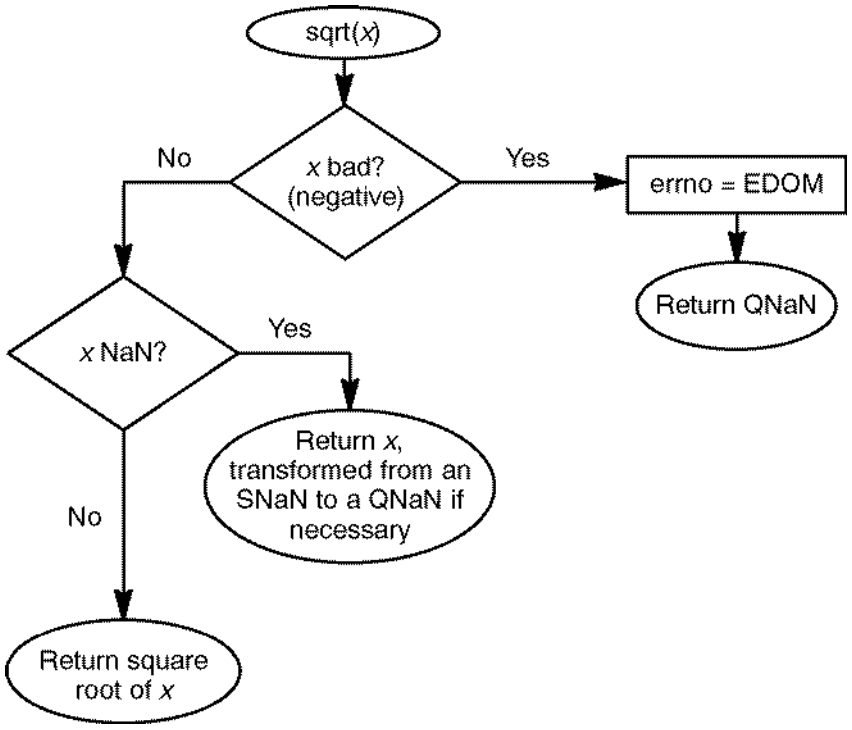
The XPG4.2 and SVID specifications specify similar math library error handling, so we describe them together. The XPG4.2 specification is a superset of the ANSI C standard.

To differentiate between XPG and SVID, HP-UX in the past implemented two separate C math libraries, `libm` and `libM`. The SVID libraries were `libm.a` and `libm.sl`; the XPG libraries were `libM.a` and `libM.sl`.

Because the SVID3 and XPG4.2 standards are essentially identical, HP-UX now supplies only one library, `libm`. The `libM` library is now obsolete. All versions of `libM` are provided only as soft links to the corresponding versions of `libm` (see "Locations of the Math Libraries at Release 10.30" on page 27 for details).

The flowchart in Figure 4-2 summarizes the error handling in the SVID and XPG4.2 math libraries. We use the `sqrt` function as an example.

Figure 4-2 C Math Library Error Handling for the sqrt Function



If a C library function such as `sqrt` encounters an invalid argument, it ordinarily returns a default result that indicates a failure—a result that the function could not ordinarily return. The default result is usually a NaN, zero, or `HUGE_VAL`, depending on the function and the argument value.

In addition, some functions set the global value `errno`, defined in the header file `errno.h`. (Many HP-UX system calls also set this value.) When a library call fails, the value of `errno` may be set to an appropriate code, also defined by `errno.h`. The math library functions set `errno` to

either EDOM or to ERANGE. EDOM indicates a domain error—that is, an error in the argument. ERANGE indicates a range error—usually an overflow in the result.

To detect errors, an application should check both the returned value and errno. It may set errno to 0 (that is, no error) at the beginning of a section of code and then examine it after one or more library calls to see if it is nonzero. A nonzero value indicates that some sort of library error has occurred.

For example, you could use the following C code fragment to print an error message when a call to pow fails:

```
errno = 0;
x = z * pow(y, w);
if (isnan(x) || errno)
    fprintf(stderr, "error in pow function: errno = %d\n", errno);
```

If y is negative and w is not an integer value, this fragment would print out

```
error in pow function: errno = 33
```

If you look up 33 in the system include file /usr/include/sys/errno.h, you find that it indicates a domain error (EDOM).

NOTE

C math library functions formerly used a function called matherr, which was required by the SVID2 specification but is not specified by ANSI C, SVID3, or XPG4.2. At HP-UX Release 10.30, this function is no longer provided.

Math Library Error Handling for Fortran

If a Fortran intrinsic function encounters an invalid argument, it returns a default result, just as a C function does. The default result depends on both the function and the nature of the argument. For example, a negative argument to the DLOG function causes it to return a NaN value. The most generally useful method of detecting errors is for the application to check for an anomalous result and then to take appropriate action.

What happens after the error depends on the following factors:

- Whether an exception trap for the error is enabled
- Whether the program contains an `ON EXTERNAL ERROR` statement (HP FORTRAN/9000 only)

We discuss the possible sequences of events in the following sections. The flowchart in Figure 4-3 summarizes HP FORTRAN/9000 math library error handling. The flowchart in Figure 4-4 summarizes HP Fortran 90 math library error handling.

Figure 4-3 Fortran 77 Math Library Error Handling

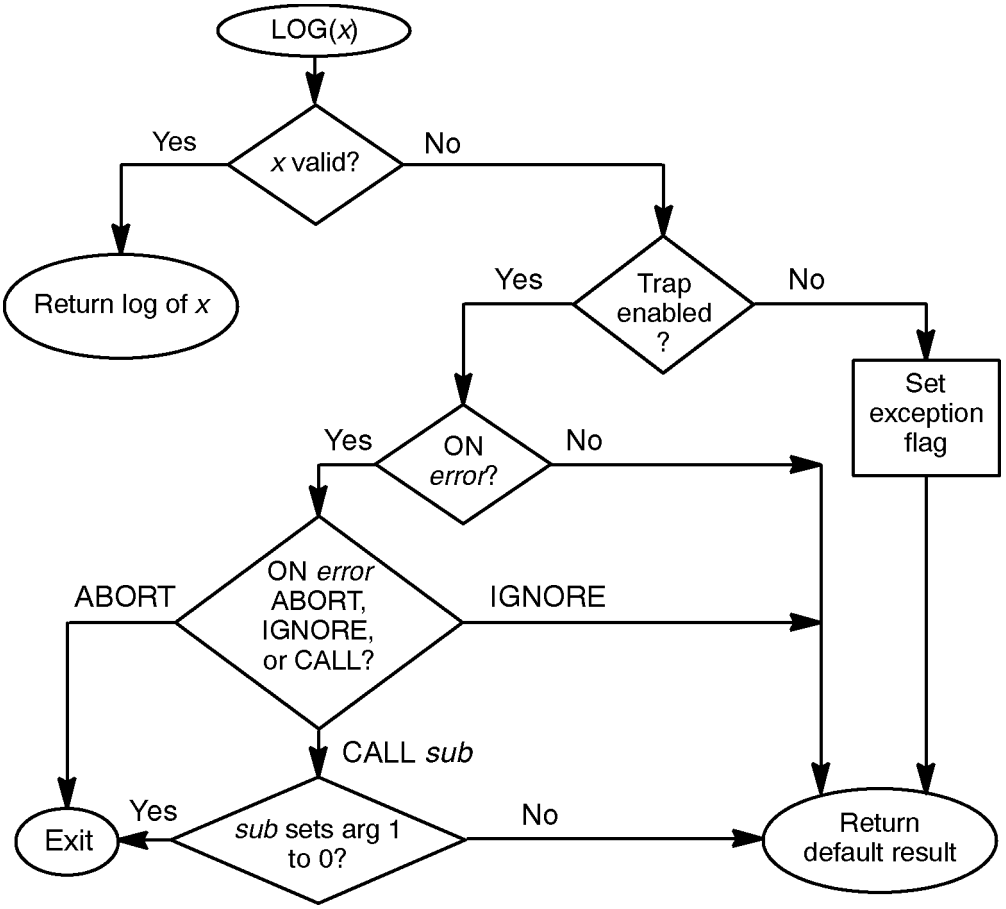
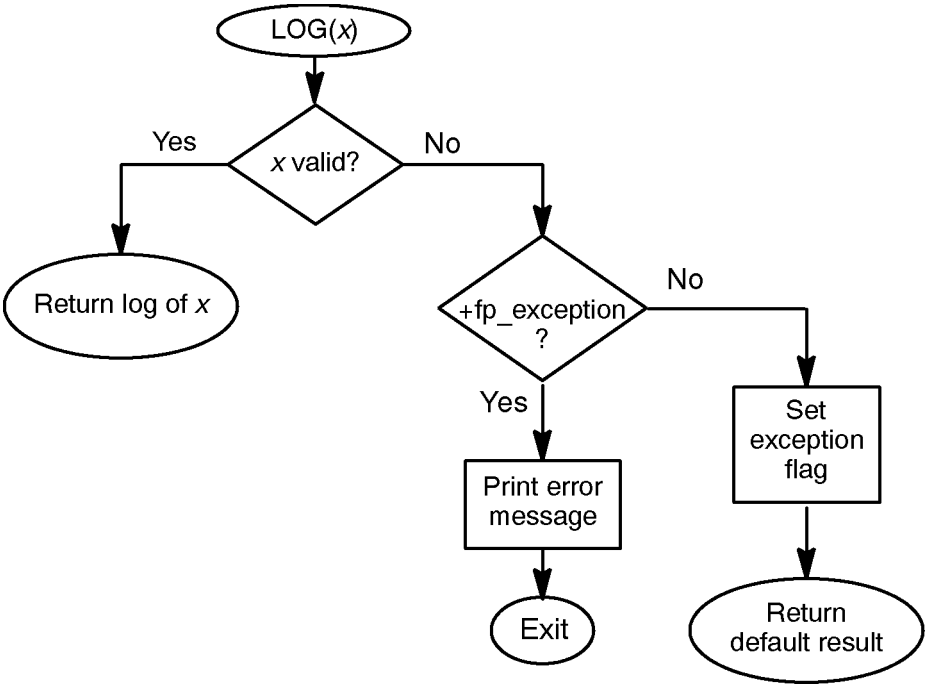


Figure 4-4 Fortran 90 Math Library Error Handling



Enabling Exception Traps for Invalid Operations

You can enable an exception trap for any of the five floating-point exception conditions described in “Exception Conditions” on page 51. The exception condition that indicates an invalid argument to a math library function is the invalid operation condition. You can enable a trap for this condition in any of several ways:

- By compiling with the option `+FPV`, which enables the invalid operation trap
- (HP Fortran 90) By compiling with the option `+fp_exception`, which enables traps for invalid operation, overflow, underflow, and division by zero.

- (HP FORTRAN/9000) By compiling with the option +T, which enables traps for invalid operation, overflow, underflow, and division by zero. (If your program contains an ON EXTERNAL ERROR statement, you must use this option.)
- By calling the `fesettrapenable` routine with the argument FE_INVALID, which enables the invalid operation trap

See Chapter 5 and Chapter 6 for information about all of these methods.

If you do not enable a trap for invalid operations, all that happens in the case of an invalid argument is that the invalid operation exception flag in the status register is set and the default result is returned. You can retrieve the value of the exception flags by calling the `fetestexcept` routine, described in "Exception Bits" on page 130.

The ON EXTERNAL ERROR Statement (HP FORTRAN/9000 only)

NOTE

HP Fortran 90 does not support the ON EXTERNAL ERROR statement, although it does support the ON statement for other kinds of error handling.

If you use +T to enable a trap for invalid operations, what happens next depends on whether your program contains an ON EXTERNAL ERROR statement. If it does not, the function merely returns the default result.

If your program contains an ON statement, you must compile with the +T option in order to enable trap handling. If your program contains an ON statement and you do not specify +T, you get a compile-time warning.

If your program does contain an ON statement, what happens depends on the action you specify in the statement. You can specify any of the following:

- ABORT. If you specify ABORT, the program exits.
- IGNORE (usually not a good idea). If you specify IGNORE, the default result is returned.
- CALL *sub* (call a subroutine). If you specify CALL *sub*, the user-defined trap-handling subroutine *sub* is called. The subroutine must have three or four arguments (four if the function takes two arguments); see the *HP FORTRAN/9000 Programmer's Guide* for details. If the subroutine sets the first argument to 0, the program exits. Otherwise, the default result is returned.

NOTE

A subroutine that handles a math library error takes a different number of arguments from a subroutine that handles an IEEE exception (as described in "Using the ON Statement (Fortran only)" on page 155). See the *HP FORTRAN/9000 Programmer's Guide* for details.

A Program Example

The following program illustrates Fortran 77 math library error handling. It calls `DLOG` with a negative argument, which is invalid.

See "Run-Time Mode Control: The fenv(5) Suite" on page 125 for information about the `fetestexcept` and `fegettrapenable` routines.

Sample Program: liberr77.f

```
C  HP FORTRAN/9000 only
C  Compile
C    1) As is;
C    2) With comment removed from ON EXTERNAL ERROR ABORT
C       statement, and with +T
C    3) With comment removed from ON EXTERNAL ERROR CALL MYSUB
C       statement, and with +T

      PROGRAM LIBERR77
$ALIAS FETESTEXCEPT = 'fetestexcept' (%val)
$ALIAS FEGETTRAPENABLE = 'fegettrapenable'

      INTEGER FE_INEXACT, FE_UNDERFLOW, FE_OVERFLOW
      INTEGER FE_DIVBYZERO, FE_INVALID, FE_ALL_EXCEPT
      PARAMETER (FE_INEXACT = Z'08000000')
      PARAMETER (FE_UNDERFLOW = Z'10000000')
      PARAMETER (FE_OVERFLOW = Z'20000000')
      PARAMETER (FE_DIVBYZERO = Z'40000000')
      PARAMETER (FE_INVALID = Z'80000000')
      PARAMETER (FE_ALL_EXCEPT = Z'f8000000')
      EXTERNAL FETESTEXCEPT, FEGETTRAPENABLE
      INTEGER  FETESTEXCEPT, FEGETTRAPENABLE
      DOUBLE PRECISION X, Y
      LOGICAL TEST
      INTEGER FLAGS, TRAPS

C  Abort if a function results in an error
C      ON EXTERNAL ERROR ABORT

C  Call mysub if a function results in an error
C  Other possible action is IGNORE
C  Setting I to 0 aborts the program
C      ON EXTERNAL ERROR CALL MYSUB
      X = 1.2345D0
      X = X*1.1D0
      Y = DLOG(0.0D0-X)
      FLAGS = FETESTEXCEPT(FE_ALL_EXCEPT)
      TRAPS = FEGETTRAPENABLE()
      WRITE(*,10) Y, FLAGS, TRAPS
      TEST = FLAGS .AND. FE_INVALID
      IF (TEST) THEN
        PRINT *, 'invalid operation occurred'
      ENDIF
 10   FORMAT(G, Z10.8, Z10.8)
      END

      SUBROUTINE MYSUB(I, A, X)
      INTEGER I
      DOUBLE PRECISION A, X
      PRINT *, 'error no. is ', I
      PRINT *, 'result is ', A
      PRINT *, 'arg is ', X
      I = 0
      RETURN
      END
```

If you compile this program with the ON statements commented out and without the +T option, no traps are enabled. Therefore, the math library error sets the appropriate exception flag. The output of the program is as follows:

```
$ f77 liberr77.f -lm
liberr77.f:
   MAIN liberr:
   mysub:
$ a.out
NaN                         88000000  00000000
 invalid operation occurred
```

The exception flag value indicates that both an invalid operation and an inexact result condition were generated. (Table 5-1 on page 127 shows the bit values for each of these conditions.).

If you compile with +T, the exception flags that correspond to the traps set by +T are not set after an error, and it is not useful to check them.

If you remove the comment from the ON EXTERNAL ERROR ABORT statement and compile with +T, the invalid operation trap is enabled. When you run the program, it exits:

```
$ f77 +T liberr77.f -lm
liberr77.f:
   MAIN liberr:
   mysub:
$ a.out
$
```

Finally, if you remove the comment from the ON EXTERNAL ERROR CALL MYSUB statement and compile with +T, the program calls the error-handling subroutine MYSUB before exiting:

```
$ f77 +T liberr77.f -lm
liberr77.f:
   MAIN liberr:
   mysub:
$ a.out
 error no. is  8
 result is  NaN
 arg is  -1.35795
$
```

HP does not support the use of the ON statement to handle math library errors in Fortran 90. The following is a Fortran 90 version of the program.

Sample Program: liberr90.f

```
C  Compile
C    1) As is;
C    2) With the +fp_exception option

      PROGRAM LIBERR90
!$HP$ ALIAS FETESTEXCEPT = 'fetestexcept' (%val)
!$HP$ ALIAS FEGETTRAPENABLE = 'fegettrapenable'

      PARAMETER (FE_INEXACT = Z'08000000')
      PARAMETER (FE_UNDERFLOW = Z'10000000')
      PARAMETER (FE_OVERFLOW = Z'20000000')
      PARAMETER (FE_DIVBYZERO = Z'40000000')
      PARAMETER (FE_INVALID = Z'80000000')
      PARAMETER (FE_ALL_EXCEPT = Z'f8000000')
      EXTERNAL FETESTEXCEPT, FEGETTRAPENABLE
      INTEGER  FETESTEXCEPT, FEGETTRAPENABLE
      DOUBLE PRECISION X, Y
      INTEGER FLAGS, TRAPS

      X = 1.2345D0
      X = X*1.1D0
      Y = DLOG(0.0D0-X)
      FLAGS = FETESTEXCEPT(FE_ALL_EXCEPT)
      TRAPS = FEGETTRAPENABLE()
      WRITE(*,10) Y, FLAGS, TRAPS
 10   FORMAT(G, Z10.8, Z10.8)
      END
```

If you compile the program as we show it, it runs as follows. Because no traps are enabled, the function returns a NaN and sets the appropriate exception flag.

```
$ f90 liberr90.f -lm
liberr90.f
   program LIBERR90

29 Lines Compiled
$ ./a.out
NaN                          88000000  00000000
```

If you compile the program with the `+fp_exception` option, the invalid operation trap is enabled, and the program aborts with an error message:

```
$ f90 +fp_exception liberr90.f -lm
liberr90.f
program LIBERR90

15 Lines Compiled
$ ./a.out
PROGRAM ABORTED :  IEEE invalid operation

PROCEDURE TRACEBACK:
( 0)  0x00009a0c   _start + 0x64  [./a.out]
```

Contents of the HP-UX Math Libraries

This section describes in some detail the contents of the math libraries. These libraries include:

- Scalar math libraries (`libm` and `libcl`)
- The BLAS library `libblas` (provided with the HP Fortran 90 and HP FORTRAN/9000 products only)
- The vector library `libvec` (provided with the HP FORTRAN/9000 product only) (obsolete)

The math libraries run well on both PA-RISC 1.1 and PA-RISC 2.0 systems. HP does not provide PA2.0 versions of the math libraries, although it does provide PA2.0 versions of millicode functions. See “Millicode Versions of Math Library Functions” on page 112 for details.

Scalar Math Libraries (libm and libcl)

The scalar math libraries are implemented using the PA-RISC 1.1 instruction set. The most important functions are carefully optimized.

For even faster performance, a number of frequently used math functions are implemented in the millicode library (`/usr/lib/milli.a`) as well as in the standard math library. See “Millicode Versions of Math Library Functions” for details.

Millicode Versions of Math Library Functions

Several of the most frequently used math functions are implemented in the millicode library as well as in the math library. The millicode versions have a streamlined calling sequence and are usually faster than their counterparts in the math library.

Millicode versions exist for the following Fortran and C functions. They have double-precision versions only, unless otherwise specified.

```
acos
asin
atan
atan2
cos       (single-precision also)
exp
log       (single-precision also)
log10
pow
sin       (single-precision also)
tan       (single-precision also)
```

Millicode versions exist for the following Pascal functions:

```
arctan
cos
exp
ln
sin
```

To get the millicode versions of any of these functions, compile your program with

- Any optimization level (0 through 4)
- The `+Olibcalls` or the `+Oaggressive` optimization option

With the `f90` and `f77` compiler commands, the `+Olibcalls` option is the default at optimization level 2 and above.

The `+Olibcalls` option is invoked by default when you specify the optimization type `+Oaggressive`; use `+Oaggressive +Onolibcalls` if you want aggressive optimization without using millicode routines. The millicode versions are implemented in the library `/usr/lib/milli.a`.

The millicode versions of functions do not provide standard-conforming error handling. This has different implications for different languages. In C programs, if an error occurs, the millicode versions return the same values as their standard library counterparts, but they do not set `errno`. Because the C and Pascal standards specify error handling for library functions, you should use millicode versions in C and Pascal programs only if your program does not require standard-conforming error handling. The Fortran standards do not specify error handling for math intrinsic functions, so using the millicode versions has no effect on standards compliance. See “Optimizing Your Program” on page 169 for more information about optimization options.

The C Math Library (libm)

NOTE

The `libM` math library is obsolete. See "Overview of HP-UX Math Libraries" on page 24 for details.

The C math library, `libm`, also supports

- `float` versions of many mathematical functions
- Degree-valued trigonometric functions
- A group of functions and macros recommended by the IEEE standard (see Table 2-12 on page 65), including floating-point classification macros, and some additional classification macros approved by the ISO/ANSI C committee for inclusion in the C9X draft standard
- A group of functions required by the COSE Common API Specification (Spec 1170) and by the XPG4.2 specification
- The *fenv*(5) suite, a collection of functions (approved by the ISO/ANSI C committee for inclusion in the C9X draft standard) that allow an application to manipulate the floating-point status register

All of these functions are defined in `math.h`. However, all of these functions are outside the ANSI C specification, and many of them are outside the XPG4.2 specification. If you compile in strict ANSI mode (with the `-Aa` option), only the declarations of functions specified by the ANSI C standard are ordinarily visible to your program. Therefore, to make the additional functions visible, do one of the following:

- Compile with the extended ANSI option `-Ae`, which makes both ANSI and non-ANSI function declarations visible. This option is the default, so you need not specify it explicitly:

  ```
  cc program_name.c -lm
  ```

- If your program does not use ANSI C features, compile in compatibility mode (with the `-Ac` option), which also makes both ANSI and non-ANSI function declarations visible:

  ```
  cc -Ac program_name.c -lm
  ```

  Do not use this option to compile programs that call `float` type math functions, because they depend on ANSI C features. See "float Type Math Functions" on page 115.

- Compile with the basic ANSI option `-Aa` and use the `-D` option to define the macro `_HPUX_SOURCE` on the command line. If this macro is defined, all the function declarations are visible. (If you compile with `-Ae` or `-Ac`, this macro is defined automatically.)

  ```
  cc -Aa -D_HPUX_SOURCE program_name.c -lm
  ```

NOTE

To compile C++ programs that call nonstandard math library functions, use the `+a1` option.

For a complete list of the contents of `libm`, see Appendix A. The functions are described in online man pages.

float Type Math Functions

Certain `libm` math functions are implemented as single-precision functions, accepting `float` arguments and returning a `float` result. On HP 9000 systems, the performance of single-precision and double-precision functions is similar. If your application uses single-precision data types heavily, you may find it convenient to use the `float` versions of the math functions the application calls.

The names of the `float` functions are the same as those of their double-precision counterparts but with the letter `f` appended. (For `float` versions of degree-valued trigonometric functions, the `f` follows the `d`.) This naming convention accords with that specified in Section 4.13 of the ANSI C standard, “Future Library Directions.”

The supplied functions are as follows:

- `cosf`, `sinf`, `tanf`, `cosdf`, `sindf`, `tandf`
- `acosf`, `asinf`, `atanf`, `atan2f`, `acosdf`, `asindf`, `atandf`, `atan2df`
- `coshf`, `sinhf`, `tanhf`
- `expf`, `fabsf`, `fmodf`, `logf`, `log10f`, `log2f`, `powf`, `sqrtf`, `cbrtf`
- `copysignf`

You must compile in ANSI mode (with `-Ae`, the default, or with `-Aa -D_HPUX_SOURCE`) to use these functions. If you compile in ANSI mode, ANSI C rules are followed, and the argument is treated as a `float` throughout the function call. If you do not compile in ANSI mode, traditional K&R C rules are followed, the compiler generates code to

promote the argument to `double`, and the function call either generates a linker error or produces incorrect results. Therefore, use a command line like the following:

```
cc program_name.c -lm
```

```
CC +a1 program_name.C -lm
```

Degree-Valued Trigonometric Functions

The Fortran math library defines a set of trigonometric functions whose arguments or results are specified in degrees rather than radians. These functions are also implemented in `libm`. The names of these functions are the same as the names of the standard versions but with the letter `d` appended: `sind`, `cosd`, and so on. (For `float` versions of degree-valued trigonometric functions, the `f` follows the `d`.)

To use these functions, compile them in any mode except strict ANSI mode (`-Aa`). Use either extended ANSI mode (`-Ae`, the default), non-ANSI mode (`-Ac`), or `-Aa -D_HPUX_SOURCE`.

Floating-Point Classification Macros

The `fpclassify` macro is the HP version of the `class` function recommended by the IEEE standard. The ANSI/ISO C committee has also approved this macro for inclusion in the C9X standard. The `fpclassify` macro accepts either a `double` or a `float` argument. It returns an integer value that describes the class of the argument—that is, what kind of floating-point value it is. Table 4-1 shows the values and their meanings, which are defined in `math.h`.

Table 4-1 fpclassify Values

Class	Name of Macro
Normalized	FP_NORMAL
Zero	FP_ZERO
Infinity	FP_INFINITE
Denormalized	FP_SUBNORMAL
NaN	FP_NAN

The list of classes is exhaustive: all IEEE floating-point values fall into one of these classes. The `fpclassify` macro never causes an exception, regardless of the operand. It is useful, therefore, for classifying an operand without risking an exception trap.

Use the `signbit` macro to determine whether a value is negative or positive.

To use the `fpclassify` macro, compile your program in any mode except strict ANSI mode (`-Aa`). Use either extended ANSI mode (`-Ae`, the default), non-ANSI mode (`-Ac`), or `-Aa -D_HPUX_SOURCE`.

The following C function shows how you might use `fpclassify` and `signbit` to display the class and value of a double-precision number:

Sample Function: print_class.c

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

typedef union {
  double y;
  struct {
     unsigned int ym, yl;
  } i;
} DBL_INT;

void print_classd(DBL_INT di)
{
    int class, sign;
    char *posneg[] = {"positive", "negative"};

    class = fpclassify(di.y);
    sign = signbit(di.y);

    if (class == FP_NORMAL)
        printf("%19.17g is %s normalized (%08x%08x)\n", di.y,
               posneg[sign], di.i.ym, di.i.yl);
    else if (class == FP_ZERO)
        printf("%19.17g is %s zero (%08x%08x)\n", di.y,
               posneg[sign], di.i.ym, di.i.yl);
    else if (class == FP_INFINITE)
        printf("%19.17g is %s infinity (%08x%08x)\n", di.y,
               posneg[sign], di.i.ym, di.i.yl);
    else if (class == FP_SUBNORMAL)
        printf("%19.17g is %s denormalized (%08x%08x)\n", di.y,
               posneg[sign], di.i.ym, di.i.yl);
    else if (class == FP_NAN)
        printf("%19.17g is NaN (%08x%08x), sign bit is %d\n",
               di.y, di.i.ym, di.i.yl, sign);
}
```

Other macros that test the class of a floating-point value are

- `isinf`, which tests whether a value is an infinity
- `isnan`, which tests whether a value is a NaN
- `isfinite`, which tests whether a value is neither infinity nor NaN
- `isnormal`, which tests whether a value is normalized

See the online man pages for information about these macros.

COSE Common API Functions

The C math library provides several functions that are not specified by the ANSI C standard but that are required by the COSE Common API Specification (Spec 1170) and the XPG4.2 specification. These functions are

`acosh(`*x*`)`	Returns inverse hyperbolic cosine of *x*
`asinh(`*x*`)`	Returns inverse hyperbolic sine of *x*
`atanh(`*x*`)`	Returns inverse hyperbolic tangent of *x*
`cbrt(`*x*`)`	Returns cube root of *x*
`expm1(`*x*`)`	Returns exp(*x*) - 1
`ilogb(`*x*`)`	Returns the integer form of the binary exponent of the floating-point value *x*
`log1p(`*x*`)`	Returns log(1 + *x*)
`logb(`*x*`)`	Returns the exponent of *x* as an integer-valued double-precision number
`nextafter(`*x*`, `*y*`)`	Returns the next representable neighbor of *x* in the direction of *y*
`remainder(`*x*`, `*y*`)`	Returns exact floating-point remainder as defined by IEEE standard

`rint(`*x*`)`	Rounds *x* to integer-valued double-precision number, in the direction of the current rounding mode
`scalb(`*x*`, `*n*`)`	Returns *x**(2***n*), computed efficiently

To use these functions, compile your program in any mode except strict ANSI mode (`-Aa`). Use either extended ANSI mode (`-Ae`, the default), non-ANSI mode (`-Ac`), or `-Aa -D_HPUX_SOURCE`.

The fenv(5) Suite

The *fenv*(5) suite is a collection of functions in the C math library that allow an application to manipulate various modifiable control modes and status flags in the floating-point status register. Most of the functions in the *fenv*(5) suite have been approved by the ISO/ANSI C committee for inclusion in the C9X draft standard.

For details on how to use these functions, see Chapter 5 and the online man pages. The tables in Appendix A also list all of the functions.

The BLAS Library (libblas)

The Basic Linear Algebra Subroutine (BLAS) library routines perform low-level vector and matrix operations. They have been tuned for maximum performance.

The BLAS library is provided with the HP Fortran 90 and HP FORTRAN/9000 products only, but the routines in this library are callable from other languages than Fortran.

To call BLAS library routines, use the `-l` compile-line option to link in the `libblas` library. For example, the following command line links a Fortran program with the BLAS library:

```
f90 prog.f -lblas
```

To link with the library from C, you must also specify the library path name on the command line.

For more information about the BLAS library routines, see “Matrix Operations” on page 183 and the *HP Fortran 90 Programmer's Reference* or the *HP FORTRAN/9000 Programmer's Reference*. In addition, online man pages for these routines are available.

NOTE To obtain the man pages for the BLAS library, you must have `/opt/fortran90/share/man` in your `MANPATH` (or `/opt/fortran/share/man`). Type `man blas` for an overview.

The Vector Library (libvec) (Obsolete)

The **vector library** (provided with the HP FORTRAN/9000 product only) performs vector and matrix operations.

NOTE This library is obsolete. It was formerly used by the FORTRAN Optimizing Preprocessor (FTNOPP). The performance benefits provided by FTNOPP are now supplied by the compiler when you use the `+Ovectorize` option with either C or Fortran programs at optimization level 3 or above. (See “Optimizing Your Program” on page 169 for details.) The vector library is provided for compatibility reasons only, in `/opt/fortran/old/lib/libvec.a`.

For more information about the `libvec` routines, see the *HP FORTRAN/9000 Programmer's Reference*.

Calling C Library Functions from Fortran

To call a C math library function from a Fortran program, you must do the following:

1. Use an `!$HP$ ALIAS` directive (Fortran 90) or an `$ALIAS` (FORTRAN 77) directive to tell the compiler that the function's arguments are passed by value.
2. Declare the function with the correct return value. See the online reference pages, Appendix A, or `/usr/include/math.h` to find the return value.
3. Link in the C math library (`libm`).

For example, the following Fortran program calls `j0`, one of the Bessel functions in the C math library:

Sample Program: bessel.f

```
C  $HP$ ALIAS directive tells the compiler to use C language
C    argument-passing conventions (Fortran 90)
C  $ALIAS is f77 version
C  Program declares j0() DOUBLE PRECISION

!$HP$ ALIAS J0 = 'j0'(%VAL)
C $ALIAS J0 = 'j0'(%VAL)
       PROGRAM BESSEL
       DOUBLE PRECISION A, B, J0

       A = 1.0
       B = J0(A)

       WRITE(*,*) "Bessel of", A, " is", B
       END
```

The `%val` argument indicates that the argument is passed by value. For details on the `!$HP$ ALIAS` and `$ALIAS` directives, see the *HP Fortran 90 Programmer's Reference* or the *HP FORTRAN/9000 Programmer's Guide*.

You can compile and run the program as follows:

```
$ f90 bessel.f -lm
bessel.f
   program BESSEL

16 Lines Compiled
$ ./a.out
Bessel of 1.0 is .7651976865579666
```

5 Manipulating the Floating-Point Status Register

The **floating-point status register** (also known as the floating-point control register) stores information about several aspects of the floating-point environment:

- The rounding mode
- What traps are enabled—that is, what exceptions your program can catch
- If traps are not enabled, what exceptions have occurred
- Whether flush-to-zero underflow mode is set (for systems that have this capability)
- The model and revision of the system's **floating-point unit (FPU)** (also called the floating-point coprocessor)

HP-UX systems provide facilities that allow you to manipulate the status register. The *fenv*(5) suite, a group of C math library functions, allows you to retrieve any of this information or to modify the environment. The +FP compiler option allows you to specify on the command line the traps to enable for a particular program and the underflow mode.

Run-Time Mode Control: The fenv(5) Suite

This section describes the *fenv*(5) suite of functions, a collection of services provided in the C math library that allow an application to manipulate several modifiable control mode and status flags in the floating-point status register. These functions and their associated parameter types are declared in the header file `/usr/include/fenv.h`.

NOTE

The *fenv*(5) suite replaces the *fpgetround*(3M) suite of functions.

You can call these functions from Fortran programs. See "A Program Example" on page 108 for examples.

The *fenv*(5) suite contains the following functions:

- `fegetenv` and `fesetenv`, which retrieve and set the floating-point environment
- `feholdexcept` and `feupdateenv`, which can be used to hide spurious exceptions
- `fegetexceptflag` and `fesetexceptflag`, which retrieve and set the accrued exception flags
- `feraiseexcept`, `fetestexcept`, and `feclearexcept`, which raise, test, and clear exceptions
- `fegetround` and `fesetround`, which retrieve and set the rounding mode
- `fegettrapenable` and `fesettrapenable`, which retrieve and set the exception trap enable bits
- `fegetflushtozero` and `fesetflushtozero`, which retrieve and set the underflow mode

All of these functions except the last four (`fegettrapenable`, `fesettrapenable`, `fegetflushtozero`, and `fesetflushtozero`) have been approved by the ANSI/ISO C committee for inclusion in the C9X standard.

NOTE

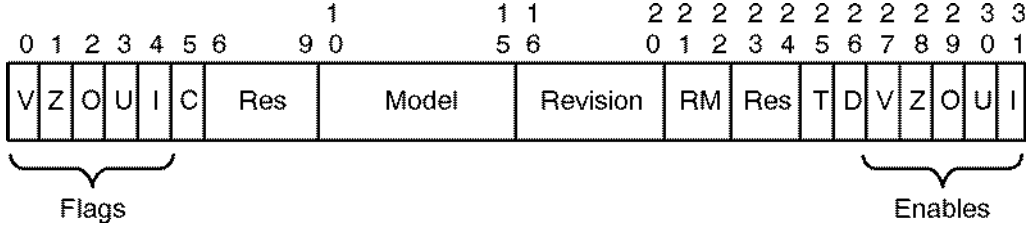
Be careful if you use these functions at higher optimization levels (2 and above). Optimization may change the order of operations in a program, so that a call to one of these functions may be placed after an operation you want the function to affect, or before an operation whose result you want the function to check. These functions will then produce unexpected results. (If it is possible to isolate the part or parts of your program that call these functions, you could place them in a separate module and compile it at a lower optimization level than the rest of the program. You may also use the `FLOAT_TRAPS_ON` pragma to suppress optimization for part of your program.)

The PA-RISC Floating-Point Status Register

The PA-RISC floating-point status register is `fr0L`, the left half of `fr0`.

Figure 5-1 shows the structure and contents of `fr0L`. Fields marked `Res` are reserved for future use.

Figure 5-1 PA-RISC Floating-Point Status Register (fr0L)



PA-RISC 2.0 Architecture and the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* describe the contents of `fr0L` in detail. The fields manipulated by the *fenv*(5) functions are as follows:

Flags **Exception flags**. A flag bit is associated with each IEEE exception. If the corresponding enable bit is not set, the floating-point unit sets an exception flag to 1 when the corresponding exception occurs, but does not cause a trap. Table 5-1 shows the bit names and the corresponding exceptions. The functions `fegetexceptflag` and `fesetexceptflag` manipulate these flags.

Table 5-1 IEEE Exception Bits

Bit Name	Description
V	Invalid operation
Z	Division by zero
O	Overflow
U	Underflow
I	Inexact result

Enables **Exception trap enable bits**. An enable bit is associated with each IEEE exception. When an enable bit equals 1, the corresponding trap is enabled. When an enable bit equals 0, the corresponding IEEE exception sets the corresponding flag to 1 instead of causing a trap. The functions `fegettrapenable` and `fegettrapenable` manipulate these bits.

RM Rounding mode for all floating-point operations. The values corresponding to each rounding mode are shown in Table 5-2. Note that these values are similar but not identical to the values used by the `fegetround` and `fesetround` functions (see “Rounding Mode: fegetround and fesetround” on page 128). The functions `fegetround` and `fesetround` manipulate these bits.

Table 5-2 Rounding Modes

Rounding Mode	Description
0	Round to nearest
1	Round toward zero
2	Round toward +infinity
3	Round toward –infinity

D	The D bit. When this bit is set to 1, flush-to-zero underflow mode is enabled (see "Underflow Mode: fegetflushtozero and fesetflushtozero" on page 143). The functions `fegetflushtozero` and `fesetflushtozero` manipulate this bit.

The fields not manipulated by the *fenv*(5) functions are as follows:

C	The Compare bit.
Res	Reserved for future use.
Model, Revision	The Model and Revision fields contain values that correspond to various implementations of HP 9000 floating-point coprocessors.
T	The Delayed Trap bit.

Rounding Mode: fegetround and fesetround

The functions `fegetround` and `fesetround` allow you to retrieve or change the rounding mode for floating-point operations. The declarations for these functions are as follows:

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

The value returned by `fegetround` and the argument of `fesetround` match one of the rounding direction macros defined in `fenv.h`. The rounding direction macros are as follows:

`FE_TONEAREST`	Round to nearest
`FE_TOWARDZERO`	Round toward zero
`FE_UPWARD`	Round toward positive infinity
`FE_DOWNWARD`	Round toward negative infinity

The `fegetround` function returns the current rounding mode. By default, the rounding mode is `FE_TONEAREST`.

The `fesetround` function sets the rounding mode to the specified value. It returns a nonzero value if and only if the argument matches a rounding direction macro.

The following C program uses `fesetround` and `fegetround` to show how the rounding mode affects the result of an overflow (see Table 2-11 on page 55).

Sample Program: fe_round.c

```
/*************************************************************/
#include <stdio.h>
#include <fenv.h>

int main(void)
{
  int save_rnd, rnd;
  double x, y, z;

  save_rnd = fegetround();
  if (save_rnd == FE_TONEAREST)
    printf("rounding direction is FE_TONEAREST\n");
  else
    printf("unexpected rounding direction\n");

  x = 1.79e308;
  y = 2.2e-308;
  z = x / y;                        /* overflow */
  printf("%g / %g = %g\n", x, y, z);

  x = -1.79e308;
  y = 2.2e-308;
  z = x / y;             /* negative overflow */
  printf("%g / %g = %g\n", x, y, z);

  fesetround(FE_TOWARDZERO);
  rnd = fegetround();
  if (rnd == FE_TOWARDZERO)
    printf("rounding direction is FE_TOWARDZERO\n");
  else
    printf("unexpected rounding direction\n");

  x = 1.79e308;
  y = 2.2e-308;
  z = x / y;                        /* overflow */
  printf("%g / %g = %g\n", x, y, z);

  fesetround(FE_UPWARD);
  rnd = fegetround();
  if (rnd == FE_UPWARD)
    printf("rounding direction is FE_UPWARD\n");
  else
    printf("unexpected rounding direction\n");
                                                    /* continued */
/*************************************************************/
```

Sample Program: fe_round.c (cont.)

```
/************************************************************/
  x = -1.79e308;
  y = 2.2e-308;
  z = x / y;              /* negative overflow */
  printf("%g / %g = %g\n", x, y, z);

/* return to round-to-nearest */
  fesetround(save_rnd);
  rnd = fegetround();
  if (rnd == FE_TONEAREST)
    printf("rounding direction is FE_TONEAREST\n");
  else
    printf("unexpected rounding direction\n");
}
/************************************************************/
```

If you compile and run this program, it produces the following output:

```
$ cc fe_round.c -lm
$ ./a.out
rounding direction is FE_TONEAREST
1.79e+308 / 2.2e-308 = inf
-1.79e+308 / 2.2e-308 = -inf
rounding direction is FE_TOWARDZERO
1.79e+308 / 2.2e-308 = 1.79769e+308
rounding direction is FE_UPWARD
-1.79e+308 / 2.2e-308 = -1.79769e+308
rounding direction is FE_TONEAREST
```

See “IEEE Rounding Modes” on page 52 for more information about rounding modes on HP 9000 systems.

Exception Bits

The IEEE-754 standard specifies five floating-point exceptions: divide-by-zero, overflow, underflow, inexact result, and invalid operation. If one of these five exceptions occurs and the corresponding **exception trap enable bit** is set to 1, the trap takes place, and a SIGFPE signal is generated. If an exception occurs and the exception trap enable bit is set to 0, the corresponding **exception flag** is set to 1 and no trap takes place. The exception trap enable bits are sometimes called **mask bits**; the exception flags are sometimes called **sticky bits**. The term **sticky** follows from the fact that once an exception flag is set by a disabled exception, it remains set for the life of the process unless it is cleared by the application (for example, by a call to `fesetexceptflag` or `feclearexcept`).

The C library supplies a group of functions, all approved for inclusion in the C9X draft standard, to manipulate the exception flags. The library also supplies two functions, not approved for the C9X draft standard and specific to HP, to manipulate the exception trap enable bits. The following subsections discuss these groups of functions.

The functions use the following exception macros, defined in `fenv.h`, for the exception flags and the exception trap enable bits:

`FE_INEXACT`	Inexact result
`FE_UNDERFLOW`	Underflow
`FE_OVERFLOW`	Overflow
`FE_DIVBYZERO`	Divide by zero
`FE_INVALID`	Invalid operation
`FE_ALL_EXCEPT`	All exceptions

For details about using these functions to detect and handle floating-point exceptions, see “Using the fesettrapenable Function” on page 152 and “Detecting Exceptions without Enabling Traps” on page 162.

Manipulating the Exception Flags: fegetexceptflag, fesetexceptflag, fetestexcept, feraiseexcept, feclearexcept

The functions `fegetexceptflag` and `fetestexcept` save and restore the current settings of the exception flags. The `fesetexceptflag` function sets the flags to a previously saved state. The `feraiseexcept` function raises exceptions, and the `feclearexcept` function clears the exception flags.

The C declarations for these functions are as follows:

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

The `fegetexceptflag` function stores the desired exception flags (as indicated by the argument *excepts*, which can be any bitwise OR of the exception macros) in the object pointed to by the argument *flagp*.

The `fesetexceptflag` function sets the status for the exception flags indicated by the argument *excepts* according to the representation in the object pointed to by *flagp*. Use it to reset the exception flags to a previously saved state.

The `fetestexcept` function determines which of a specified subset of the exception flags are currently set.

The `feraiseexcept` function raises the exceptions represented by its argument (and causes traps if the corresponding traps are enabled). It allows you to raise exceptions directly without having to write operations in your program that generate an exception.

The `feclearexcept` function clears the exception flags represented by its argument.

The following program uses all these functions. First it calls `fegetexceptflag` to retrieve the initial settings of the exception flags. Then it calls `feraiseexcept` to raise a couple of exceptions, and calls `fetestexcept` to verify that the flags are set correctly. Then it calls `feclearexcept` to clear the accumulated flags, and after another call to `fetestexcept` it calls `feraiseexcept` again to set the underflow exception. The next call to `fetestexcept` shows that, as with basic operations, raising the underflow exception also raises the inexact exception on HP systems. Finally, the program calls `fesetexceptflag` to restore the initial state of the exception flags.

Sample Program: fe_flags.c

```
/*************************************************************/
#include <stdio.h>
#include <fenv.h>

int main(void)
{
  fexcept_t flags;
  int excepts;
  void print_flags(int);

  fegetexceptflag(&flags, FE_ALL_EXCEPT);
  printf("at start:\n");
  print_flags(flags);

  /* raise divide-by-zero exception */
  feraiseexcept(FE_DIVBYZERO);
  excepts = fetestexcept(FE_DIVBYZERO | FE_INEXACT);
  printf("after raising divide-by-zero exception:\n");
  print_flags(excepts);
                                                  /* continued */
/*************************************************************/
```

Sample Program: fe_flags.c (cont.)

```
/*************************************************************/
  /* raise divide-by-zero exception */
  feraiseexcept(FE_DIVBYZERO);
  excepts = fetestexcept(FE_DIVBYZERO | FE_INEXACT);
  printf("after raising divide-by-zero exception:\n");
  print_flags(excepts);

  /* raise inexact exception */
  feraiseexcept(FE_INEXACT);
  excepts = fetestexcept(FE_DIVBYZERO | FE_INEXACT);
  printf("after raising inexact exception:\n");
  print_flags(excepts);

  /* clear exceptions, retrieve new setting */
  feclearexcept(FE_ALL_EXCEPT);
  excepts = fetestexcept(FE_ALL_EXCEPT);
  printf("after clearing exceptions:\n");
  print_flags(excepts);

  /* raise underflow exception; inexact also set */
  feraiseexcept(FE_UNDERFLOW);
  excepts = fetestexcept(FE_ALL_EXCEPT);
  printf("after raising underflow exception:\n");
  print_flags(excepts);

  /* restore original state of flags */
  fesetexceptflag(&flags, FE_ALL_EXCEPT);
  printf("after fesetexceptflag:\n");
  excepts = fetestexcept(FE_ALL_EXCEPT);
  print_flags(excepts);
}

void print_flags(int flags)
{
  if (flags & FE_INEXACT)
    printf(" inexact result occurred\n");
  if (flags & FE_UNDERFLOW)
    printf(" underflow occurred\n");
  if (flags & FE_OVERFLOW)
    printf(" overflow occurred\n");
  if (flags & FE_DIVBYZERO)
    printf(" division by zero occurred\n");
  if (flags & FE_INVALID)
    printf(" invalid operation occurred\n");

  if (!(flags & FE_ALL_EXCEPT))
    printf(" no exceptions are set\n");
}
/*************************************************************/
```

If you run this program, it generates the following output:

```
$ cc fe_flags.c -lm
$ ./a.out
at start:
 no exceptions are set
after raising divide-by-zero exception:
 division by zero occurred
after raising inexact exception:
 inexact result occurred
 division by zero occurred
after clearing exceptions:
 no exceptions are set
after raising underflow exception:
 inexact result occurred
 underflow occurred
after fesetexceptflag:
 no exceptions are set
```

Manipulating the Exception Trap Enable Bits: fegettrapenable and fesettrapenable

The `fegettrapenable` function retrieves the current setting of the exception trap enable bits. The `fesettrapenable` function sets the trap enable bits.

The C declarations for these functions are as follows:

```
int fegettrapenable(void);

void fesettrapenable(int excepts);
```

The `fegettrapenable` function returns the bitwise OR of the exception macros corresponding to the currently set exception trap enable bits.

The `fesettrapenable` function sets the exception trap enable bits indicated by the argument *excepts*, which is a bitwise OR of the exception macros corresponding to the desired exception trap enable bits. The function also clears the trap enable bits for any exceptions not indicated by the argument *excepts*.

The following program calls these functions.

Sample Program: fe_traps.c

```
/*************************************************************/
#include <stdio.h>
#include <fenv.h>
#include <signal.h>

int traps;
fexcept_t flags;

/* trap handler for floating-point exceptions */
void handle_sigfpe(int sig)
{
  void print_flags(int);
  void print_traps(int);

  printf("Raised signal %d -- floating point error\n", sig);
  print_traps(traps);
  print_flags(flags);
  exit(-1);
}

int main(void)
{
  void print_flags(int);
  void print_traps(int);
  struct sigaction act;

  /* establish the trap handler */
  act.sa_handler = &handle_sigfpe;
  sigaction(SIGFPE, &act, NULL);

  /* retrieve initial settings of flags and traps */
  traps = fegettrapenable();
  fegetexceptflag(&flags, FE_ALL_EXCEPT);
  printf("at start:\n");
  print_traps(traps);
  print_flags(flags);

  /* set a trap for the divide by zero exception */
  fesettrapenable(FE_DIVBYZERO);
  traps = fegettrapenable();
  printf("after fesettrapenable(FE_DIVBYZERO):\n");
  print_traps(traps);

  /* raise divide by zero exception with trap enabled;
     trap handler is called */
  printf("raising division by zero exception now\n");
  feraiseexcept(FE_DIVBYZERO);
}                                              /* continued */
/*************************************************************/
```

Sample Program: fe_traps.c (cont.)

```
/*************************************************************/
void print_flags(int flags)
{
  if (flags & FE_INEXACT)
    printf(" inexact result flag set\n");
  if (flags & FE_UNDERFLOW)
    printf(" underflow flag set\n");
  if (flags & FE_OVERFLOW)
    printf(" overflow flag set\n");
  if (flags & FE_DIVBYZERO)
    printf(" division by zero flag set\n");
  if (flags & FE_INVALID)
    printf(" invalid operation flag set\n");

  if (!(flags & FE_ALL_EXCEPT))
    printf(" no exception flags are set\n");
}

void print_traps(int traps)
{
  if (traps & FE_INEXACT)
    printf(" inexact result trap enabled\n");
  if (traps & FE_UNDERFLOW)
    printf(" underflow trap enabled\n");
  if (traps & FE_OVERFLOW)
    printf(" overflow trap enabled\n");
  if (traps & FE_DIVBYZERO)
    printf(" division by zero trap enabled\n");
  if (traps & FE_INVALID)
    printf(" invalid operation trap enabled\n");

  if (!(traps & FE_ALL_EXCEPT))
    printf(" no traps are enabled\n");
}
/*************************************************************/
```

If you run this program, it produces the following output:

```
$ cc fe_traps.c -lm
$ ./a.out
at start:
 no traps are enabled
 no exception flags are set
after fesettrapenable(FE_DIVBYZERO):
 division by zero trap enabled
raising division by zero exception now
Raised signal 8 -- floating point error
 division by zero trap enabled
 no exception flags are set
```

Manipulating the Floating-Point Environment: fegetenv, fesetenv, feupdateenv, feholdexcept

The *fenv*(5) suite includes a group of functions that allow you to manage the floating-point environment as a whole. The declarations for these functions are as follows.

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

All these functions take as argument a value representing the floating-point environment. On HP-UX systems, this value is that of the floating-point status register.

The `fegetenv` function stores the current floating-point environment in the object pointed to by *envp*.

The `fesetenv` function establishes the floating-point environment represented by the object pointed to by *envp*. The argument *envp* must point to an object set by a call to `fegetenv` or `feholdexcept`, or equal the macro `FE_DFL_ENV`.

The `feupdateenv` function saves the current exceptions in its automatic storage, installs the floating-point environment represented through *envp*, and then raises the saved exceptions. The argument *envp* must point to an object set by a call to `fegetenv` or `feholdexcept`, or equal the macro `FE_DFL_ENV`.

The `feholdexcept` function function saves the current floating-point environment in the object pointed to by *envp*, clears the exception flags, and disables all traps.

The functions can be used together to save and restore the floating-point environment at different parts of your program and in different ways, so that you can control whether selected exceptions are hidden from calling routines. For example, you can call `feholdexcept` to store the floating-point environment temporarily and start afresh with no flags or traps set. Later on, you can use either `fesetenv` or `feupdateenv` to restore the saved environment, or save and update it. A call to `fesetenv` does not raise any saved exceptions, however, whereas a call to `feupdateenv` does raise the exceptions.

The following program shows the use of all these functions except `feupdateenv`.

Sample Program: fe_env.c

```
/*************************************************************/
#include <stdio.h>
#include <fenv.h>

int main(void)
{
  double x, y, z;
  fenv_t env, holdenv;
  int set_excepts;
  fexcept_t flags;
  char c = ' ';
  void print_flags(int);

  fegetenv(&env);
  printf("at start, env is %08x\n", env);

  do {
    printf("\nEnter x and y: ");
    scanf("%lf %lf", &x, &y);
    printf("x and y are %g and %g\n", x, y);

    z = x/y;       /* perform calculations */

    set_excepts = fetestexcept(FE_ALL_EXCEPT);
    print_flags(set_excepts);
    printf("result is %g\n", z);
    printf("again? (y or n) ");
    fflush(stdin);
    scanf("%c", &c);
  } while (c != 'n');

  printf("enabling trap for inexact\n");
  fesettrapenable(FE_INEXACT);

  fegetenv(&env);
  printf("after calculations and enabling inexact trap, \
env is %08x\n", env);

  printf("saving environment\n");
                                                  /* continued */
/*************************************************************/
```

Sample Program: fe_env.c (cont.)

```
/*************************************************************/
  feholdexcept(&holdenv);

  do {
    printf("\nEnter x and y: ");
    scanf("%lf %lf", &x, &y);
    printf("x and y are %g and %g\n", x, y);

    z = x/y;       /* perform calculations */

    set_excepts = fetestexcept(FE_ALL_EXCEPT);
    print_flags(set_excepts);
    printf("result is %g\n", z);
    printf("again? (y or n) ");
    fflush(stdin);
    scanf("%c", &c);
  } while (c != 'n');

  fegetenv(&env);
  printf("after more calculations, env is %08x\n", env);

  printf("resetting env to saved version\n");
  fesetenv(&holdenv);
  set_excepts = fetestexcept(FE_ALL_EXCEPT);
  print_flags(set_excepts);

  fegetenv(&env);
  printf("env is %08x\n", env);

  feclearexcept(FE_UNDERFLOW | FE_INEXACT);
  fegetenv(&env);
  printf("after feclearexcept(FE_UNDERFLOW | FE_INEXACT), \
env is %08x\n", env);
  set_excepts = fetestexcept(FE_ALL_EXCEPT);
  print_flags(set_excepts);

  fesetenv(FE_DFL_ENV);
  fegetenv(&env);
  printf("after fesetenv(FE_DFL_ENV), env is %08x\n", env);
}

void print_flags(int flags)
{
  if (flags & FE_INEXACT)
    printf(" inexact result occurred\n");
  if (flags & FE_UNDERFLOW)
    printf(" underflow occurred\n");
  if (flags & FE_OVERFLOW)
    printf(" overflow occurred\n");
  if (flags & FE_DIVBYZERO)
    printf(" division by zero occurred\n");
  if (flags & FE_INVALID)
    printf(" invalid operation occurred\n");

  if (!(flags & FE_ALL_EXCEPT))
    printf(" no exceptions are set\n");
}
/*************************************************************/
```

If you compile and run this program, the call to `fesetenv` restores the environment without raising the inexact exception, so the inexact trap is not taken even though it is set.

```
$ fe_env
at start, env is 000b0800
Enter x and y: 1.0e308 1.0e-308
x and y are 1e+308 and 1e-308
 inexact result occurred
 overflow occurred
result is inf
again? (y or n) n
setting trap for inexact
after calculations, env is 280b0801
saving environment
Enter x and y: 1.0e-308 1.0e308
x and y are 1e-308 and 1e+308
 inexact result occurred
 underflow occurred
result is 0
again? (y or n) n
after more calculations, env is 1c0b0800
resetting env to saved version
 inexact result occurred
 overflow occurred
env is 2c0b0801
after feclearexcept(FE_UNDERFLOW | FE_INEXACT), env is 240b0801
 overflow occurred
after fesetenv(FE_DFL_ENV), env is 040b0800
```

The following program illustrates the use of the `feupdateenv` function.This program computes a sum of squares. Some of the intermediate computations may underflow, but only if the final result underflows or overflows do we want to raise an exception. Therefore, after setting traps for underflow and overflow exceptions, the program calls `feholdexcept` to save the previously accumulated exceptions. After performing the intermediate computations, it clears any underflow or inexact exceptions, then calls `feupdateenv` to merge the current exceptions with the previously accumulated ones.

Sample Program: fe_update.c

```
/*************************************************************/
#include <stdio.h>
#include <values.h>
#include <fenv.h>

#define ARRLEN 11

static double argarr[ARRLEN] = { -1.73205, -0.57735,
                                  0.0174551, 0.57735, 1.0, 1.73205,
                                  2.0, -1.22465e-244,
                                  -2.44929e-207, -1.0 };

int main(void)
{
  double x, y, z;
  fenv_t env, holdenv;
  int set_excepts, i;
  fexcept_t flags;
  char c = ' ';
  void print_flags(int);

  fegetenv(&env);
  printf("at start, env is %08x\n", env);

  do {
    printf("\nEnter x and y: ");
    scanf("%lf %lf", &x, &y);
    printf("x and y are %g and %g\n", x, y);

    z = x/y;       /* perform calculations */

    set_excepts = fetestexcept(FE_ALL_EXCEPT);
    print_flags(set_excepts);
    printf("result is %g\n", z);
    printf("again? (y or n) ");
    fflush(stdin);
    scanf("%c", &c);
  } while (c != 'n');

  printf("setting traps for overflow and underflow\n");
  fesettrapenable(FE_OVERFLOW | FE_UNDERFLOW);

  fegetenv(&env);
  printf("after calculations, env is %08x\n", env);

  printf("saving environment\n");
                                                    /* continued */
/*************************************************************/
```

Sample Program: fe_update.c (cont.)

```
/*************************************************************/
  feholdexcept(&holdenv);

  /* accumulate sum of squares */
  for (i = 0; i < ARRLEN; i++) {
        z = argarr[i] * argarr[i];
        y +=z;
  }

  set_excepts = fetestexcept(FE_ALL_EXCEPT);
  print_flags(set_excepts);
  printf("sum of squares is %g\n", y);

  fegetenv(&env);
  printf("after more calculations, env is %08x\n", env);

  if (y > MINDOUBLE) {
      printf("clearing underflow & inexact exceptions\n");
      feclearexcept(FE_UNDERFLOW | FE_INEXACT);
  }
  else {
      printf("clearing inexact exception\n");
      feclearexcept(FE_INEXACT);
  }

  fegetenv(&env);
  printf("after feclearexcept, env is %08x\n", env);

  printf("merging env with saved version\n");
  feupdateenv(&holdenv);
  fegetenv(&env);
  printf("after feupdateenv, env is %08x\n", env);

  set_excepts = fetestexcept(FE_ALL_EXCEPT);
  print_flags(set_excepts);
}

void print_flags(int flags)
{
  if (flags & FE_INEXACT)
    printf(" inexact result occurred\n");
  if (flags & FE_UNDERFLOW)
    printf(" underflow occurred\n");
  if (flags & FE_OVERFLOW)
    printf(" overflow occurred\n");
  if (flags & FE_DIVBYZERO)
    printf(" division by zero occurred\n");
  if (flags & FE_INVALID)
    printf(" invalid operation occurred\n");

  if (!(flags & FE_ALL_EXCEPT))
    printf(" no exceptions are set\n");
}
/*************************************************************/
```

If you compile and run this program, it produces results like the following:

```
$ cc fe_update.c -lm
$ ./a.out
at start, env is 000e0000

Enter x and y: 3 0
x and y are 3 and 0
 inexact result occurred
 division by zero occurred
result is inf
again? (y or n) n
setting traps for overflow and underflow
after calculations, env is 480e0006
saving environment
 inexact result occurred
 underflow occurred
sum of squares is 12.667
after more calculations, env is 180e0000
clearing underflow & inexact exceptions
after feclearexcept, env is 040e0000
merging env with saved version
after feupdateenv, env is 4c0e0006
 inexact result occurred
 division by zero occurred
```

Underflow Mode: fegetflushtozero and fesetflushtozero

The functions `fegetflushtozero` and `fesetflushtozero` allow the programmer to retrieve the current underflow mode or to change the way the system handles underflows by setting the D bit in the status register to either 1 or 0.

Flush-to-zero mode, also known as **fast underflow mode**, **sudden underflow mode**, or **fastmode**, is an alternative to IEEE-754-compliant underflow mode. On HP 9000 systems, a floating-point underflow involves a fault into the kernel, where the IEEE-754-specified conversion of the result into a denormalized value or zero is accomplished by software emulation. On many HP 9000 systems, flush-to-zero mode allows the hardware to simply substitute a zero for the result of an operation, with no fault occurring. (The zero has the sign of the result for which it is substituted.) This may be a significant performance optimization for applications that underflow frequently. For many operations, flush-to-zero mode also causes denormalized floating-point operands to be treated as if they were true zero operands.

NOTE

Flush-to-zero mode is supported on all HP 9000 systems except those with chip levels of PA7000 and PA7100LC. You can look up the chip level of your system in `/opt/langtools/lib/sched.models`. See "Determining Your System's Architecture Type" on page 26 for more information.

The C declarations for these functions are as follows:

```
int fegetflushtozero(void);
void fesetflushtozero(int value);
```

The `fegetflushtozero` function returns the current flush-to-zero mode setting: a result of 1 means that flush-to-zero mode is set, a result of 0 means that the default IEEE-754-compliant underflow mode is set. On systems that do not support flush-to-zero mode, this function always returns 0.

On systems that support flush-to-zero mode, the `fesetflushtozero` function sets flush-to-zero mode to *value*, which must be either 1 (flush-to-zero mode) or 0 (IEEE-754-compliant underflow mode). On systems that do not support flush-to-zero mode, this function has no effect.

On systems that support flush-to-zero mode, the default setting is IEEE-754-compliant underflow mode (0).

The following program calls `fegetflushtozero` and `fesetflushtozero` to retrieve and set the underflow mode.

Sample Program: fe_flush.c

```
/*************************************************************/
#include <stdio.h>
#include <fenv.h>

typedef union {
  double y;
  struct {
     unsigned int ym, yl;
  } i;
} DBL_INT;

int main(void)
{
  DBL_INT dix, diy, diz;
  int fm, fm_saved;

  fm_saved = fegetflushtozero();
  printf("underflow mode is %d\n", fm_saved);

  dix.y = -4.94066e-324;
  printf("denormalized value is %g [%08x%08x]\n", dix.y,
         dix.i.ym, dix.i.yl);

  fesetflushtozero(1);
  fm = fegetflushtozero();
  printf("after fesetflushtozero(1), mode is %d\n", fm);
  printf("denormalized value is %g [%08x%08x]\n", dix.y,
         dix.i.ym, dix.i.yl);

  fesetflushtozero(fm_saved);
  fm = fegetflushtozero();
  printf("after fesetflushtozero(%d), mode is %d\n",
         fm_saved, fm);
  printf("denormalized value is %g [%08x%08x]\n", dix.y,
         dix.i.ym, dix.i.yl);
}
/*************************************************************/
```

If you run this program on a system that supports flush-to-zero mode, it produces the following output:

```
$ cc fe_flush.c -lm
$ ./a.out
underflow mode is 0
value is -4.94066e-324 [8000000000000001]
after fesetflushtozero(1), mode is 1
value is 0 [8000000000000001]
after fesetflushtozero(0), mode is 0
value is -4.94066e-324 [8000000000000001]
```

Command-Line Mode Control: The +FP Compiler Option

The compiler and linker option +FP allows you to specify what traps to enable for your program and can also enable or disable flush-to-zero mode. This option is available with the HP Fortran, C, and Pascal compilers. It has the following syntax:

+FP*flags*

where *flags* is a series of uppercase or lowercase letters from the set [VvZzOoUuIiDd] with no spaces, tabs, or other characters between them. If the uppercase letter is selected, that behavior is enabled. If the lowercase letter is selected or if the letter is not present in the flags, the behavior is disabled. By default, all traps are disabled.

Table 5-3 describes the behavior specified by each argument.

For example, the following command line sets traps for overflow, divide by zero, and invalid operations, and enables fast underflow mode:

```
f77 +FPOZVD program_name.f
```

The linker (ld) also accepts the +FP option. If you specify this option to a separately invoked ld command, the option is effective only if you link in one of the supported startup files (/opt/langtools/lib/*crt0.o).

The +FP option affects the various floating-point mode settings at program startup. Subsequent calls to fesettrapenable, fesetflushtozero, or fpsetenv may override some or all of the values set by +FP.

NOTE

If you use Fortran, you may also use the +fp_exception (Fortran 90) or +T (HP FORTRAN/9000) option to set traps at compile time. (If your HP FORTRAN/9000 program contains an ON statement, you must use this option.) These options enable traps for invalid operation, overflow, underflow, and division by zero exceptions. If you specify both +FP and +fp_exception (or +T), the +fp_exception (or +T) option is always processed after +FP. It does not turn off any bits set by +FP, however.

See “Using the +FP Compiler Option” on page 151 for more information on using the +FP option.

Table 5-3 +FP Option Arguments

Value	Behavior
V	Enable traps on invalid floating-point operations.
v	Disable traps on invalid floating-point operations.
Z	Enable traps on divide by zero.
z	Disable traps on divide by zero.
O	Enable traps on floating-point overflow.
o	Disable traps on floating-point overflow.
U	Enable traps on floating-point underflow.
u	Disable traps on floating-point underflow.
I	Enable traps on floating-point operations that produce inexact results.
i	Disable traps on floating-point operations that produce inexact results.
D	Enable flush-to-zero (fast underflow) mode for denormalized values. (Selecting this value enables flush-to-zero mode only if it is available on the processor that is used at run time.)
d	Disable flush-to-zero (fast underflow) mode for denormalized values.

6 Floating-Point Trap Handling

By default, trapping on floating-point exceptions is disabled on HP 9000 systems, in accordance with the IEEE standard. If you want your program to continue through floating-point exceptions without trapping, it will do so automatically.

Floating-point trap handling requires two main steps:

1. Setting the exception trap enable bits in the floating-point status register
2. Defining the action to be taken when the trap occurs

Both steps vary somewhat from language to language on HP-UX systems. In this section we suggest some methods of enabling and handling traps in Fortran and C.

In C, you can also use the exception flags in the floating-point status register to detect exceptions without taking a trap. See “Detecting Exceptions without Enabling Traps” on page 162 for details.

If your code handles integer arithmetic as well as floating-point arithmetic, you may encounter an integer exception. We discuss integer exceptions briefly in “Handling Integer Exceptions” on page 163.

See “Exception Conditions” on page 51 for information about IEEE exceptions. See “The PA-RISC Floating-Point Status Register” on page 126 for information about the floating-point status register, including the exception flags and the exception trap enable bits.

Enabling Traps

When you enable a trap without providing a trap handler (a mechanism for handling it), the trap causes a SIGFPE signal. The signal, in turn, causes your program to abort with an error message that is more or less informative, depending on the method you use to enable the trap. If you want your program to abort when it encounters a trap, then enabling the trap may be all you want to do. However, you may want to handle the trap gracefully; see "Handling Traps" on page 155 for information on trap handling.

HP 9000 systems provide several methods of enabling traps:

- The +FP compiler option (all compilers)
- The `fesettrapenable` function
- The `+fp_exception` (Fortran 90) and +T (HP FORTRAN/9000) compile-line options

We discuss these briefly in the following subsections.

Using the +FP Compiler Option

The +FP option, described in "Command-Line Mode Control: The +FP Compiler Option" on page 146, allows you to enable traps from the compiler command line. No change in your program is required.

The disadvantage of using this option is that you cannot know exactly where in your program the exception occurred. Moreover, unless you enable a trap for only one exception, you cannot know the type of exception that occurred. You merely get a core dump. The following Fortran program, for example, generates an overflow.

Sample Program: overflow.f

```
      PROGRAM OVERFLOW
      DOUBLE PRECISION X, Y, Z

      X = 1.79D308
      Y = 2.2D-308
      Z = X / Y
      PRINT 30, X, Y, Z
30    FORMAT (1PE11.4, ' divided by', 1PE11.4, ' = ', 1PE11.4)
      END
```

If you compile this program with traps disabled (the default), it produces the following output:

```
 1.7900+308 divided by 2.2000-308 = +INF
```

If you compile it with the +FP option, however, you get a stack trace and core dump. (The O flag of the +FP option enables traps for overflow exceptions.)

```
$ f90 +FPO overflow.f
overflow.f
   program OVERFLOW

11 Lines Compiled
$ ./a.out
( 0)  0xc1602c5c   traceback + 0x14  [/usr/lib/pa1.1/libcl.1]
( 1)  0xc0126500   _sigreturn  [/usr/lib/libc.1]
( 2)  0x000024b4   _start + 0x6c  [./a.out]
Floating exception (core dumped)
```

Using the fesettrapenable Function

The `fesettrapenable` function is part of the *fenv*(5) suite and is described in detail in “Exception Bits” on page 130. It is provided in the C math library.

The `fesettrapenable` routine can enable one or more traps in any combination.

Using `fesettrapenable` to enable traps has the same disadvantages as the +FP option. You get a core dump, and you cannot determine the exception type. Moreover, the use of these routines lacks the simplicity of the +FP option, because you must modify your code in order to use them.

NOTE

Do not use this function at an optimization level greater than 0. See “Run-Time Mode Control: The fenv(5) Suite” on page 125. (You may instead use the `FLOAT_TRAPS_ON` pragma to suppress optimization.)

Here is the program from “Using the +FP Compiler Option” on page 151, modified to enable a trap for the overflow exception using `fesettrapenable`. It does this by using an `!$HP$ ALIAS` or `$ALIAS` directive to tell the Fortran compiler that the function's arguments are passed by value.

Sample Program: overflow_trap.f

```
      PROGRAM OVERFLOW_TRAP
C F90:
!$HP$ ALIAS FESETTRAPENABLE = 'fesettrapenable' (%val)
C F77:
C $ALIAS FESETTRAPENABLE = 'fesettrapenable' (%val)
      PARAMETER (FE_OVERFLOW = Z'20000000')
      EXTERNAL FESETTRAPENABLE
      DOUBLE PRECISION X, Y, Z

      CALL FESETTRAPENABLE(FE_OVERFLOW)

      X = 1.79D308
      Y = 2.2D-308
      Z = X / Y
      WRITE(*,*) X, ' divided by', Y, ' = ', Z
      END
```

You do not need to specify any options when you compile this program, but you need to link in the C math library. The result is even less useful than in the preceding section:

```
$ f90 overflow_trap.f -lm
overflow_trap.f
   program OVERFLOW_TRAP

13 Lines Compiled
$ ./a.out
Floating exception (core dumped)
```

Using the +fp_exception or +T Compiler Option (Fortran only)

The `+fp_exception` option, available with the HP Fortran 90 compiler, enables traps for four IEEE exceptions: invalid operation, division by zero, overflow, and underflow. (Few programs need to trap for the inexact exception, which occurs often and conveys little information.) The HP FORTRAN/9000 equivalent is `+T`. For Fortran applications, these options have the simplicity of `+FP`. They also provides more information than either `+FP` or `fesettrapenable`. They tell what kind of exception occurred and the virtual address of the statement that triggered the exception. Moreover, they do not cause a core dump.

For example, if you use +fp_exception or +T to compile the program in "Using the +FP Compiler Option" on page 151, it produces a result like the following:

```
$ f77 +T overflow.f
overflow.f:
   MAIN overflow:
$ ./a.out
PROGRAM ABORTED :  IEEE overflow

PROCEDURE TRACEBACK:

( 0)  0x000024e8   _start + 0x88  [./a.out]
```

The effect is similar with +fp_exception:

```
$ f90 +fp_exception overflow.f
overflow.f
   program OVERFLOW

11 Lines Compiled
$ ./a.out
PROGRAM ABORTED :  IEEE overflow

PROCEDURE TRACEBACK:

( 0)  0x000024ec   _start + 0x74  [././a.out]
```

See "Command-Line Mode Control: The +FP Compiler Option" on page 146 for information about how +fp_exception and +T interact with +FP if you use them together. See the *f90*(1) man page and the *HP Fortran 90 Programmer's Reference* for details about using +fp_exception. See the *f77*(1) man page and the *HP FORTRAN/9000 Programmer's Guide* for details about using +T.

Handling Traps

Once you have enabled traps, either by a compiler option or by a call to a routine, you need a mechanism for handling them when they occur. It may be convenient simply to have your program abort (particularly if you enable traps with a method that does not cause a core dump). You may, however, prefer to establish an error-handling routine that generates a helpful error message and exits the program gracefully.

HP 9000 systems provide the following methods of handling traps:

- The `ON` statement (Fortran only)
- The *sigaction*(2) function (C only)

We discuss these briefly in the following subsections.

You may want to continue execution after handling a trap, but you should be very cautious about doing so. The most important reason, of course, is that if your program encounters an exception, the result of at least that portion of your calculations is likely to be useless.

Another reason occurs if you use the C *sigaction*(2) function for error handling. Floating-point exceptions are hardware exceptions, whereas an error-handling process (such as the C *sigaction*(2) function or the HP Fortran `ON` statement) occurs in software. A hardware exception usually causes an operation to be interrupted at a point when values are stored in registers but not yet stored in memory. If you use *sigaction*(2) to try to substitute a new value for the one that caused the error, chances are you will return to a point later in the instruction sequence than the point at which the error occurred, so some essential steps may be eliminated and the register values may be invalid. This particular problem does not occur with the HP Fortran `ON` statement; the software updates the registers appropriately.

Using the ON Statement (Fortran only)

The `ON` statement may be used to handle arithmetic exceptions in both HP Fortran 90 and HP FORTRAN/9000. In HP FORTRAN/9000 the statement must be used in conjunction with the `+T` compiler option. In HP Fortran 90 it may be used in conjunction with the `+fp_exception` option, but this is not required.

The ON statement allows you to specify a particular action to be taken when a particular exception arises. The action may be any of the following:

- ABORT (the default action). Specifying ABORT allows you to get the address where the error occurred, which may be useful in debugging.
- IGNORE (usually not a good idea).
- CALL *sub* (call a subroutine). If you call a subroutine to handle an IEEE exception, you must pass it one argument, which is of the same type as the type associated with the exception. If the subroutine returns, the program uses the assigned value of the argument as the result value of the operation that caused the handler to be invoked.

NOTE

A subroutine that handles an IEEE exception takes a different number of arguments from a subroutine that handles a math library error (as described in "Math Library Error Handling for Fortran" on page 104). See the *HP FORTRAN/9000 Programmer's Guide* for details. HP discourages the use of the ON statement to handle library errors in HP Fortran 90, but using it to handle arithmetic errors poses no problems.

For example, the following program calls the subroutine HANDLE_OFL to handle a DOUBLE PRECISION overflow. The subroutine prints a message describing the error, prints the biased value passed to the subroutine, then the correct value, and exits.

Sample Program: overflow_on.f

```
      PROGRAM OVERFLOW_ON
      DOUBLE PRECISION X, Y, Z

      ON DOUBLE PRECISION OVERFLOW CALL HANDLE_OFL

      X = 1.79D308
      Y = 2.2D-308
      Z = X / Y
      WRITE(*,*) X, ' divided by', Y, ' = ', Z
      END

      SUBROUTINE HANDLE_OFL(A)
      DOUBLE PRECISION A, T
      INTEGER I
      WRITE(*,*) 'overflow occurred'
      WRITE(*,*) 'argument to HANDLE_OFL is ', A
C  The result is 2**1536 too small
      T = LOG10(A) + 1536 * LOG10(2D0)
      I = T                    ! Get exponent
      T = T - I
      WRITE(*,*) 'the correct answer is', 10**T,
     x           ' times 10 to the power', I
      STOP
      END
```

See the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* for a full explanation of the bias of 1536; when traps are enabled, the IEEE standard requires that a biased result be returned.

If you compile and run this program using HP Fortran 90 and HP FORTRAN/9000 respectively, it produces the following result:

```
$ f90 overflow_on.f
overflow_on.f
   program OVERFLOW_ON
   external subroutine HANDLE_OFL

26 Lines Compiled
$ ./a.out
 overflow occurred
 argument to HANDLE_OFL is  3.375646885228544E+153
 the correct answer is 8.13636363636275  times 10 to the power 615

$ f77 +T overflow_on.f
overflow_on.f:
   MAIN overflow_on:
   handle_ofl:
$ ./a.out
 overflow occurred
 argument to HANDLE_OFL is  3.375646885228544+153
 the correct answer is 8.13636363636275 times 10 to the power 615
```

One situation where it is useful to assign a value to the trap handler argument and continue program execution is that of an underflow exception (described in “Underflow Conditions” on page 55). Substituting a value of 0 for the result of an operation that underflows may be exactly

what you want to do. In fact, the system may perform this substitution for you; the IEEE standard specifies that 0 may be the result of an operation that underflows, and on HP 9000 systems it often is (when the result of the operation is less than the smallest denormalized value). If you want to guarantee a result of 0, you can call a handler as follows:

Sample Program: underflow_on.f

```
      PROGRAM UNDERFLOW_ON
      DOUBLE PRECISION X, Y, Z

      ON DOUBLE PRECISION UNDERFLOW CALL HANDLE_UFL

      X = 1.0D0
      Y = 1.79D308
      Z = X / Y
      PRINT 30, X, Y, Z
30    FORMAT (1PE11.4, ' divided by', 1PE11.4, ' = ', 1PE11.4)
      END

      SUBROUTINE HANDLE_UFL(A)
      DOUBLE PRECISION A

      WRITE(*,*) 'underflow occurred'
      A = 0.0
      RETURN
      END
```

If you compile and run this program, it produces the following result:

```
$ f90 underflow_on.f
underflow_on.f
   program UNDERFLOW_ON
   external subroutine HANDLE_UFL

21 Lines Compiled
$ ./a.out
 underflow occurred
 1.0000E+00 divided by 1.7900+308 =  0.0000E+00

$ f77 +T underflow_on.f
underflow_on.f:
   MAIN underflow_on:
   handle_ufl:
$ ./a.out
 underflow occurred
 1.0000E+00 divided by 1.7900+308 =   .0000E+00
```

If your system supports fast underflow mode, you can use it both to guarantee a result of 0 for all underflows and to avoid the overhead of incurring a trap. You can enable fast underflow mode with either the +FP option (see “Command-Line Mode Control: The +FP Compiler Option” on page 146) or the `fesetflushtozero` routine (see “Underflow Mode: fegetflushtozero and fesetflushtozero” on page 143).

The `ON` statement is documented fully in the *HP Fortran 90 Programmer's Reference* and the *HP FORTRAN/9000 Programmer's Guide*.

Using the sigaction(2) Function (C only)

For C programs, the standard method of handling errors is to use the *sigaction*(2) function. The function establishes the address of a signal-handling function that is called whenever the specified HP-UX signal is raised. The major problem with this method is that there is only one HP-UX signal, `SIGFPE`, for all IEEE and integer exceptions. When `SIGFPE` is raised, the only way to find out which exception generated the signal is to examine the floating-point exception registers, which can be obtained from the `sigcontext` structure that is an argument to the *sigaction*(2) function. (Or you can enable a trap for only one exception.)

The following C program uses `fesettrapenable` to enable a trap for the overflow exception, and calls *sigaction*(2) to specify that the function `handle_sigfpe` is to be called when `SIGFPE` is raised.

Sample Program: overflow_sig.c

```
/*************************************************************/
#include <stdio.h>
#include <math.h>
#include <fenv.h>
#include <signal.h>

int traps;
fexcept_t flags;

/* signal handler for floating-point exceptions */
void handle_sigfpe(int signo, siginfo_t *siginfo,
                   ucontext_t *ucontextptr)
{
    void print_flags(int);
    void print_traps(int);

    printf("Raised signal %d -- floating point error\n", signo);
    printf("code is %d\n", siginfo->si_code);
    printf("fr0L is %08x\n", ucontextptr->uc_mcontext.ss_frstat);
    print_traps(traps);
    fegetexceptflag(&flags, FE_ALL_EXCEPT);
    print_flags(flags);
    exit(-1);
}

int main(void)
{
    void print_flags(int);
    void print_traps(int);
    double x, y, z;
    struct sigaction act;

    act.sa_handler = &handle_sigfpe;
    act.sa_flags = SA_SIGINFO;

    /* establish the signal handler */
    sigaction(SIGFPE, &act, NULL);

    fesettrapenable(FE_OVERFLOW);
    traps = fegettrapenable();
    print_traps(traps);

    x = 1.79e308;
    y = 2.2e-308;
    z = x / y;    /* divide very big by very small */
    printf("%g / %g = %g\n", x, y, z);
}                                                  /* continued */
/*************************************************************/
```

Sample Program: overflow_sig.c (Cont.)

```
/*************************************************************/
void print_flags(int flags)
{
    if (flags & FE_INEXACT)
      printf(" inexact result flag set\n");
    if (flags & FE_UNDERFLOW)
      printf(" underflow flag set\n");
    if (flags & FE_OVERFLOW)
      printf(" overflow flag set\n");
    if (flags & FE_DIVBYZERO)
      printf(" division by zero flag set\n");
    if (flags & FE_INVALID)
      printf(" invalid operation flag set\n");

    if (!(flags & FE_ALL_EXCEPT))
        printf(" no exception flags are set\n");
}

void print_traps(int traps)
{
    if (traps & FE_INEXACT)
      printf(" inexact result trap set\n");
    if (traps & FE_UNDERFLOW)
      printf(" underflow trap set\n");
    if (traps & FE_OVERFLOW)
      printf(" overflow trap set\n");
    if (traps & FE_DIVBYZERO)
      printf(" division by zero trap set\n");
    if (traps & FE_INVALID)
      printf(" invalid operation trap set\n");

    if (!(traps & FE_ALL_EXCEPT))
        printf(" no trap enables are set\n");
}
/*************************************************************/
```

If you compile and run this program, it produces a result like the following:

```
$ cc overflow_sig.c -lm
$ ./a.out
 overflow trap set
Raised signal 8 -- floating point error
code is 14
fr0L is 08081844
 overflow trap set
 inexact result flag set
```

The code 14 signifies an assist exception trap (see *signal*(5)), which indicates a floating-point exception.

The main advantage of using a signal handler is that it eliminates the core dump that you get without it if you compile with +FP or use `fesettrapenable`.

Detecting Exceptions without Enabling Traps

If you do not enable traps for floating-point exceptions, you can still determine whether the exceptions have occurred. When an exception occurs and the corresponding exception trap enable bit is not set, the system sets the corresponding exception flag. The exception flags are cumulative; once a flag is set, it remains set for the duration of the program unless you clear it. (This is why the exception flags are also called sticky bits.)

At critical points in your program, you can call the `fetestexcept` function (described in "Manipulating the Exception Flags: fegetexceptflag, fesetexceptflag, fetestexcept, feraiseexcept, feclearexcept" on page 131) to determine whether an exception has occurred. You can then respond to the exception as you wish. The program example in that section illustrates the use of this function.

NOTE

Be careful if you use these functions at higher optimization levels (2 and above). See "Run-Time Mode Control: The fenv(5) Suite" on page 125.

Handling Integer Exceptions

In Fortran and C there are two kinds of integer exceptions, division by zero and overflow. Default behavior for these exceptions differs.

Handling Integer Division by Zero

Trapping on integer division by zero is enabled by default for both Fortran and C. The trap generates a `SIGFPE` error. This error may be confusing, because the error is really an integer error, and you cannot disable or enable it by manipulating the floating-point status register (for example, by using `+FP` or `fesettrapenable`).

If for some reason you wish to disable the trap, you can do so in Fortran by using an `ON INTEGER DIV 0 IGNORE` statement. You cannot disable the trap in C.

If you want to establish a handler for an integer division by zero, you can do so using either of the mechanisms described in “Handling Traps” on page 155: the C *sigaction*(2) function or the Fortran `ON` statement.

Handling Integer Overflow

Trapping on integer overflow is disabled by default for Fortran and C; an integer overflow does not generate a `SIGFPE` error. Detecting integer overflows requires not only that the trap be enabled but also that the compiler insert special code in the executable file to check for overflows.

To enable integer overflow checking for Fortran, use a `!$HP$ CHECK_OVERFLOW ON` directive (in HP Fortran/9000, use `$CHECK_OVERFLOW INTEGER_4` or `INTEGER_2`) to obtain the overflow checking code, and use an `ON INTEGER OVERFLOW` statement to handle the trap. (The `!$HP$ CHECK_OVERFLOW` directive does not enable checking for operations in libraries. Using the exponentiation operator involves a library call in HP Fortran, so it is not possible to enable integer overflow checking for exponentiation operations.)

There is no way to enable integer overflow checking in C. HP C provides no mechanism to insert overflow checking code into your executable, because the C language does not define integer overflow as an error.

7 Performance Tuning

Performance tuning is the process of refining a program to make it run faster. Many of the techniques described in this chapter are general techniques that work for any program running on any system. Other techniques are specific to floating-point applications running on HP 9000

systems. We do not discuss input/output efficiency, which often has the most dramatic impact on a program's execution speed. Keep in mind that some performance tuning techniques carry a price—they may make the program less portable or less easy to understand and maintain. You should weigh these pros and cons before making major changes to your program.

Identifying and Removing Performance Bottlenecks

Computers are so fast that the only way to change the performance of your program noticeably is to change something that happens many, many times. For this reason, the first step in tuning an application for increased performance is to identify the **performance bottlenecks**, those sections of your code that collectively require the most execution time. By concentrating on these areas, you can obtain the greatest performance improvement for your effort.

Some programs spend a high percentage of execution time in small loops, while seemingly long pieces of sequential code actually account for a minor fraction of the total execution time. In such programs, the performance bottlenecks are likely to be small subroutines that are called frequently and small loops that execute over many iterations.

With its compilers HP provides the performance analysis tool Puma, which can help you locate bottlenecks in programs. Puma takes samples of the program counter, the call/return stack, and other performance statistics of executing programs, saving the results in a data file. You can then use Puma to display the data file in a variety of graphical formats. See the *puma*(1) man page and the *HP PAK Performance Analysis Tools User's Guide* for more information.

HP-UX systems also support the two UNIX tools *prof*(1) and *gprof*(1). Both of these tools tell you what percentage of the total running time of your program is spent in each routine called. See the *prof*(1) and *gprof*(1) man pages for information about these tools.

A more general system performance monitoring and diagnostic tool is the HP GlancePlus product, which provides immediate performance information about your computer system. It lets you easily examine system activities, identify and resolve performance bottlenecks, and tune your system for more efficient operation.

After you have identified the bottlenecks in your program, the next step is to figure out why the bottlenecks exist and, if possible, to remove them. The following sections apply to all code, but the emphasis is on

floating-point-intensive code that is running unacceptably slowly. There are several reasons why a piece of floating-point-intensive code might consume a lot of execution time:

- The code generated by the compiler is inefficient.
- The program does not link in the fastest version of the math library.
- The program links in shared libraries, which are slower than archive libraries.
- The data being processed involves denormalized operands or underflowing operations.
- The data being processed contains mixed-precision expressions.
- The code contains highly iterative loops (for example, vector and/or matrix operations). Or the code contains loops that perform vector and/or matrix operations, but the loops are not being vectorized.
- The data is not optimally aligned in memory.
- The program causes adverse cache aliasing effects.
- The code contains many static variables.
- The code performs quad-precision computations.

The following sections discuss each of these problems individually.

Inefficient Code

The HP-UX compilers are highly optimizing and generally produce extremely efficient code. However, you can control the degree of efficiency and the types of optimizations with compiler options and directives. Particularly important from a performance standpoint are the compiler options that do the following:

- Optimize your program
- Specify the architecture type, causing the compiler to generate code for a specific type of machine
- Cause the compiler to emit debugger information
- Cause the compiler to produce position-independent code, which generally runs more slowly than absolute code
- Enable performance-based optimization (PBO)
- (Fortran only) Cause the compiler to make all local variables static and to initialize all uninitialized static data to zero

The following sections describe each of these options. Many of the options are available both as command-line options and as directives or pragmas that you can place in your source code. For more information and specific syntax, refer to the appropriate HP language reference manual.

If the compiler generates inefficient code even when you use the appropriate options, you may choose to write parts of your program in assembly language. “Writing Routines in Assembly Language” on page 176 describes the advantages and disadvantages of this choice.

Optimizing Your Program

For a thorough discussion of optimization on HP 9000 systems, see the *HP PA-RISC Compiler Optimization Technology White Paper*. See the appropriate HP language manual for additional information.

The most important compiler option affecting efficiency is the optimization option, `+O`, which allows you to optimize your program in several different ways:

Levels of optimization Optimization levels, numbered from 0 to 4, allow you to select a broad category of optimizations, from minimal optimization to full optimization.

Types of optimization Optimization types allow you to select groups of optimizations that fall into a particular category. For example, `+Osize` suppresses optimizations that significantly increase code size. If you specify an optimization type, you must also specify an optimization level.

Specific optimizations Specific optimizations allow you to turn on or off particular optimizations that may be appropriate or inappropriate for your program. For example, `+Opipeline` (the default at optimization levels 2, 3, and 4) enables software pipelining. If you specify a specific optimization, you must also specify an optimization level.

In general, the higher the optimization level, the more efficient the code. In performing optimizations, the compiler often rearranges code and makes assumptions about the way variables will be used in other modules. There is some risk, therefore, in choosing a high optimization level, since the compiler may make some invalid assumptions that can cause code to run more slowly. This is particularly true if your code makes frequent use of pointers. It is always a good idea to compile a program at different optimization levels and compare the results to make sure that the optimizations are not affecting either the performance or the results. See “Compiler Behavior and Compiler Version” on page 76 and “Compiler Options” on page 77 for information about how compiler optimizations can affect program results.

The following specific optimizations are particularly relevant to floating-point programs. Most of them are available at optimization levels 2, 3, and 4.

`+O[no]dataprefetch`

`+Odataprefetch`, which has an effect only when you use the `+DA2.0` option, inserts instructions within innermost loops to fetch data from memory into the data cache ahead of time so that it is already there when it is needed. Data prefetch instructions are inserted only for data structures referenced within innermost loops using simple loop varying addresses (that is, in a simple linear sweep across large amounts of memory). It is useful for applications that have high data cache miss overhead; that is, it improves the performance of operations on arrays that are so large they exceed the size of the cache.

As a general rule of thumb, using `+Odataprefetch` will probably help performance if your application contains numerous references to arrays, and if the sum of the sizes of all the arrays in your program totals more than a megabyte. It can also help if your application contains only a single pass through an extremely large array (tens of megabytes in size). However, if your program contains very frequent references to *small* arrays, `+Odataprefetch` can actually impair performance. Therefore, the only way to find out for sure whether this option will help your program is to try it.

The `+Odataprefetch` option is effective with both vectorized and unvectorized loops. In fact, if your PA2.0 application uses very large arrays, you may gain considerable performance benefit from using

	`+Odataprefetch` in conjunction with `+Ovectorize`. The math library contains special prefetching versions of the vector routines, which are called if you specify both options.
`+O[no]fltacc`	`+Ofltacc`, which is the default at levels 2, 3, and 4, disables optimizations that are algebraically correct but that may result in numerical differences. (Usually these differences are insignificant.) To enable these optimizations, use `+Onofltacc`. On PA2.0 systems at level 2 and higher, if you specify neither `+Ofltacc` nor `+Onofltacc`, or if you specify `+Onofltacc`, the compiler generates FMA (fused multiply-add) instructions (see "Architecture Type of Run-Time System" on page 78 for details). Specify `+Ofltacc` to suppress the generation of these instructions. The `+Onofltacc` option is invoked by default when you specify the optimization type `+Oaggressive`; use `+Oaggressive +Ofltacc` if you want aggressive optimization without sacrificing floating-point accuracy.
`+O[no]inline`, `+Oinline_budget=`*n*	`+Oinline`, which is available at levels 3 and 4 and is the default at those levels, enables inlining of function calls. Inlining can improve performance significantly if your application makes many math library calls. It is especially effective on PA2.0 systems. The `+Oinline_budget` option, also available at levels 3 and 4, can be

	used to specify how aggressively you want the compiler to pursue inlining opportunities. The default value of *n* is 100.
`+O[no]libcalls`	`+Olibcalls`, which is available at all optimization levels (0 through 4), invokes millicode versions of several frequently called math library functions. It also inlines the double-precision versions of the `sqrt` and `fabs` C functions. This option is invoked by default when you specify the optimization type `+Oaggressive`; use `+Oaggressive +Onolibcalls` if you want aggressive optimization without using millicode routines. Do not use this option on a C program that depends on the setting of `errno` by math library functions. See "Millicode Versions of Math Library Functions" on page 112 for details.
`+O[no]moveflops`	`+Omoveflops`, which is the default at levels 2, 3, and 4, moves conditional floating-point instructions out of loops. This option may alter floating-point exception behavior. Use `+Onomoveflops` if you depend on floating-point exception behavior and you do not want this behavior to be altered by the relocation of floating-point instructions.
`+O[no]vectorize`	`+Ovectorize`, available with the Fortran and C compilers only, replaces eligible loops with calls to vector routines in the math library. The `+Ovectorize` option is invoked by default when you specify the optimization type `+Oaggressive`;

use `+Oaggressive +Onovectorize` if you want aggressive optimization without vector calls.

Any files that were compiled with `+Ovectorize` must also be linked with `+Ovectorize` (this happens automatically when the compiler invokes the linker).

This option can be used at optimization levels 3 and 4. The default is `+Onovectorize`. This option is valid only when you compile for PA1.1 and PA2.0 systems.

If your PA2.0 application uses very large arrays, you may gain considerable performance benefit from using `+Odataprefetch` in conjunction with `+Ovectorize`. The math library contains special prefetching versions of the vector routines, which are called if you specify both options.

Specifying the Architecture Type

All HP 9000 compilers support the `+DA` option, which specifies a particular target architecture type, either PA-RISC 1.1 or PA-RISC 2.0. Use of this option causes the compiler to produce architecture-specific instructions and calls to special architecture-specific run-time libraries.

Specifying the architecture type of the systems on which your code will run will probably improve the performance of your code if it makes substantial use of floating-point arithmetic or math library calls. See “Selecting Different Versions of the Math Libraries” on page 27, “Architecture Type of Run-Time System” on page 78, and “BLAS Library Versions” on page 178 for more information.

Use of the +DA2.0 option to generate PA2.0 code will improve the performance of your application even more if the source provides opportunities for the compiler to generate FMA (fused multiply-add) instructions (see "Architecture Type of Run-Time System" on page 78 for details). For example, if two statements like

```
c = a * b
```

and

```
e = c - d
```

are separated by intervening statements in your program, you may want to place them one right after the other or to combine them into

```
e = a * b - d
```

This kind of rearrangement will be most effective if done within loops.

The +DS option also has a significant effect on performance, because it specifies an architecture-specific instruction scheduler. If your code must be portable across all HP 9000 architectures, you must compile with +DA1.1, but you may compile with either +DS1.1 or +DS2.0. Use +DS2.0 if you want to achieve the best possible performance on PA2.0 systems. See the appropriate HP language reference manual for more information about this option.

Including Debugging Information

All HP 9000 compilers allow you to include debugging information in the object file at optimization levels 0, 1, and 2. Debugging information increases the size of the object code. The debugging option is extremely useful during program development, but for the final product you should compile without it.

Producing Position-Independent Code

By default, compilers produce absolute code for HP 9000 systems. You can produce position-independent code (PIC) for use in building shared libraries. In general, absolute code is faster than PIC because addressing calculations are simpler and shorter. Consult *Programming on HP-UX* for more information about absolute and position-independent code. See "Shared Libraries versus Archive Libraries" on page 179 for more information on the performance impact of shared libraries.

Using Profile-Based Optimization

HP C, HP C++, HP FORTRAN/9000, and HP Pascal support profile-based optimization (PBO) on HP 9000 systems. PBO can improve the performance of programs that are branch-intensive and that exhibit poor instruction memory locality. Although these tend not to be issues in floating-point-intensive applications, if you suspect that they may be degrading the performance of your program, you can use PBO to minimize their impact on your program. Under PBO, the compiler and linker work to optimize the executable file, using profile data for a typical data set to produce an executable file that will result in fewer instruction cache misses, **Translation Lookaside Buffer (TLB)** misses, and memory page faults. For information about PBO, see the HP-UX Linker and Libraries Online User Guide and the appropriate compiler documentation.

Creating and Zeroing Static Data (Fortran only)

HP Fortran 90 provides an option, `+save`, that forces static storage for all local variables and that forces the compiler to initialize all uninitialized static variables to zero. HP FORTRAN/9000 provides an equivalent option, `-K`; the `+e` option also automatically saves all local variables, if possible.

Use these options judiciously. They are costly from a performance standpoint and also from a software engineering perspective because they change the semantics of an entire module rather than altering specific problem areas.

The optimization option `+Oinitcheck` performs initialization in a more selective way that has less impact on the performance of your program. Use this option in Fortran 90 programs. See the *f90*(1) or *f77*(1) man page for details.

See “Static Variables” on page 188 for more information about static data.

Writing Routines in Assembly Language

If you have compiled with all of the correct compiler options and you are still not satisfied with the program’s performance, you may want to examine the generated code to see exactly what is happening. To get an

expanded listing, specify the `-S` option. You can also code parts of your program directly in assembly language. Assembly language is useful if performance is critical and portability is not.

When deciding whether to write something in assembly language, keep in mind that the HP 9000 compilers are highly optimizing. If the code section is large, the compiler can probably generate code as good as or better than an assembly language program. Good candidates for assembly language are short, frequently called routines. However, using the `+Oinline` compiler option may improve the performance of these routines enough to make it unnecessary to rewrite them in assembly language.

BLAS Library Versions

As we said in “Math Libraries and System Architecture” on page 25, the main HP-UX math libraries, `libm` and `libcl`, are provided in a version tuned for the PA-RISC 1.1 architecture that also runs well on PA-RISC 2.0 systems.

The two HP Fortran products, however, provide a version of the BLAS library, `libblas`, that is tuned to the PA-RISC 2.0 architecture. If you are compiling code that will run only on PA2.0 systems, use this library for optimum performance. If you use the `+DA2.0` option and link in the BLAS library, you will automatically link in the PA2.0 version.

See Chapter 4 for more information about math libraries. See “Determining Your System's Architecture Type” on page 26 if you do not know what kind of system you have.

Shared Libraries versus Archive Libraries

A program that is linked to shared libraries will generally run more slowly than a program that is linked to archive libraries. If you use archive libraries, the linker binds into your executable code an actual copy of each library routine you call. If you use shared libraries, the linker merely notes in your executable code that the code calls a routine in a shared library. Then, when the code begins execution, the dynamic loader loads and maps the shared libraries into the process's address space and calls the routines indirectly as they are needed by means of a linkage table. Using shared libraries saves space in the executable file, but at the expense of the time needed to resolve references to the routines in the shared libraries.

The performance impact of shared libraries is likely to be noticeable only if a program makes heavy use of library functions, as many floating-point applications do. If your program seems to be running unacceptably slowly with shared libraries, you may want to find out whether archive libraries make a difference.

For performance reasons, HP provides the BLAS library `libblas` only as an archive library, not as a shared library. The C math library `libm` and the Fortran and Pascal library `libcl`, however, are provided in both shared and archive versions. The linker by default looks for shared libraries before it looks for archive libraries, so if you want to use the archive library version of `libm` or `libcl`, you need to specify the `-a archive` option to the linker. (To do this on the compile command line, specify `-Wl,-a,archive`.)

See the HP-UX Linker and Libraries Online User Guide for more information about shared libraries and archive libraries.

Denormalized Operands

Denormalized values usually occur as the result of an operation that underflows. On HP 9000 systems, the occurrence of a denormalized operand or result, either at an intermediate stage of a computation or at the end, can reduce the speed of an operation significantly. There are several solutions to this problem:

- Change single-precision data to double-precision.
- Assign the value zero to data that would normally be denormalized.
- On systems that support it, enable flush-to-zero mode.
- Scale the entire data set upwards in magnitude so that the smallest values that occur are guaranteed to be normalized.

The first solution applies primarily to code that uses single-precision data. Because of the range of the two formats, a value that is denormalized in single-precision will be normalized in double-precision. On HP 9000 systems, denormalized numbers are so costly that it is worth converting to double-precision even if this means converting several operands from single-precision to double-precision, performing several operations, and then converting the result back to single-precision. The code will still run much faster than it would if it had to process a single denormalized operand.

The second solution is useful only if you can determine that your algorithm will work equally well when a denormalized value is treated as zero. In this case, you can explicitly assign it the value 0 before entering a loop where it is repeatedly accessed. For example, if A is a denormalized operand in the following example, this code will run very slowly on a HP 9000 system:

```
      SUBROUTINE VECTOR_SCALE(A, V)
      REAL A, V(1000)

      PRINT *, 'A IS', A
      DO 10 I = 1,1000
10    V(I) = V(I) * A
      RETURN
      END
```

If A is likely to be denormalized once in a while, it may be a good idea to add the following line before the loop:

```
      IF (ABS(A) .LT. 1.1754944E-38) A = 0.0
```

Of course, changing the denormalized value to zero can affect the accuracy of the algorithm. You need to determine whether this change will affect the usefulness of your program's output.

The third solution is to use flush-to-zero mode. If your target execution system supports flush-to-zero mode, you have the option of disabling gradual (that is, IEEE-754-compliant) underflow. To do so, use the `+FPD` compiler and linker option or the `fesetflushtozero` function. (The `fesetflushtozero` function always executes successfully. However, on system types that do not support fast underflow, the call has no effect.) These methods are described in Chapter 5. Running the sample program in "Underflow Mode: fegetflushtozero and fesetflushtozero" on page 143 will tell you whether your system supports fast underflow.

The final way to avoid denormalized numbers is to scale your entire algorithm upwards in magnitude. The exact way in which you do this depends on your algorithm because the results of most expressions do not scale up linearly as their operands are scaled. This method is more of a general way to rethink your overall algorithm than a way to fix the specific problem of denormalized values.

Mixed-Precision Expressions

Expressions that contain a mixture of single-precision and double-precision data generally execute more slowly than expressions of just one precision type because they require extra conversions. (Whether the compiler converts a mixed expression to single-precision or double-precision depends on the rules of the high-level language.) The reason is that HP 9000 instructions require their operands to be of the same precision. For example, for an add or multiply or any other two-operand instruction, both operands must be either single-precision or double-precision; they cannot be of mixed precision.

Aside from performance considerations, it is generally a good idea to avoid mixed-precision expressions for numerical analysis reasons. If you must mix precisions in a single expression, choose the precisions carefully so that the compiler does not need to perform repetitive conversions.

Matrix Operations

If a bottleneck contains vector and/or matrix operations, you may be able to improve program performance by specifying the `+Ovectorize` option. See "Optimizing Your Program" on page 169 for details.

Alternatively, you may be able to replace the operations with calls to the BLAS library, `libblas` (provided with the HP Fortran 90 and HP FORTRAN/9000 products only).

The `libblas` and `+Ovectorize` calls are faster than code loops that you can write yourself because they take into account alignment, data cache, and other machine-dependent characteristics. Not all matrices, however, are good candidates for `libblas` calls or for `+Ovectorize`. If the array contains fewer than about twenty elements, the overhead incurred by making the calls may offset the increased performance yielded by these routines.

For more information about the `libblas` routines, see "The BLAS Library (libblas)" on page 119, the *HP Fortran 90 Programmer's Reference*, and the *HP FORTRAN/9000 Programmer's Reference*.

Data Alignment

The term **alignment** refers to the type of address that a data object has in memory. Data objects can have 1-byte, 2-byte, 4-byte, or 8-byte alignment, meaning that the object is stored at an address evenly divisible by 1, 2, 4, or 8. In general, the best performance is obtained by **natural alignment**, which is the alignment that corresponds to the length of the object. For example, the natural alignment of an HP C `int` is 4-byte alignment.

For simple static variables, the compilers always naturally align data. The alignment problem occurs in C structures, Pascal records, and Fortran common blocks. By default, HP-UX compilers align this data naturally. However, you can specify directives in your program or on the command line that align data in a way that is compatible with data alignment on other HP systems. It is also possible to use Fortran `EQUIVALENCE` statements or other programming methods to obtain nondefault data alignment. Specifying nondefault data alignment causes the compiler to generate extra instructions, which both substantially increase code size and substantially degrade performance. Moreover, aligning data on a boundary less than its natural alignment boundary (for example, aligning a `double` on a 2-byte boundary) may result in a bus error or some other kind of run-time error.

In some situations, you can improve performance by aligning data on greater than natural addresses. This improvement is due to two factors, both of which concern the vector routines enabled by the `+Ovectorize` option:

- A single-precision array of numbers will sometimes allow better performance if it is 8-byte aligned, because the vector routines can use double-precision load and store operations to move two operands at a time.
- A double-precision array of numbers will sometimes allow better performance if it is 32-byte, or cache-line, aligned. This is because of the way the vector routines interact with the data caches on HP 9000 systems.

Cache Aliasing

HP 9000 systems employ high-speed cache memory to store the most recently used instructions and data. The location of instructions and data in the cache is a function of the *n* low-order bits of the address of the data or instruction. Consequently, instructions and data that have the same *n* low-order bits compete for the same cache space. Such competing objects are called **cache aliases** of each other.

Programs that contain loops in which two or more cache aliases are referenced will run more slowly than expected because the system must repeatedly swap aliased objects into and out of the cache.

Data cache aliasing can occur when two data objects that are far apart in virtual memory (so that their addresses have the same low-order bits but different high-order bits) are referenced in the same loop. **Instruction cache aliasing**, which is less likely to happen but has more serious effects, can occur when two routines invoked in a loop are aliases of each other. In this case, the invocation of the second routine forces the first routine out of the cache, and the next time through the loop the first routine pushes the second routine out of the cache.

Table 7-1 illustrates this situation using a 1K-byte cache example. If the cache addresses of `subr1` and `subr2` overlap significantly, instruction cache aliasing can produce a severe performance degradation.

Table 7-1 Typical Instruction Cache Aliasing Situation

Code	Description	Address

`DO 10 I = 1,1000`
`CALL subr1`
`CALL subr2`
`10 CONTINUE`
`END`
	Main program loop	

`SUBROUTINE subr1`
`.`
`.`
`RETURN`
`END`
	First subroutine	`P`
		
		`P+L1`
		`(L1 = length of subr1 code)`

`SUBROUTINE subr2`
`.`
`.`
`RETURN`
`END`
	Second subroutine	`Q`
		
		`Q+L2`
		`(L2 = length of subr2 code)`

Detecting problems with instruction cache aliases is an inexact process because the system does not keep records of cache hits and misses. You may notice, after making some modifications to one piece of code, that a seemingly unrelated routine runs more slowly than before. This can happen if your modifications change the virtual address map of your program and cause increased cache aliasing occurrences. Or you may suspect that a routine is running much more slowly than it should.

Fixing instruction cache problems is also difficult because you do not have much control over what virtual addresses are assigned to data and instructions. However, you can move routines around within a source module, and you can change the order in which object modules are joined by the linker. There are no hard and fast rules about how the linker orders various data and text sections, but it tends to place them in the same order in which you list them in the compiler or `ld` command line. You can take advantage of this fact by reordering the modules on the command line.

Data cache aliasing is a more common performance problem than instruction cache aliasing, but it is also easier to deal with. If you suspect that your application is experiencing data cache performance problems on scalar or small vectors of data, you can usually correct the situation by rearranging the order in which your variables are allocated in memory. In Fortran you can do this by reordering common block assignments.

If your application uses very large arrays of data, hundreds of kilobytes or megabytes in length, than chances are good that you will experience data cache aliasing problems no matter how your data arrays are allocated in memory, simply because the arrays are too big to fit in the data cache all at once. In this case, you may be able to improve performance by using the `+Odataprefetch` option, either alone or in conjunction with `+Ovectorize`. (See “Optimizing Your Program” on page 169 for information about these options.)

Another way to improve performance for very large arrays is to increase the locality of references to your data. This technique, sometimes called **tiling**, requires selecting an algorithm that processes as much data as possible while the data is resident in the cache so as to minimize the number of times the data must be re-cached later. The routines in the BLAS library use tiling when they operate on large matrices of data. For example, in multiplying two large matrices, each matrix is cut into tiles, which are processed in pairs. The size of the tiles is determined at run time by the size of the matrices and the size of the data cache on the system executing the application.

Static Variables

A **static variable** retains its value between invocations of the routine in which it is declared. From a performance standpoint, static variables are costly because they prohibit the compiler from making certain types of optimizations. For example, a subroutine that contains a static variable is ineligible for certain optimizations.

There are several ways to give a variable static storage class. In C, you can declare the variable globally, or you can use the `static` storage class specifier. In Fortran, you can use the `+save`, `-K` or `+e` option (see "Creating and Zeroing Static Data (Fortran only)" on page 176); place variables in a common block; specify them in a `SAVE` statement; use a `DATA` statement to initialize variables; or give the initial value in a declaration. The preferred method is always to specify static duration only for those variables that absolutely must be static. For example, you should be particularly cautious about using the `+save`, `-K` and `+e` options because they place all of a program's local variables in static storage.

Quad-Precision Computations

Computations involving quad-precision operands are inherently much slower than those involving double-precision or single-precision operands. The reason is that HP 9000 hardware supports only single-precision and double-precision operations. All quad-precision operations, even the simplest arithmetic ones, are performed in software. For this reason, you should use quad-precision for performance-sensitive code only if your application absolutely requires this degree of precision.

A The C Math Library

The HP-UX C math library, `libm`, supports all mathematical functions specified by the ANSI C standard, *ANS X3.159-1989*, as well as functions specified by the XPG4.2, SVID, and COSE Common API (Spec 1170) specifications.

In addition, the library supports the following value-added functions specific to HP-UX:

- `float` versions of many mathematical functions
- Degree-valued trigonometric functions
- A group of functions and macros recommended by the IEEE standard (see Table 2-12 on page 65), including `fpclassify`, `copysign`, and `isfinite`
- An additional group of floating-point classification macros approved by the ISO/ANSI C committee for inclusion in the C9X draft standard
- The *fenv*(5) suite, a collection of functions (approved by the ISO/ANSI C committee for inclusion in the C9X draft standard) that allow an application to manipulate the floating-point status register (see Chapter 5 for more information)

If your program calls `libm` math functions, you must link in the appropriate library explicitly. See "Locations of the Math Libraries at Release 10.30" on page 27 for a list of the different versions of `libm` and their directory locations.

For more information about math libraries, including C math library error handling, see "HP-UX Math Libraries on HP 9000 Systems" on page 95. For details about these functions, see the online man pages.

C Math Library Tables

Table A-1 lists by category all of the functions provided by the C math library. Table A-2 lists the functions alphabetically. All of these functions are defined in the header files `/usr/include/math.h` and `/usr/include/fenv.h`.

If a millicode version of a function exists, the tables note it in parentheses. See “Millicode Versions of Math Library Functions” on page 112 for details on how to obtain these versions.

Table A-1 The C Math Library (By Category)

Function	What It Does
Trigonometric Functions for Radian-Value Arguments	
`double acos(double x);`	Returns arccosine of *x* in radians. (Millicode version available)
`double asin(double x);`	Returns arcsine of *x* in radians. (Millicode version available)
`double atan(double x);`	Returns arctangent of *x* in radians. (Millicode version available)
`double atan2(double y, double x);`	Returns arctangent of *y*/*x* in radians. (Millicode version available)
`double cos(double x);`	Returns cosine of *x* (*x* in radians). (Millicode version available)
`double sin(double x);`	Returns sine of *x* (*x* in radians). (Millicode version available)
`double tan(double x);`	Returns tangent of *x* (*x* in radians). (Millicode version available)
Trigonometric Functions for Degree-Value Arguments	
`double acosd(double x);`	Returns arccosine of *x* in degrees.
`double asind(double x);`	Returns arcsine of *x* in degrees.
`double atand(double x);`	Returns arctangent of *x* in degrees.
`double atan2d(double y, double x);`	Returns arctangent of *y*/*x* in degrees.
`double cosd(double x);`	Returns cosine of *x* (*x* in degrees).
`double sind(double x);`	Returns sine of *x* (*x* in degrees).
`double tand(double x);`	Returns tangent of *x* (*x* in degrees).
Other Transcendental Functions	
`double acosh(double x);`	Returns inverse hyperbolic cosine of *x*.
`double asinh(double x);`	Returns inverse hyperbolic sine of *x*.

Function	What It Does
Other Transcendental Functions (cont.)	
`double atanh(double x);`	Returns inverse hyperbolic tangent of *x*.
`double cosh(double x);`	Returns hyperbolic cosine of *x*.
`double erf(double x);`	Returns error function of *x*.
`double erfc(double x);`	Returns 1.0 – erf(*x*).
`double exp(double x);`	Returns exponential function of *x*. (Millicode version available)
`double expm1(double x);`	Returns exp(*x*) – 1.
`double gamma(double x);`	Returns logarithm of the absolute value of the gamma function of *x* (same as `lgamma`). (Obsolescent; will become true gamma function at a future release)
`int ilogb(double x);`	Returns the integer form of the binary exponent of the floating-point value *x*.
`double j0(double x);`	Returns Bessel function of *x* of the first kind of order 0.
`double j1(double x);`	Returns Bessel function of *x* of the first kind of order 1.
`double jn(int n, double x);`	Returns Bessel function of *x* of the first kind of order *n*.
`double lgamma(double x);`	Returns logarithm of the absolute value of the gamma function of *x*.
`double lgamma_r(double x, int *sign);`	Returns logarithm of the absolute value of the gamma function of *x* (reentrant version of `lgamma`).
`double log(double x);`	Returns natural logarithm of *x*. (Millicode version available)
`double log10(double x);`	Returns base 10 logarithm of *x*. (Millicode version available)

Function	What It Does
Other Transcendental Functions (cont.)	
`double log1p(double x);`	Returns log(1 + *x*).
`double log2(double x);`	Returns base 2 logarithm of *x*.
`double logb(double x);`	Returns the exponent of *x* as an integer-valued double-precision number; formally, the integral part of $\log_2 \lvert x \rvert$.
`double pow(double x, double y);`	Returns *x* to the power *y*. (Millicode version available)
`double scalb(double x, double n);`	Returns *x**(2***n*), computed efficiently.
`double sinh(double x);`	Returns hyperbolic sine of *x*.
`double tanh(double x);`	Returns hyperbolic tangent of *x*.
`double y0(double x);`	Returns Bessel function of *x* of the second kind of order 0.
`double y1(double x);`	Returns Bessel function of *x* of the second kind of order 1.
`double yn(int n, double x);`	Returns Bessel function of *x* of the second kind of order *n*.
Miscellaneous Mathematical Functions	
`double cabs(struct {double x, y} z);`	Obsolete (replaced by `hypot`).
`double cbrt(double x);`	Returns cube root of *x*.
`double ceil(double x);`	Returns smallest integral value not less than *x*.
`double copysign(double x, double y);`	Returns *x* with its sign changed to *y*'s.
`double drem(double x, double y);`	Obsolete (replaced by `remainder`).
`double fabs(double x);`	Returns absolute value of *x*.
`double floor(double x);`	Returns largest integral value not greater than *x*.

Function	What It Does
Miscellaneous Mathematical Functions (cont.)	
`double fmod(double x, double y);`	Returns floating-point remainder (*f*) of the division of *x* by *y*, where *f* has the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $\|f\| < \|y\|$.
`double frexp(double value, int *eptr);`	Returns fraction part of a `double` *value*; stores exponent in the location pointed to by *eptr*.
`double hypot(double x, double y);`	Returns sqrt(*x* * *x* + *y* * *y*).
`double ldexp(double value, int exp);`	Returns *value**(2***exp*).
`int _matherr(struct exception *x);`	Obsolete; see "Math Library Error Handling for C" on page 102.
`double modf(double value, double *iptr);`	Returns signed fractional part of *value*; stores integral part in the location pointed to by *iptr*.
`double remainder(double x, double y);`	Returns the remainder $r = x - n*y$ where *n* is the integer nearest the exact value of *x*/*y*. Implements the IEEE remainder operation.
`double rint(double x);`	Rounds *x* to integer-valued double-precision number, in the direction of the current rounding mode.
`double sqrt(double x);`	Returns nonnegative square root of *x*.
float Versions of Math Functions	
`float acosdf(float x);`	Returns arccosine of *x* in degrees.
`float acosf(float x);`	Returns arccosine of *x* in radians.
`float asindf(float x);`	Returns arcsine of *x* in degrees.
`float asinf(float x);`	Returns arcsine of *x* in radians.
`float atandf(float x);`	Returns arctangent of *x* in degrees.

Function	What It Does
float Versions of Math Functions (cont.)	
`float atanf(float x);`	Returns arctangent of *x* in radians.
`float atan2df(float y, float x);`	Returns arctangent of *y*/*x* in degrees.
`float atan2f(float y, float x);`	Returns arctangent of *y*/*x* in radians.
`float cbrtf(float x);`	Returns cube root of *x*.
`float copysignf(float x, float y);`	Returns *x* with its sign changed to *y*'s.
`float cosdf(float x);`	Returns cosine of *x* (*x* in degrees).
`float cosf(float x);`	Returns cosine of *x* (*x* in radians). (Millicode version available)
`float coshf(float x);`	Returns hyperbolic cosine of *x*.
`float expf(float x);`	Returns exponential function of *x*.
`float fabsf(float x);`	Returns absolute value of *x*.
`float fmodf(float x, float y);`	Returns floating-point remainder (*f*) of the division of *x* by *y*, where *f* has the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $\|f\| < \|y\|$.
`float logf(float x);`	Returns natural logarithm of *x*. (Millicode version available)
`float log10f(float x);`	Returns base 10 logarithm of *x*.
`float log2f(float x);`	Returns base 2 logarithm of *x*.
`float powf(float x, float y);`	Returns *x* to the power *y*.
`float sindf(float x);`	Returns sine of *x* (*x* in degrees).
`float sinhf(float x);`	Returns hyperbolic sine of *x*.
`float sinf(float x);`	Returns sine of *x* (*x* in radians). (Millicode version available)
`float sqrtf(float x);`	Returns nonnegative square root of *x*.

Function	What It Does
float Versions of Math Functions (cont.)	
`float tandf(float x);`	Returns tangent of *x* (*x* in degrees).
`float tanf(float x);`	Returns tangent of *x* (*x* in radians). (Millicode version available)
`float tanhf(float x);`	Returns hyperbolic sine of *x*.
Floating-Point Classification Macros and Function	
`int isfinite(`*floating-type x*`);`	Returns a nonzero value just when –infinity < *x* < +infinity; returns 0 otherwise (when |*x*| = infinity or *x* is NaN).
`int fpclassify(`*floating-type x*`);`	Returns the IEEE class of *x*.
`int isinf(`*floating-type x*`);`	Returns a nonzero integer if *x* is an infinity. Otherwise returns zero.
`int isnan(`*floating-type x*`);`	Returns a nonzero integer if *x* is NaN (not-a-number). Otherwise returns zero.
`int isnormal(`*floating-type x*`);`	Returns a nonzero integer if *x* is a normalized value. Otherwise returns zero.
`double nextafter(double x, double y);`	Returns the next representable neighbor of *x* in the direction of *y*.
`int signbit(`*floating-type x*`);`	Returns a nonzero integer if the sign of *x* is negative. Otherwise returns zero.
fenv Suite	
`void feclearexcept(int excepts);`	Clears the exception flags represented by *excepts*.
`void fegetenv(fenv_t *envp);`	Stores the current floating-point environment in the object pointed to by *envp*.

Function	What It Does
fenv Suite (cont.)	
`void fegetexceptflag (fexcept_t *`*flagp*`, int` *excepts*`);`	Stores the exception flags indicated by the argument *excepts* in the object pointed to by the argument *flagp*.
`int fegetflushtozero(void);`	Retrieves the value representing the current underflow mode.
`int fegetround(void);`	Returns the current rounding direction.
`int fegettrapenable(void);`	Determines which exception trap enable bits are currently set.
`int feholdexcept(fenv_t *`*envp*`);`	Saves the current floating-point environment in the object pointed to by *envp*, clears the exception flags, and disables all traps.
`void feraiseexcept(int` *excepts*`);`	Raises the exceptions represented by *excepts*.
`void fesetenv(const fenv_t *`*envp*`);`	Establishes the floating-point environment respresented by the object pointed to by *envp*.
`void fesetexceptflag (const fexcept_t *`*flagp*`, int` *excepts*`);`	Sets the status for the exception flags indicated by the argument *excepts* according to the representation in the object pointed to by *flagp*.
`void fesetflushtozero(int` *mode*`);`	Establishes the underflow mode represented by *mode*.
`int fesetround(int` *round*`);`	Establishes the rounding direction represented by *round*.
`void fesettrapenable(int` *excepts*`);`	Sets the exception trap enable bits as indicated by the argument *excepts*.

Function	What It Does
fenv Suite (cont.)	
`int fetestexcept(int` *excepts*`);`	Determines which of a specified subset of the exception flags are currently set.
`void feupdateenv (const fenv_t *`*envp*`);`	Saves the current exceptions in its automatic storage, installs the floating-point environment represented through *envp*, and then raises the saved exceptions.

Table A-2 The C Math Library (Alphabetical Listing)

Function	What It Does
`double acos(double x);`	Returns arccosine of *x* in radians. (Millicode version available)
`double acosd(double x);`	Returns arccosine of *x* in degrees.
`float acosdf(float x);`	Returns arccosine of *x* in degrees.
`float acosf(float x);`	Returns arccosine of *x* in radians.
`double acosh(double x);`	Returns inverse hyperbolic cosine of *x*.
`double asin(double x);`	Returns arcsine of *x* in radians. (Millicode version available)
`double asind(double x);`	Returns arcsine of *x* in degrees.
`float asindf(float x);`	Returns arcsine of *x* in degrees.
`float asinf(float x);`	Returns arcsine of *x* in radians.
`double asinh(double x);`	Returns inverse hyperbolic sine of *x*.
`double atan(double x);`	Returns arctangent of *x* in radians. (Millicode version available)
`double atand(double x);`	Returns arctangent of *x* in degrees.
`float atandf(float x);`	Returns arctangent of *x* in degrees.
`float atanf(float x);`	Returns arctangent of *x* in radians.
`double atanh(double x);`	Returns inverse hyperbolic tangent of *x*.
`double atan2(double y, double x);`	Returns arctangent of *y*/*x* in radians. (Millicode version available)
`double atan2d(double y, double x);`	Returns arctangent of *y*/*x* in degrees.
`float atan2df(float y, float x);`	Returns arctangent of *y*/*x* in degrees.
`float atan2f(float y, float x);`	Returns arctangent of *y*/*x* in radians.
`double cabs(struct {double x, y} z);`	Obsolete (replaced by `hypot`).
`double cbrt(double x);`	Returns cube root of *x*.

Function	What It Does
`float cbrtf(float x);`	Returns cube root of *x*.
`double ceil(double x);`	Returns smallest integral value not less than *x*.
`double copysign(double x, double y);`	Returns *x* with its sign changed to *y*'s.
`float copysignf(float x, float y);`	Returns *x* with its sign changed to *y*'s.
`double cos(double x);`	Returns cosine of *x* (*x* in radians). (Millicode version available)
`double cosd(double x);`	Returns cosine of *x* (*x* in degrees).
`float cosdf(float x);`	Returns cosine of *x* (*x* in degrees).
`float cosf(float x);`	Returns cosine of *x* (*x* in radians). (Millicode version available)
`double cosh(double x);`	Returns hyperbolic cosine of *x*.
`float coshf(float x);`	Returns hyperbolic cosine of *x*.
`double drem(double x, double y);`	Obsolete (replaced by `remainder`).
`double erf(double x);`	Returns error function of *x*.
`double erfc(double x);`	Returns 1.0 – erf(*x*).
`double exp(double x);`	Returns exponential function of *x*. (Millicode version available)
`float expf(float x);`	Returns exponential function of *x*.
`double expm1(double x);`	Returns exp(*x*) – 1.
`double fabs(double x);`	Returns absolute value of *x*.
`float fabsf(float x);`	Returns absolute value of *x*.
`void feclearexcept(int excepts);`	Clears the exception flags represented by *excepts*.
`void fegetenv(fenv_t *envp);`	Stores the current floating-point environment in the object pointed to by *envp*.

Function	What It Does
`void fegetexceptflag (fexcept_t *`*flagp*`, int` *excepts*`);`	Stores the exception flags indicated by the argument *excepts* in the object pointed to by the argument *flagp*.
`int fegetflushtozero(void);`	Retrieves the value representing the current underflow mode.
`int fegetround(void);`	Returns the current rounding direction.
`int fegettrapenable(void);`	Determines which exception trap enable bits are currently set.
`int feholdexcept(fenv_t *`*envp*`);`	Saves the current floating-point environment in the object pointed to by *envp*, clears the exception flags, and disables all traps.
`void feraiseexcept(int` *excepts*`);`	Raises the exceptions represented by *excepts*.
`void fesetenv(const fenv_t *`*envp*`);`	Establishes the floating-point environment respresented by the object pointed to by *envp*.
`void fesetexceptflag (const fexcept_t *`*flagp*`, int` *excepts*`);`	Sets the status for the exception flags indicated by the argument *excepts* according to the representation in the object pointed to by *flagp*.
`void fesetflushtozero(int` *mode*`);`	Establishes the underflow mode represented by *mode*.
`int fesetround(int` *round*`);`	Establishes the rounding direction represented by *round*.
`void fesettrapenable(int` *excepts*`);`	Sets the exception trap enable bits as indicated by the argument *excepts*.
`int fetestexcept(int` *excepts*`);`	Determines which of a specified subset of the exception flags are currently set.

Function	What It Does
`void feupdateenv (const fenv_t *envp);`	Saves the current exceptions in its automatic storage, installs the floating-point environment represented through *envp*, and then raises the saved exceptions.
`double floor(double x);`	Returns largest integral value not greater than *x*.
`double fmod(double x, double y);`	Returns floating-point reainder (*f*) of the division of *x* by *y*, where *f* has the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $\|f\| < \|y\|$.
`float fmodf(float x, float y);`	Returns floating-point remainder (*f*) of the division of *x* by *y*, where *f* has the same sign as *x*, such that $x = iy + f$ for some integer *i*, and $\|f\| < \|y\|$.
`int fpclassify(floating-type x);`	Returns the IEEE class of *x*.
`double frexp(double value, int *eptr);`	Returns fraction part of a `double` *value*; stores exponent in the location pointed to by *eptr*.
`double gamma(double x);`	Returns logarithm of the absolute value of the gamma function of *x* (same as `lgamma`). (Obsolescent; will become true gamma function at a future release)
`double hypot(double x, double y);`	Returns sqrt(*x* * *x* + *y* * *y*).
`int ilogb(double x);`	Returns the integer form of the binary exponent of the floating-point value *x*.
`int isfinite(floating-type x);`	Returns a nonzero value just when -infinity < *x* < +infinity; returns 0 otherwise (when $\|x\|$ = infinity or *x* is NaN).
`int isinf(floating-type x);`	Returns a nonzero integer if *x* is an infinity. Otherwise returns zero.

Function	What It Does
`int isnan(`*floating-type x*`);`	Returns a nonzero integer if *x* is NaN (not-a-number). Otherwise returns zero.
`int isnormal(`*floating-type x*`);`	Returns a nonzero integer if *x* is a normalized value. Otherwise returns zero.
`double j0(double` *x*`);`	Returns Bessel function of *x* of the first kind of order 0.
`double j1(double` *x*`);`	Returns Bessel function of *x* of the first kind of order 1.
`double jn(int` *n*`, double` *x*`);`	Returns Bessel function of *x* of the first kind of order *n*.
`double ldexp(double` *value*`, int` *exp*`);`	Returns *value**(2***exp*).
`double lgamma(double` *x*`);`	Returns logarithm of the absolute value of the gamma function of *x*.
`double lgamma_r(double` *x*`,` `int *`*sign*`);`	Returns logarithm of the absolute value of the gamma function of *x* (reentrant version of `lgamma`).
`double log(double` *x*`);`	Returns natural logarithm of *x*. (Millicode version available)
`double logb(double` *x*`);`	Returns the exponent of *x* as an integer-valued double-precision number; formally, the integral part of $\log_2 \lvert x \rvert$.
`float logf(float` *x*`);`	Returns natural logarithm of *x*. (Millicode version available)
`double log10(double` *x*`);`	Returns base 10 logarithm of *x*. (Millicode version available)
`float log10f(float` *x*`);`	Returns base 10 logarithm of *x*.
`double log1p(double` *x*`);`	Returns log(1 + *x*).
`double log2(double` *x*`);`	Returns base 2 logarithm of *x*.
`float log2f(float` *x*`);`	Returns base 2 logarithm of *x*.

Function	What It Does
`int _matherr(struct exception *x);`	Obsolete; see "Math Library Error Handling for C" on page 102.
`double modf(double value, double *iptr);`	Returns signed fractional part of *value*; stores integral part in the location pointed to by *iptr*.
`double nextafter(double x, double y);`	Returns the next representable neighbor of *x* in the direction of *y*.
`double pow(double x, double y);`	Returns *x* to the power *y*. (Millicode version available)
`float powf(float x, float y);`	Returns *x* to the power *y*.
`double remainder(double x, double y);`	Returns the remainder $r = x - n*y$ where *n* is the integer nearest the exact value of x/y. Implements the IEEE remainder operation.
`double rint(double x);`	Rounds *x* to integer-valued double-precision number, in the direction of the current rounding mode.
`double scalb(double x, double n);`	Returns *x**(2***n*), computed efficiently.
`int signbit(floating-type x);`	Returns a nonzero integer if the sign of *x* is negative. Otherwise returns zero.
`double sin(double x);`	Returns sine of *x* (*x* in radians). (Millicode version available)
`double sind(double x);`	Returns sine of *x* (*x* in degrees).
`float sindf(float x);`	Returns sine of *x* (*x* in degrees).
`float sinf(float x);`	Returns sine of *x* (*x* in radians). (Millicode version available)
`double sinh(double x);`	Returns hyperbolic sine of *x*.
`float sinhf(float x);`	Returns hyperbolic sine of *x*.
`double sqrt(double x);`	Returns nonnegative square root of *x*.

Function	What It Does
`float sqrtf(float x);`	Returns nonnegative square root of *x*.
`double tan(double x);`	Returns tangent of *x* (*x* in radians). (Millicode version available)
`double tand(double x);`	Returns tangent of *x* (*x* in degrees).
`float tandf(float x);`	Returns tangent of *x* (*x* in degrees).
`float tanf(float x);`	Returns tangent of *x* (*x* in radians). (Millicode version available)
`double tanh(double x);`	Returns hyperbolic tangent of *x*.
`float tanhf(float x);`	Returns hyperbolic sine of *x*.
`double y0(double x);`	Returns Bessel function of *x* of the second kind of order 0.
`double y1(double x);`	Returns Bessel function of *x* of the second kind of order 1.
`double yn(int n, double x);`	Returns Bessel function of *x* of the second kind of order *n*.

B The Fortran Math Library

The HP-UX Fortran and Pascal library, `libcl`, contains all Fortran and Pascal library routines (for example, input/output routines) as well as mathematical functions.

The mathematical functions that `libcl` supports are as follows:

- The Fortran intrinsic functions, which are described in the *HP Fortran 90 Programmer's Reference* and the *HP FORTRAN/9000 Programmer's Reference*. Fortran intrinsic functions have single-precision, double-precision, and quad-precision versions. (See "Quad-Precision Computations" on page 189 for information about the performance impact of quad-precision computations.)
- The Pascal predefined math functions, which are described in the *HP Pascal/HP-UX Reference Manual*.

If you compile and link with the `f90`, `f77`, or `pc` command, a version of `libcl` is linked in automatically. You may specify a nondefault location for `libcl` by linking in the library explicitly. See "Locations of the Math Libraries at Release 10.30" on page 27 for a list of the different versions of `libcl` and their directory locations.

You can call routines from the BLAS library, `libblas`, by specifying `-lblas` on the `f77` command line. See "The BLAS Library (libblas)" on page 119 and the *HP FORTRAN/9000 Programmer's Reference* for information about this library.

NOTE

The BLAS library is provided with the HP Fortran 90 and HP FORTRAN/9000 products only.

For more information about math libraries, including Fortran math library error handling, see Chapter 4.

C Floating-Point Problem Checklist

This appendix provides a checklist of problems you might encounter when compiling, linking, or running a floating-point application. For each kind of problem, we list possible reasons for the problem and direct you to the section of this manual where the reason is discussed.

We divide the problems into two categories:

- Results that are different from those produced when the application was run previously
- Results that are clearly incorrect or are much less precise than expected

These categories are not mutually exclusive. If you find your application produces results that are both different from before and inaccurate, look at the suggestions in both categories.

Finally, we discuss the possible causes of compiling and linking errors you may get when you build a floating-point application.

Results Different from Those Produced Previously

If your application produces results that are somewhat different from those produced when you ran it before (either on HP-UX or on another vendor's system), consider the following possibilities:

- Are you porting an application to HP-UX from another vendor's system? If the other system does not comply with the IEEE standard, your results will almost certainly be different on HP-UX. If the other system is also IEEE-compliant, slight differences in results may come from differences in the rounding errors introduced by variations in the ordering of basic operations within expressions or within library functions.

  See Chapter 2 for an overview of the IEEE standard. See "How Basic Operations Affect Application Results" on page 69 for information on how basic operations affect results.

- Did you use a different version of the math library? Different versions of the HP-UX operating system will often contain improved versions of the math libraries. Your application may be affected by these changes if it uses shared libraries or if you have rebuilt it on a system that is running a different version of HP-UX. Using the +DA option on the compile command line also affects which library version you use.

  For details, see "Math Libraries and System Architecture" on page 25, "Operating System Release of Build-Time System" on page 79, and "Operating System Release of Run-Time System" on page 79.

  For more information about the effects of math library changes, see "How Mathematical Library Functions Affect Application Results" on page 71 and "Operating System Release of Build-Time System" on page 79. For specific information about the enhanced HP-UX math libraries, see "Contents of the HP-UX Math Libraries" on page 112, Appendix A, and Appendix B.

- Are you using a different compiler, or a different version of the compiler? A new compiler version may perform such operations as constant parsing and constant folding somewhat differently. It may also order some expressions somewhat differently.

For information about the effects of compiler version changes, see “Compiler Behavior and Compiler Version” on page 76.

- Are you using different compiler options? Compiling at a different optimization level may change the way the compiler performs such operations as expression reordering and operation reordering.

  For more information, see “Compiler Options” on page 77 and “Run-Time Mode Control: The fenv(5) Suite” on page 125.

- Have you changed the rounding mode or the underflow mode for your application? If you use the `fesetround` function to change the rounding mode from the default (round to nearest), application results will probably change. If you use the `fesetflushtozero` function or the `+FP` compiler option to change the underflow mode to flush-to-zero mode, results that underflow are flushed to 0 instead of denormalized, and application results may also change.

  For more information, see “Values of Certain Modifiable Hardware Status Register Fields” on page 80 and all of Chapter 5.

Incorrect or Imprecise Results

If your application produces results that are either clearly incorrect or much less precise than you expected, consider the following possibilities:

- Did your application encounter an exception condition, either in a basic operation or in a math library function? Exceptions such as overflows, or invalid arguments to library functions, can introduce a dramatic amount of error into a computation. On HP-UX systems, exceptions by default do not prevent an application from continuing; you must detect exceptions explicitly.

  For information on how exceptions can affect application results, see "How Exceptions and Library Errors Affect Application Results" on page 72. For information on math library error handling, see "Math Library Basics" on page 99. For information on how to detect and trap exceptions, see Chapter 6.

- Did you forget to include `math.h` in a C program that calls math library functions? Or did you compile your C program with `-Aa` (strict ANSI) but call a math function not specified by the ANSI C standard? If so, the C compiler generates code that expects the default C function return value of type int, and the function returns an incorrect result.

  For more information, see "The C Math Library (libm)" on page 114.

- Does your application convert values properly from decimal (ASCII) to binary and vice versa? Check the points at which your program assigns constant values to variables, reads file or keyboard input into variables, and writes variables to files or to the screen. Make sure that the format specifications in your program allow the right amount of precision in the values being read and written.

  For more information on conversions between decimal and binary, see "Conversions Between Decimal and Binary Floating-Point Format" on page 52 and "Conversions Between Binary and Decimal" on page 74.

- Does your program test floating-point values for equality? Because of the inherent inexactness of floating-point representations, values that should be equal from a purely algebraic perspective in fact rarely are.

For more information, see “How Basic Operations Affect Application Results” on page 69 and “Testing Floating-Point Values for Equality” on page 82.

- Does your application take the difference of floating-point values that are similar in magnitude? Such subtraction can result in the loss of many significant digits.

  For more information, see “Taking the Difference of Similar Values” on page 85.

- Conversely, does your application add values that are very different in magnitude? Such addition can also result in the loss of many significant digits.

  For more information, see “Adding Values with Very Different Magnitudes” on page 86.

- Does your application underflow? An unintentional underflow during a calculation can result in the loss of many significant digits.

  For more information, see “Unintentional Underflow” on page 89.

- Does your application truncate floating-point values to integer? In C and Fortran, assigning a floating-point value to an integer variable causes the value to be truncated, not rounded to the nearest integer.

  For more information, see “Truncation to an Integer Value” on page 90.

- Does your application contain ill-conditioned computations—those that use arguments near a function’s singularity, where small differences in input produce results so different that the results are meaningless?

  For more information, see “Ill-Conditioned Computations” on page 92.

- (Fortran only) Are you computing in double or quad precision, but have one or more undeclared variables, which in Fortran default to single precision?

  For more information, see the *HP FORTRAN/9000 Programmer’s Reference*.

- Does your program call functions in the *fenv*(5) suite of routines, and did you compile it at an optimization level greater than `+O0`? If so, operation reordering around the function calls may have unexpected effects.

For more information, see “Run-Time Mode Control: The fenv(5) Suite” on page 125.

Compiling and Linking Errors

If the C compiler issues an error message like the following:

```
cc: "myprog.c", line 16: error 1588: "FE_ALL_EXCEPT" undefined.
cc: "atanh.c", line 59: error 1588: "HUGE_VAL" undefined.
```

make sure your program includes the `fenv.h` and/or the `math.h` header file. If it does, make sure you are compiling with the default `-Ae` option or with both the `-Aa` and `-D_HPUX_SOURCE` options. See "Scalar Math Libraries (libm and libcl)" on page 112 for details.

If the linker issues an error message like the following:

```
/usr/ccs/bin/ld: Unsatisfied symbols:
   log10 (code)
   sqrt (code)
   pow (code)
```

you may have forgotten to link in a math library (`-lm`). See "Scalar Math Libraries (libm and libcl)" on page 112 for details. If the linker issues a similar message for a macro:

```
/usr/ccs/bin/ld: Unsatisfied symbols:
   isfinite (code)
```

make sure your program includes the `math.h` header file.

If the linker issues an error message like the following:

```
/usr/ccs/bin/ld: Illegal argument combination
(/usr/lib/libm.a(cs_fabsf.o), fabsf)
```

make sure you are compiling with the default `-Ae` option or with both the `-Aa` and `-D_HPUX_SOURCE` options. See "Scalar Math Libraries (libm and libcl)" on page 112 for details.

If you get these kinds of messages when you compile or link a C++ program, make sure you are compiling with the `+a1` and `-D_HPUX_SOURCE` options.

Glossary

Items in **this font** represent terms that are defined elsewhere in the glossary.

addition One of the basic operations defined by the **IEEE standard**.

alignment The type of address that a data object has in memory. Data objects can have 1-byte, 2-byte, 4-byte, or 8-byte alignment, meaning that the object is stored at an address evenly divisible by 1, 2, 4, or 8.

archive library A collection of object modules. When an application is linked with an archive **library**, the linker scans the contents of the archive library and extracts the object modules that satisfy any unresolved references in the application. The linker copies the archive library modules into the application's code section. *See also* **shared library**.

aware A version of a **comparison** assertion that treats a **NaN (Not-a-Number)** as a special value that compares as neither less than nor greater than any **normalized value**, and as unequal to any value, including another NaN and itself. *See also* **non-aware**.

bias In a **floating-point** representation, a value that is subtracted from the represented **exponent** to get the actual exponent. For **single-precision** formats, the bias is 127; for **double-precision** formats, it is 1023; for **quad-precision** formats, it is 16383. *See also* **biased representation**.

biased representation A **floating-point** representation that adds a constant value (the **bias**) to the actual **exponent**, so that actual exponents, which may be negative or positive, are always represented as positive.

BLAS library The Basic Linear Algebra Subroutine (BLAS) library, a **math library** that contains routines that perform low-level vector and matrix (array) operations. This library is provided with the HP Fortran 90 and FORTRAN/9000 products only.

cache aliases Two sets of data or instruction addresses that have the same *n* low-order bits and therefore occupy the same cache

address. *See also* **data cache aliasing**, **instruction cache aliasing**.

ceiling The ceiling of a value is the smallest whole number greater than that value. *See also* **floor**.

comparison One of the basic operations defined by the **IEEE standard**. The comparison operation determines the truth of an assertion about the relationship of two **floating-point** values. There are four possible relations: less than, equal, greater than, and **unordered**. *See also* **aware**, **non-aware**.

constant folding A compile-time expression evaluation that determines whether an expression evaluates to a constant and, if it does, replaces the expression with the constant.

control register See **floating-point status register**.

conversion One of the basic operations defined by the **IEEE standard**.

data cache aliasing The form of cache aliasing that occurs when the addresses of two data objects have the same low-order bits but different high-order bits. *See also* **cache aliases**, **instruction cache aliasing**.

denormalized value A **floating-point** value that is represented by a **sign bit**, a zero **exponent**, and a non-zero **fraction**. (If the fraction were also zero, the floating-point value would be **zero**.) A denormalized value has a magnitude greater than zero and less than any **normalized value**.

division One of the basic operations defined by the **IEEE standard**.

division by zero condition The **exception condition** that occurs when the system attempts to divide a nonzero, finite value by **zero**; more generally, when an exact infinity is produced from finite operands.

domain errors Errors generated by **math library** routines when they encounter invalid arguments. *See also* **range errors**.

double-extended precision *See* **quad-precision**.

double-precision An IEEE **floating-point** format in which a value occupies 64 bits: 1 bit for the

sign, 11 bits for the **exponent**, and 52 bits for the **fraction**. *See also* **single-precision**, **quad-precision**.

error condition *See* **exception condition**.

exception *See* **exception condition**.

exception condition A condition that may require special handling to make further execution of an application meaningful. In many applications, the occurrence of an exception indicates an error. The **IEEE standard** specifies five exception conditions: the **inexact result condition**, the **overflow condition**, the **underflow condition**, the **invalid operation condition**, and the **division by zero condition**. *See also* **trap handler**.

exception flags A group of bits in the **floating-point status register**. If an **exception condition** occurs and the corresponding **exception trap enable bit** is not set, the **floating-point unit (FPU)** sets the corresponding exception flag to 1, but does not cause a trap.

exception trap enable bits A group of bits in the **floating-point status register**. If an **exception condition** occurs and the corresponding exception trap enable bit is set, the **floating-point unit (FPU)** causes a trap. When the enable bit equals 0, the exception usually sets the corresponding **exception flag** to 1 instead of causing a trap.

exponent In a **floating-point** representation, the bits that represent a value to which 2.0 is raised. *See also* **fraction**, **sign bit**.

fast underflow mode *See* **flush-to-zero mode**.

fastmode *See* **flush-to-zero mode**.

finite value A representable floating-point value (that is, not an **infinity** or a **NaN (Not-a-Number)**).

floating-point Of or pertaining to the method by which computer systems represent and operate on real numbers. The **IEEE standard** specifies that a floating-point value consists of a **sign bit**, a **fraction**, and an **exponent**.

floating-point status register A register in the **floating-point unit (FPU)** on PA systems that controls the arithmetic **rounding mode**, controls the underflow mode (on many systems), enables user-level traps, indicates exceptions that have occurred, indicates the result of a **comparison**, and contains information to identify the implementation of the floating-point unit.

floating-point unit (FPU) A coprocessor that performs IEEE **floating-point** operations on PA-RISC systems. Also called the floating-point coprocessor.

floor The floor of a value is the greatest whole number less than that value. *See also* **ceiling**.

flush-to-zero mode On some systems, a method of handling **underflow condition**s in which the hardware simply substitutes a **zero** for the result of an operation that underflows, with no fault occurring. (Normally, an underflow involves a fault into the kernel, where the IEEE-754-specified **conversion** of the result into a **denormalized value** or zero is accomplished by software emulation.)

FMA Fused Multiply-Add, a kind of instruction that combines a multiplication and an addition into a single operation. Also called FMAC (floating-point multiply accumulate).

fraction In a **floating-point** representation, the bits that for **normalized value**s represent a value between 1.0 and 2.0 that is raised to a power of 2. *See also* **exponent**, **sign bit**.

fraction implicit bit In a **floating-point** representation, the bit in the **fraction** that would represent 1.0. Since this bit would always be set, it is not included in the actual format, but it is implied.

hidden bit *See* **fraction implicit bit**.

IEEE standard The IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985), which defines specifications for representing and manipulating **floating-point** values so that programs written on one IEEE-conforming machine can be moved to another conforming machine with predictable results. The international version of the IEEE

standard is *Binary floating-point arithmetic for microprocessor systems* (IEC 559:1989).

ill-conditioned Of or pertaining to a computation in which small changes to the input or to the intermediate results cause relatively large changes in the final output.

inexact result condition The **exception condition** that occurs when a **floating-point** operation produces a result that cannot be represented exactly in the specified floating-point format. Because most floating-point operations produce inexact results most of the time, the inexact result condition is not usually considered to be an error. *See also* **rounding**.

infinity A **floating-point** value that is represented by a **sign bit**, a **fraction** that is all zeros, and an **exponent** that is all ones. An infinity has a magnitude greater than any **normalized value**.

instruction cache aliasing The form of cache aliasing that occurs when two routines reside at addresses that have the same low-order bits but different high-order bits. *See also* **cache aliases**, **data cache aliasing**.

invalid operation condition The **exception condition** that occurs whenever the system attempts to perform an operation that has no numerically meaningful interpretation (for example, 0.0/0.0).

least significant word In **double-precision** and **quad-precision floating-point** formats, the 32-bit word in the representation that contains the last portion of the fraction. *See also* **most significant word**.

library A collection of commonly used routines, pre-compiled in object format and ready to be linked to an application.

mask bits *See* **exception trap enable bits**.

math library A **library** that contains routines that perform higher-level mathematical operations. On HP 9000 systems, C math library functions are located in the `libm` math library; Fortran and Pascal intrinsic functions are located in the `libcl` library; and Basic Linear Algebra Subroutine (BLAS) library routines are located in the **BLAS library** (`libblas`).

most significant word In **double-precision** and **quad-precision floating-point** formats, the 32-bit word in the representation that contains the sign bit, the exponent field, and the first part of the fraction. *See also* **least significant word**.

multiplication One of the basic operations defined by the **IEEE standard**.

NaN (Not-a-Number) A **floating-point** value that is represented by a **sign bit**, a **fraction** with at least one bit set to 1, and an **exponent** with all bits set to 1. *See also* **signaling NaN (SNaN)**, **quiet NaN (QNaN)**.

natural alignment The **alignment** that corresponds to the length of the object. For example, the natural alignment of an HP C `int` is 4-byte alignment.

non-aware A version of a **comparison** assertion that behaves the same as the **aware** version, except that if either or both operands is a **quiet NaN (QNaN)**, it also signals an **invalid operation condition** for the <, <=, >, and >= assertions. *See also* **aware**.

normal *See* **normalized value**.

normalization bit *See* **fraction implicit bit**.

normalized value A **floating-point** value that is represented by a **sign bit**, a **fraction**, and an **exponent** whose bits are not all zeros and not all ones. A normalized value has a magnitude greater than any **denormalized value** and less than **infinity**.

numerically unstable *See* **ill-conditioned**.

operand errors *See* **invalid operation condition**.

operation errors *See* **invalid operation condition**.

overflow condition The **exception condition** that occurs when a **floating-point** operation attempts to produce a result whose magnitude, after **rounding**, is greater than the maximum representable value.

performance bottlenecks The sections of code that require the most execution time.

performance tuning The process of refining a program to make it run faster.

precision The number of bits or digits in which a value can be represented. The precision of a value indicates how close a **floating-point** approximation can be to the exact numeric value being represented.

QNaN *See* **quiet NaN (QNaN)**.

quad-precision The HP-UX implementation of the IEEE double-extended precision **floating-point** format, in which a value occupies 128 bits: 1 bit for the sign, 15 bits for the **exponent**, and 112 bits for the **fraction**. *See also* **single-precision**, **double-precision**.

quiet NaN (QNaN) A **NaN (Not-a-Number)** that usually does not generate an exception; instead, it silently propagates unmodified through an operation. On HP 9000 systems, a QNaN has the most significant bit of the **fraction** set to 0. *See also* **signaling NaN (SNaN)**.

range errors Errors generated by **math library** routines when they underflow or overflow. *See also* **domain errors**.

remainder One of the basic operations defined by the **IEEE standard**.

round to nearest The default IEEE **rounding mode**, which specifies that the result of an operation should be the representable value closest to the true value. If two representable values are equally close to the true value, the result is the one whose least significant bit is 0 (that is, whose last digit is even).

round to nearest integral value One of the basic operations defined by the **IEEE standard**.

round toward +INFINITY The IEEE **rounding mode** that specifies that the result of an operation should be the representable value closest to positive **infinity** (that is, the algebraically greater value).

round toward –INFINITY The IEEE **rounding mode** that specifies that the result of an operation should be the

representable value closest to negative **infinity** (that is, the algebraically lesser value).

round toward zero The IEEE **rounding mode** that specifies that the result of an operation should be the representable value closest to **zero** (that is, the value with the smaller magnitude).

rounding The act of choosing a representable value when the exact value produced by a **floating-point** operation is not representable. The **IEEE standard** specifies four methods of rounding, called **rounding mode**s. *See also* **rounding error**.

rounding error The error that occurs when the result of an operation is rounded to the nearest representable value using an algorithm specified by the **rounding mode**.

rounding mode One of four **rounding** methods specified by the **IEEE standard**: **round to nearest** (the default), **round toward +INFINITY**, **round toward –INFINITY**, and **round toward zero**.

shared library A collection of object modules. When the linker scans a shared library, it does not copy modules into the application's code section, as it does with an **archive library**. Instead, the linker preserves information in the application's code section about which unresolved references were resolved in each shared library. At run time, the shared library is mapped into memory.

sign bit In a **floating-point** representation, the bit that indicates the sign of the value. In IEEE formats, the sign bit is the leftmost bit. *See also* **exponent**, **fraction**.

signal handler *See* **trap handler**.

signaling NaN (SNaN) A **NaN (Not-a-Number)** that generates an **invalid operation condition** whenever it is used. On HP 9000 systems, an SNaN has the most significant bit of the **fraction** set to 1. *See also* **quiet NaN (QNaN)**.

significand *See* **fraction**.

single-precision An IEEE **floating-point** format in which the value occupies 32 bits: 1 bit for the sign, 8 bits for the **exponent**,

and 23 bits for the **fraction**. *See also* **double-precision**, **quad-precision**.

SNaN *See* **signaling NaN (SNaN)**.

square root One of the basic operations defined by the **IEEE standard**.

static variable A variable that retains its value between invocations of the routine in which it is declared. From a performance standpoint, static variables are costly because they prohibit the compiler from making certain types of optimizations.

sticky bits *See* **exception flags**.

subnormal *See* **denormalized value**.

subtraction One of the basic operations defined by the **IEEE standard**.

sudden underflow mode *See* **flush-to-zero mode**.

tiling Rearranging the implementation of your algorithms to process as much data as possible while the data is resident in the cache so as to minimize the number of times the data must be re-cached later. *See also* **cache aliases**.

Translation Lookaside Buffer (TLB) A hardware unit that serves as a cache for virtual-to-absolute memory address mapping.

trap A change in a program's flow of control due to the occurrence of an **exception condition**.

trap handler A routine that is invoked whenever a particular **exception condition** is detected, if the trap is enabled.

ULP (Unit in the Last Place) The rightmost bit of a **floating-point** representation. ULPs measure the distance between two numbers in terms of their representation in binary. One ULP is the distance from one value to the next representable value in the direction away from 0. *See also* **precision**.

underflow condition The **exception condition** that occurs when a **floating-point** operation attempts to produce a result that may suffer extraordiary loss of

accuracy because it is smaller in magnitude than the smallest **normalized value**.

unordered In **comparison** operations, the relation that exists between two operands if one or both of them is a **NaN (Not-a-Number)**; one is neither less than, equal to, nor greater than the other.

vectorization The replacement of a section of code that contains operations on arrays with a call to a special library routine. The compiler attempts automatic vectorization when invoked with appropriate options.

zero A **floating-point** value that is represented by a **sign bit**, a **fraction** that is all zeros, and an **exponent** that is all zeros. A zero has a magnitude less than any **denormalized value**.

Index

Index

Index

Index

Index

Index