OS/390

OS/390 C/C++ Run-Time Library Reference
IEEE Floating-Point Supplement

OS/390

OS/390 C/C++ Run-Time Library Reference
IEEE Floating-Point Supplement

Contents

C/C++ Run-Time Library Reference

New C/C++ Run-Time Library Functions (IEEE Floating-Point)

This section contains the new IEEE floating-point functions that will go in the C/C++ Run-Time Library Reference.

copysign() — Copy Sign

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

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

double copysign (x, y)
double x, y;
```

General Description

The copysign() function returns the *x* parameter with the same sign as the *y* parameter.

Parameters	Description
x	Specifies a long double floating-point value
y	Specifies a long double floating-point value

Returned Value

Long double floating-point value.

Related Information

- nextafter()
- scalb()
- logb()
- ilogb()

finite() — Determine the Infinity Classification of a Floating-Point Number

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

```
#include <math.h>

int finite(x)
double x;
```

General Description

The finite() function determines the infinity classification of floating-point number *x*.

Returned Value

The finite() function returns a nonzero value if the *x* parameter is a finite number,i.e. if *x* is not +–, INF, NaNQ, or NaNS.

The finite() function does not return errors or set bits in the floating-point exception status, even if a parameter is a NaNS.

Related Information

- IEEE Standard for Binary Floating-Point Arithmatic (ANSI/IEEE Standards 754-1985 and 854-1987).

fp_clr_flag() — Reset Floating-Point Exception Status Flag

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

```
#include <float.h>
#include <fpxcp.h>

void fp_clr_flag(mask)
fp_flag_t mask;
```

General Description

The fp_clr_flag() function resets the exception status flags defined by the *mask* parameter to 0 (false). The remaining flags in the exception status remain unchanged.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

FP_INVALID Invalid operation summary

FP_OVERFLOW Overflow

FP_UNDERFLOW Underflow

FP_DIV_BY_ZERO Division by 0

FP_INEXACT Inexact result

Users can reset multiple exception flags using the fp_clr_flag() function by ORing the names of individual flags. For example, the following resets both the overflow and inexact flags.

```
fp_clr_flag(FP_OVERFLOW   FP_INEXACT)
```

Returned Value

None

Related Information

- fp_raise_xcp()
- fp_read_flag()

fp_raise_xcp() — Raise a Floating-Point Exception

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

```
#include <fpxcp.h>

int fp_raise_xcp(mask)
fpflag_t mask;
```

General Description

The fp_raise_xcp() function causes floating-point exceptions defined by the *mask* parameter to be raised immediately.

If the exceptions defined by the *mask* parameter are enabled and the program is running in serial mode, the signal for floating-point exceptions, SIGFPE, is raised.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

FP_INVALID Invalid operation summary

FP_OVERFLOW Overflow

FP_UNDERFLOW Underflow

FP_DIV_BY_ZERO Division by 0

FP_INEXACT Inexact result

Users can cause multiple exceptions using fp_raise_xcp() by ORing the names of individual flags. For example, the following causes both overflow and division by 0 exceptions to occur.

```
fp_raise_xcp(FP_OVERFLOW   FP_DIV_BY_ZERO)
```

If more than one exception is included in the mask variable, the exceptions are raised in the following order:

1. Invalid operation
2. Division by zero
3. Underflow
4. Overflow
5. Inexact result

Thus, if the user exception handler does not disable further exceptions, one call to the fp_raise_xcp() function can cause the exception handler to be entered many times.

Returned Value

The fp_raise_xcp() function returns 0 for normal completion and returns a nonzero value if an error occurs.

Related Information

- fp_clr_flag()
- fp_read_flag()

fp_read_flag() — Returns the Current Floating-Point Exception Status

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

```
#include <float.h>
#include <fpxcp.h>

fp_flag_t fp_read_flag()
```

General Description

The fp_read_flag() function returns the current floating-point exception status.

These functions aid in determining both when an exception has occurred and the exception type. These functions can be called explicitly around blocks of code that may cause a floating-point exception.

According to the IEEE Standard for Binary Floating-Point Arithmetic, the following types of floating-point operations must be signaled when detected in a floating-point operation:

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

An invalid operation occurs when the result cannot be represented (for example, a sqrt operation on a number less than 0).

The IEEE Standard for Binary Floating-Point Arithmetic states: “For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time.”

Floating-point operations can set flags in the floating-point exception status but cannot clear them. Users can clear a flag in the floating-point exception status using an explicit software action such as the fp_clr_flag (0) subroutine.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

FP_INVALID Invalid operation summary

FP_OVERFLOW Overflow

FP_UNDERFLOW Underflow

FP_DIV_BY_ZERO
Division by 0

FP_INEXACT Inexact result

Returned Value

The fp_read_flag() function returns the current floating-point exception status. The flags in the returned exception status can be tested using the flag definitions above. You can test individual flags or sets of flags.

Related Information

- fp_clr_flag()
- fp_raise_xcp()

fp_read_rnd() — Determine the Current Rounding Mode

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

```
#define _AIX_COMPATIBILITY 1
#include <float.h>

fprnd_t fp_read_rnd;
```

General Description

The fp_read_rnd() function returns the current IEEE floating-point rounding mode. Float.h defines the following macros for values indicating floating-point rounding mode:

FP_RND_RZ Round toward zero

FP_RND_RN Round to nearest (default)

FP_RND_RP Round toward +infinity

FP_RND_RM Round toward –infinity

Returned Value

fp_read_rnd() returns the current floating-point rounding mode.

Related Information

- fp_swap_rnd()

fp_swap_rnd() — Swap the IEEE Floating-Point Rounding Mode

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

```
#define _AIX_COMPATIBILITY 1
#include <float.h>

fprnd_t fp_swap_rnd(RoundMode)
fprnd_t RoundMode
```

General Description

The fp_swap_rnd() function sets the floating-point rounding mode to the mode specified by *RoundMode* and returns the previous rounding mode in effect.

float.h defines the following macros for values indicating floating-point rounding mode.

FP_RND_RZ Round toward zero

FP_RND_RN Round to nearest (default)

FP_RND_RP Round toward +infinity

FP_RND_RM Round toward –infinity

Note: The fdlibm math library and other C-RTL functions such as the printf family of functions only support IEEE floating-point “round to nearest mode”. If the rounding mode of an IEEE floating-point application is different from “round to nearest” when it calls an fdlibm function, the rounding mode is changed by the C-RTL to be “round to nearest”. The C-RTL saves the application's rounding mode, changes to “round to nearest” while executing fdlibm code, and then restores the applications rounding mode before returning. If a data exception occurs while executing in fdlibm, any signal handler registered application to handle the exception will be run in the application's rounding mode.

Returned Value

The previous (changed from) rounding mode.

Related Information

- fp_read_rnd()

__isBFP() — Determine Floating-Point Format

Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

Format

```
#include <_Ieee754.h>
int __isBFP(void)
```

General Description

The __isBFP() function determines the mode of floating-point operations.

Returned Value

__isBFP() returns 0 if the floating-point mode is hexadecimal and returns 1 if the floating-point mode is IEEE.

Changed C/C++ Run-Time Library Functions (IEEE Floating-Point)

This section contains changed C/C++ Run-Time Library functions (IEEE Floating-Point)

atof() — Convert Character String to Double

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <stdlib.h>

double atof(const char *nptr);
```

General Description

The atof() function converts the initial portion of the string pointed to by *nptr* to a 'double'. This is equivalent to

```
strtod(nptr, NULL)
```

The double value is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking the atof() function. This function uses __isBFP() to determine the floating-point mode of the invoking thread.

See the "*fscanf* Family of Formatted Input Functions" on page 35 for a description of special infinity and NaN sequences recognized by OS/390 formatted input functions, including atof() and strtod() in IEEE floating-point mode.

Returned Value

There are no documented errnos for this function.

Related Information

- "stdlib.h"
- "atoi() — Convert Character String to Integer"
- "atol() — Convert Character String to Long"
- "fscanf() - scanf() - sscanf() — Read and Format Data"
- "strtod() — Convert Character String to Double"
-
- "strtol() — Convert Character String to Long"
-
- "strtoul() — Convert String to Unsigned Integer"

difftime() — Compute Time Difference

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

Format

```
#include <time.h>

double difftime(time_t time2, time_t time1);
```

General Description

Computes the difference in seconds between *time2* and *time1*, which are calendar times returned by time().

The difftime() function returns the difference between two calendar times as a double. The return value is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking difftime(). The difftime() function uses __isBFP() to determine which floating-point format (hexadecimal floating-point or IEEE floating-point) to return on the invoking thread.

Returned Value

Returns the elapsed time in seconds from *time1* to *time2* as a double.

Example

CBC3BD04

```
/* CBC3BD04
   This example shows a timing application using difftime().  The example
   calculates how long, on average, it takes a user to input some data to the program.
 */
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t start, finish;
   int i, n, num;
   int answer;

   printf("11 x 55 = ? Enter your answer below\n");
   time(&start);
   scanf("%d",&answer);
   time(&finish);
   printf("You answered %s in %.0f seconds.\n",
           answer == 605 ? "correctly" : "incorrectly",
           difftime(finish,start));
}
```

Output

```
 11 x 55 = ? Enter your answer below
 605
 You answered correctly in 20 seconds
```

Related Information

- “time.h”
- “asctime() — Convert Time to a Character String”
- “ctime() — Convert Time to a Character String”
- “gmtime() — Convert Time to Broken-Down UTC Time”
- “localtime() — Convert Time and Correct for Local Time”
- “mktime() — Convert Local Time”
- “strftime() — Convert to Formatted Time”
- “time() — Determine Current Time”

drand48() — Pseudo-random Number Generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

double drand48(void);
```

General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0). These functions have been extended to determine floating-point format (hexadecimal floating-point or IEEE floating-point) of the returned value using the __isBFP()function.

The functions lrand48() and nrand48() return non-negative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The drand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0) = 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The value, X(n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function. Likewise, the values of a and c, may be changed by calling the lcong48() function. Thereafter, whenever the seed48() or srand48() function is called to change X(n), the initial values of a and c are also reestablished.

Special Behavior for OS/390 UNIX Services

You can make the drand48() function and other functions in the drand48 family thread specific by setting the environment variable _RAND48 to the value `THREAD` before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested, and the drand48() function is called from thread t, the drand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread t. The sequence of values for a thread is generated according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

The initial values of X(t), a(t) and c(t) for the thread t are:

```
X(t,0) = 1
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

C/370 provides storage which is specific to the thread t to save the most recent 48-bit integer value of the sequence, X(t,i), generated by the drand48(), lrand48() or mrand48() function. The value, X(t,n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function from the thread t. Likewise, the values of a(t) and c(t) for thread t may be changed by calling the lcong48() function from the thread. Thereafter, whenever the seed48() or srand48() function is called from the thread t to change X(t,n), the initial values of a(t) and c(t) are also reestablished.

Returned Value

The drand48() function transforms the generated 48-bit value, X(n+1), to a double-precision, floating-point value on the interval [0.0,1.0) and returns this transformed value.

Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the drand48 family and the drand48() function is called on thread t, the drand48() function transforms the generated 48-bit value, X(t,n+1), to a double-precision, floating-point value on the interval [0.0,1.0) and returns this transformed value.

Related Information

- “stdlib.h”
- “erand48() — Pseudo-random Number Generator”
- “jrand48() — Pseudo-random Number Generator”
- “lcong48() — Pseudo-random Number Initializer”
- “lrand48() — Pseudo-random Number Generator”
- “mrand48() — Pseudo-random Number Generator”
- “nrand48() — Pseudo-random Number Generator”
- “seed48() — Pseudo-random Number Initializer”
- “srand48() — Pseudo-random Number Initializer”

ecvt() — Convert Double to String

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *ecvt(double x, int ndigit,
           int *decpt, int *sign);
```

General Description

The ecvt() function converts double floating-point argument values to floating-point output strings. The ecvt() function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of double argument values by using __isBFP().

OS/390 (C/C++) formatted output functions, including the ecvt() function, convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See “*fprintf* Family of of Formatted Output Functions” on page 28 for a description of the special infinity and Nan output sequences.

The ecvt() function converts *x* to a null-terminated string of *ndigit* digits (where *ndigit* is reduced to an unspecified limit determined by the precision of a double) and returns a pointer to the string. The high-order digit is nonzero, unless the value is 0. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored in the integer pointed to by *decpt* (negative means left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the integer pointed to by *sign* is nonzero, otherwise it is 0.

The function returns a pointer to a buffer used only by the calling thread which may be overwritten by subsequent calls to ecvt(), “fcvt() — Convert Double to String” on page 22 and “gcvt() — Convert Double to String” on page 42.

If the converted value is out of range or is not representable, the function returns NULL.

Returned Value

If it succeeds, ecvt() returns the character equivalent of *x* as specified above.

If it is unable to allocate the return buffer, or the conversion fails, ecvt() returns NULL.

Related Information

- “stdlib.h”
- “fcvt() —Convert Double to String”
- “gcvt() —Convert Double to String”

erand48() — Pseudo-random Number Generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

double erand48(unsigned short int x16v[3]);
```

General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0]. These functions have been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of the returned value using the __isBFP() function.

The functions lrand48() and nrand48() return non-negative, long integers, uniformly distributed over the interval [0,2**31].

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31].

The erand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)      n>=0
```

The erand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(i). The erand48() function uses *x16v*[0] for the low order (rightmost) 16 bits, *x16v*[1] for the middle order 16 bits, and *x16v*[2] for the high order 16 bits of this value.

The initial values of a, and c are:

```
a   = 5deece66d (base 16)
c   = b         (base 16)
```

The values a and c, may be changed by calling the lcong48() function. The initial values of a and c are restored if either the seed48() or srand48() function is called.

Special Behavior for OS/390 UNIX Services

You can make the erand48() function and other functions in the drand48 family thread specific by setting the environment variable _RAND48 to the value `THREAD` before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested and the erand48() function is called from thread `t`, the erand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

The erand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(t,i). The erand48() function uses *x16v*[0] for the low order (rightmost) 16 bits, *x16v*[1] for the middle order 16 bits, and *x16v*[2] for the high order 16 bits of this value.

The initial values of a(t) and c(t) on the thread `t` are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

The values a(t) and c(t) may be changed by calling the lcong48() function from the thread `t`. The initial values of a(t) and c(t) are restored if either the seed48() or srand48() function is called from the thread.

Returned Value

The erand48() function saves the generated 48-bit value, X(n+1), in storage provided by the argument array, *x16v*[3]. The erand48() function transforms the generated 48-bit value to a double-precision, floating-point value on the interval [0.0,1.0] and returns this transformed value.

Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the drand48 family and the erand48() function is called on thread `t`, the erand48() function saves the generated 48-bit value, X(t,n+1), in storage provided by the argument array, *x16v*[3]. The erand48() function transforms the generated 48-bit value to a double-precision, floating-point value on the interval [0.0,1.0] and returns this transformed value.

Related Information

- “stdlib.h”
- “drand48() — Pseudo-random Number Generator”
- “jrand48() — Pseudo-random Number Generator”
- “lcong48() — Pseudo-random Number Initializer”
- “lrand48() — Pseudo-random Number Generator”
- “mrand48() — Pseudo-random Number Generator”
- “nrand48() — Pseudo-random Number Generator”
- “seed48() — Pseudo-random Number Initializer”
- “srand48() — Pseudo-random Number Initializer”

fcvt() — Convert Double to String

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *fcvt(double x, int ndigit,
           int *decpt, int *sign);
```

General Description

The fcvt() function converts double floating-point argument values to floating-point output strings. The fcvt() function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of double argument values by using __isBFP().

OS/390 (C/C++) formatted output functions, including the fcvt() function, convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See “*fprintf* Family of of Formatted Output Functions” on page 28 for a description of the special infinity and Nan output sequences.

The fcvt() function converts *x* to a null-terminated string which has *ndigit* digits to the right of the radix point (where the total number of digits in the output string is restricted by the precision of a double) and returns a pointer to the string. The function behaves identically to “ecvt() —Convert Double to String” in all respects other than the number of digits in the return value.

Returned Value

If it succeeds, fcvt() returns the character equivalent of *x* as specified above.

If it is unable to allocate the return buffer, or the conversion fails, fcvt() returns NULL.

Related Information

- “stdlib.h”
- “ecvt() — Convert Double to String”
- “gcvt() — Convert Double to String”

fprintf() - printf() - sprintf() — Format and Write Data

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <stdio.h>

int fprintf(FILE *stream, const char *format-string, ...);
int printf(const char *format-string, ...);
int sprintf(char *buffer, const char *format-string, ...);
```

General Description

These three related functions are referred to as the *fprintf family*.

The fprintf() function formats and writes output to a *stream*. It converts each entry in the *argument list*, if any, and writes to the stream according to the corresponding format specification in the *format-string*. The fprintf() function cannot be used with a file that is opened using `type=record`.

The printf() function formats and writes output to the standard output stream `stdout`. printf() cannot be used if `stdout` has been reopened using `type=record`.

The sprintf() function formats and stores a series of characters and values in the array pointed to by *buffer*. Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string*. If the strings pointed to by *buffer* and *format* overlap, behavior is undefined.

fprintf() and printf() have the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

The *format-string* consists of ordinary characters, escape sequences, and conversion specifications. The ordinary characters are copied in order of their appearance. Conversion specifications, beginning with a percent sign (%) or the sequence (%n$) where n is a decimal integer in the range [1,NL_ARGMAX], determine the output format for any *argument-list* following the *format-string*. The *format-string* can contain multibyte characters beginning and ending in the initial shift state.

Special Behavior for XPG4

- If the %n$ conversion specification is found, the value of the nth *argument* after the *format-string* is converted and output according to the conversion specification. Numbered arguments in the argument list can be referenced from *format-string* as many times as required.
- The *format-string* can contain either form of the conversion specification, that is, % or %n$ but the two forms cannot be mixed within a single *format-string* except that %% can be mixed with the %n$ form. When numbered conversion specifications are used, specifying the 'nth' argument requires that the first to (n–1)th arguments are specified in the *format-string*.

The *format-string* is read from left to right. When the first format specification is found, the value of the first *argument* after the *format-string* is converted and output according to the format specification. The second format specification causes the second *argument* after the *format-string* to be converted and output, and so on through the end of the *format-string*. If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. The format specification is illustrated below.

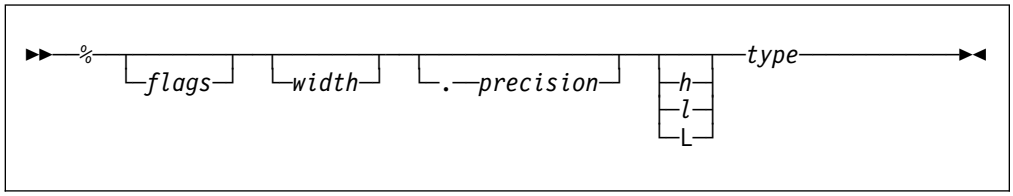


Figure 1. Format Specification for fprintf(), printf(), and sprintf()

Each field of the format specification is a single character or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

The percent sign

If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied to `stdout`. For example, to print a percent sign character, use %%.

The flag characters

The *flag* characters in Table 1 are used for the justification of output and printing of thousands' grouping characters, signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for `wchar_t` precision unit. Notice that more than one *flag* can appear in a format specification. This is an optional field.

Table 1 (Page 1 of 2). Flag Characters for fprintf() Family

Flag	Meaning	Default
'	**Added for XPG4:**The integer portion of the result of a decimal conversion(%i,%d,%u, %f,%g or %G) will be formatted with the thousands' grouping characters.	No grouping.
–	Left-justify the result within the field width.	Right-justify.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
blank(' ')	Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the *blank* flag if both appear, and a positive signed value will be output with a sign.	No blank.

Table 1 (Page 2 of 2). Flag Characters for fprintf() Family

Flag	Meaning	Default
#	When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No prefix.
	When used with the f, e, or E formats, the # flag forces the output value to contain a decimal point in all cases. The decimal point is sensitive to the LC_NUMERIC category of the same current locale.	Decimal point appears only if digits follow it.
	When used with the g or G formats, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it; trailing zeros are truncated.
	When used with the ls or S format, the # flag causes precision to be measured in wide characters.	Precision indicates the maximum number of bytes to be output.
0	When used with the d, i, o, u, x, X, e, E, f, g, or G formats, the 0 flag causes leading 0's to pad the output to the field width. The 0 flag is ignored if precision is specified for an integer or if the – flag is specified.	Space padding.

The code point for the # character varies between the EBCDIC encoded character sets. The definition of the # character is based on the current LC_SYNTAX category. The default C locale expects the # character to use the code point for encoded character set IBM-1047.

When the LC_SYNTAX category is set using setlocale(), the format strings passed to the printf() functions must use the same encoded character set as is specified for the LC_SYNTAX category.

The # flag should not be used with c, lc, C, d, i, u, s, or p types.

The Width of the Output

Width is a non-negative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width*, blanks are added on the left or the right (depending on whether the – flag is specified) until the minimum width is reached.

Width never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are output (subject to the *precision* specification).

The *width* specification can be an asterisk (*); if it is, an argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. This is an optional field.

If *format-string* contains the %n$ form of conversion specification, *width* can be indicated by the sequence *m$, where m is a decimal integer in the range

[1,NL_ARGMAX] giving the position of an integer argument in the argument list containing the field width.

The Precision of the Output

precision is a non-negative decimal integer preceded by a period. It specifies the number of characters to be output, or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value or rounding of a floating-point value.

The *precision* specification can be an asterisk (*); if it is, an argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list. The *precision* field is optional.

If *format-string* contains the %n$ form of conversion specification, *precision* can be indicated by the sequence *m$, where m is a decimal integer in the range [1,NL_ARGMAX] giving the position of an integer argument in the argument list containing the field precision.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type*, as shown in Table 2.

Table 2. Precision Argument in fprintf() family

Type	Meaning	Default
i d u o x X	*Precision* specifies the minimum number of digits to be output. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	If *precision* is 0 or omitted entirely, or if the period (.) appears without a number following it, the *precision* is set to 1.
f e E	*Precision* specifies the number of digits to be output after the decimal point. The last digit output is rounded. The decimal point is sensitive to the LC_NUMERIC category of the current locale.	Default *precision* is 6. If *precision* is 0 or the period appears without a number following it, no decimal point is output.
g G	*Precision* specifies the maximum number of significant digits output.	All significant digits are output.
c	No effect.	The character is output.
C lc	No effect.	The wide character is output.
s	*Precision* specifies the maximum number of characters to be output. Characters in excess of *precision* are not output.	Characters are output until a null character is encountered.
S ls	*Precision* specifies the maximum number of bytes to be output. Bytes in excess of *precision* are not output; however, multibyte integrity is always preserved.	`wchar_t` characters are output until a null character is encountered.

Optional prefix

Used to indicate the size of the argument expected:

`h` A prefix with the integer types `d`, `i`, `o`, `u`, `x`, `X`, and `n` that specifies that the argument is `short int` or `unsigned short int`.

`l` A prefix with `d`, `i`, `o`, `u`, `x`, `X`, and `n` types that specifies that the argument is a `long int` or `unsigned long int`.

The `l` prefix with the `c` type conversion specifier indicates that the argument is a wint_t. The `l` prefix with the `s` type conversion specifier indicates that the argument is a pointer to a wchar_t.

`L` A prefix with `e`, `E`, `f`, `g`, or `G` types that specifies that the argument is `long double`.

Note: If you pass a `long double` value and do not use the L qualifier or if you pass a `double` value only and use the L qualifier, errors occur.

Table 3 below shows the meaning of the type characters used in the precision argument.

Table 3 (Page 1 of 2). Type Characters and their Meanings

Type	Argument	Output Format
d, i	Integer	Signed decimal integer.
u	Integer	Unsigned decimal integer.
o	Integer	Unsigned octal integer.
x	Integer	Unsigned hexadecimal integer, using abcdef.
X	Integer	Unsigned hexadecimal integer, using ABCDEF.
f	Double	Floating-point signed value having the form [–]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number. The number of digits after the decimal point is equal to the requested precision. The decimal point is sensitive to the LC_NUMERIC category of the current locale.
e	Double	Floating-point signed value having the form [–]*d.dddd*e[*sig n*]*ddd*, where *d* is a single-decimal digit, *dddd* is one or more decimal digits, *ddd* is 2 or more decimal digits, and *sign* is + or –.
E	Double	Identical to the *e* format, except that `E` introduces the exponent, not `e`.
g	Double	Floating-point signed value output in `f` or `e` format. The *e* format is used only when the exponent of the value is less than –4 or greater than or equal to the *precision*. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	Double	Identical to the `g` format, except that `E` introduces the exponent (where appropriate), not *e*.
D(n,p)	Decimal type argument.	Fixed–point value consisting of a series of one or more decimal digits possibly containing a decimal point.
c	Character	Single character.
C or lc	Wide Character	The argument of wchar_t type is converted to an array of bytes representing a multibyte character as if by call to wctomb().

Table 3 (Page 2 of 2). Type Characters and their Meanings

Type	Argument	Output Format
s	String	Characters output up to the first null character (\0) or until *precision* is reached.
S or ls	Wide String	The argument is a pointer to an array of wchar_t type. Wide characters from the array are converted to multibyte characters up to and including a terminating null wide character. Conversion takes place as if by a call to wcstombs(), with the conversion state described by the mbstate_t object initialized to 0. The result written out will not include the terminating null character. If no precision is specified, the array contains a null wide character. If a precision is specified, its sets the maximum number of characters written, including shift sequences. A partial multibyte character cannot be written.
n	Pointer to integer	Number of characters successfully output so far to the *stream* or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer to void converted to a sequence of printable characters. Refer to the individual system reference guides for the specific format.

fprintf Family of Formatted Output Functions

fprintf family functions match e, E, f, g or G conversion specifiers to floating-point arguments for which they produce floating-point number substrings in the output stream. *fprintf* family functions have been extended to determine the floating-point format, hexadecimal floating-point or IEEE floating-point, of types e, E, f, g or G by using __isBFP().

fprintf family functions convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences.

- The special output sequence for infinity values is a plus or minus sign, then the character sequence INF followed by a a white-space character (space, tab, or new line), a null character (\0) or EOF.
- The special output sequence for NaN values is a plus or minus sign, then the character sequence NANS for a signalling NaN or NANQ for a quiet NaN, then a NaN ordinal sequence, and then a a white-space character (space, tab, or new line), a null character (\0) or EOF.

  A NaN ordinal sequence is a left-parenthesis character, “(”, followed by a digit sequence representing an integer n, where 1 <= n <= INT_MAX–1, followed by a right-parenthesis character, “)”. The integer value, n, is determined by the fraction bits of the NaN argument value as follows:

  1. For a signalling NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an even integer value, 2*n. Then formatted output functions produce a (signalling) NaN ordinal sequence corresponding to the integer value n.
  2. For a quiet NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an odd integer value, 2*n–1. Then formatted output functions produce a (quiet) NaN ordinal sequence corresponding to the integer value n.

Some compatibility with NaN sequences output by AIX formatted output functions can be achieved by setting a new environment variable,

| _AIX_NAN_COMPATIBILITY, which OS/390 formatted output functions recognize,
| to one of the following (string) values:

| **Value** | **Output Function**

| 1 | Formatted output functions which produce special NaN output
| sequences omit the NaN ordinal output sequence (1). This results in
| output NaN sequences of plus or minus sign followed by NANS or
| NANQ instead of plus or minus sign followed by NANS(1) or NANQ(1).
| All other NaN ordinal sequences are explicitly output.

| ALL | Formatted output functions which produce special NaN output
| sequences omit the NaN ordinal output sequence for all NaN values.
| This results in output NaN sequences of plus or minus sign followed by
| NANS or NANQ instead of plus or minus sign followed by NANS(n) or
| NANQ(n) for all NaN values.

The sprintf() function is available to C applications in a stand-alone Systems Programming Environment.

Returned Value

The fprintf(), printf(), and sprintf() functions return the number of characters output, or a negative value if an output error occurs. The ending null character is not counted.

Example

CBC3BF30

```
/* CBC3BF30
   This example prints data using printf() in a variety of formats.
 */
#include <stdio.h>

int main(void)
{
   char ch = 'h', *string = "computer";
   int count = 234, hex = 0x10, oct = 010, dec = 10;
   double fp = 251.7366;
   unsigned int a = 12;
   float b = 123.45;
   int c;
   void *d = "a";

   printf("the unsigned int is %u\n\n",a);

   printf("the float number is %g, and %G\n\n",b,b);

   printf("RAY%n\n\n",&c);

   printf("last line prints %d characters\n\n",c);

   printf("Address of d is %p\n\n",d);

   printf("%d   %+d    %06d     %X    %x     %o\n\n",
      count, count, count, count, count, count);

   printf("1234567890123%n4567890123456789\n\n", &count);

   printf("Value of count should be 13; count = %d\n\n", count);

   printf("%10c%5c\n\n", ch, ch);
```

```
   printf("%25s\n%25.4s\n\n", string, string);

   printf("%f    %.2f    %e    %E\n\n", fp, fp, fp, fp);

   printf("%i    %i     %i\n\n", hex, oct, dec);
}
```

Output

```
the unsigned int is 12

the float number is 123.45 and 123.45

RAY

last line prints 3 characters

Address of d is DD72F9

234   +234    000234     EA    ea     352

12345678901234567890123456789

Value of count should be 13; count = 13

         h    h

                 computer
                     comp

251.736600    251.74    2.517366e+02    2.517366E+02

16    8     10
```

CBC3BF31

```
/* CBC3BF31
   The following example illustrates the use of printf() to print
   fixed-point decimal data types.
   This example works under C only, not C++.
 */
#include <stdio.h>
#include <decimal.h>

decimal(10,2) pd01 = -12.34d;
decimal(12,4) pd02 = 12345678.9876d;
decimal(31,10) pd03 = 123456789013579246801.9876543210d;

int main(void) {
  printf("pd01 %%D(10,2)      = %D(10,2)\n", pd01);
  printf("pd02 %%D( 12 , 4 )  = %D( 12 , 4 )\n", pd02);

  printf("pd01 %%010.2D(10,2) = %010.2D(10,2)\n", pd01);
  printf("pd02 %%20.2D(12,4)  = %20.2D(12,4)\n", pd02);
  printf("\n Give strange result if the specified size is wrong!\n");
  printf("pd03 %%D(15,3)      = %D(15,3)\n\n", pd03);
}
```

Output

```
pd01 %D(10,2)      = -12.34
pd02 %D( 12 , 4 )  = 12345678.9876
pd01 %010.2D(10,2) = -000012.34
pd02 %20.2D(12,4)  =           12345678.98

Give strange result if the specified size is wrong!
pd03 %D(15,3)      = -123456789013.579
```

CBC3BF32

```
/* CBC3BF32
   This example illustrates the use of sprintf() to format and print
   various data.
 */
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
   c = 'l';
   i = 35;
   fp = 1.7320508;

   /* Format and print various data */
   j = sprintf(buffer, "%s\n", s);
   j += sprintf(buffer+j, "%c\n", c);
   j += sprintf(buffer+j, "%d\n", i);
   j += sprintf(buffer+j, "%f\n", fp);
   printf("string:\n%s\ncharacter count = %d\n", buffer, j);
}
```

Output

```
string:
Baltimore
l
35
1.732051

character count = 24
```

Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “System Programming Facilities” in the *OS/390 C/C++ Programming Guide*
- “locale.h”
- “stdio.h”
- “wchar.h”
- “fscanf() - scanf() - sscanf() — Read and Format Data”
- “localeconv() — Query Numeric Conventions”
- “setlocale() — Set Locale”
- “wcrtomb() — Convert a Wide Character to a Multibyte Character”

fscanf() – scanf() – sscanf() — Read and Format Data

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <stdio.h>

int fscanf (FILE *stream, const char *format-string, ...);
int scanf(const char *format-string, ...);
int sscanf(const char *buffer, const char *format, ... );
```

General Description

These three related functions are referred to as the *fscanf* family.

Reads data from the current position of the specified *stream* into the locations given by the entries in the argument list, if any. The argument list, if it exists, follows the format string. The fscanf() function cannot be used for a file opened with `type=record`.

The scanf() function reads data from the standard input stream `stdin` into the locations given by each entry in the argument list. The argument list, if it exists, follows the format string. scanf() *cannot* be used if `stdin` has been reopened as a `type=record` file.

The sscanf() function reads data from *buffer* into the locations given by *argument-list*. Reaching the end of the string pointed to by *buffer* is equivalent to fscanf() reaching EOF. If the strings pointed to by *buffer* and *format* overlap, behavior is undefined.

fscanf() and scanf() have the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

For all three functions, each entry in the argument list must be a pointer to a variable of a type that matches the corresponding conversion specification in *format-string*. If the types do not match, the results are undefined.

For all three functions, the *format-string* controls the interpretation of the argument list. The *format-string* can contain multibyte characters beginning and ending in the initial shift state.

The format string pointed to by *format-string* can contain one or more of the following:

- White-space characters, as specified by isspace(), such as blanks and new-line characters. A white-space character causes fscanf(), scanf(), and sscanf() to read, but not to store, all consecutive white-space characters in the input up to the next character that is not white space. One white-space character in *format-string* matches any combination of white-space characters in the input.

- Characters that are not white space, except for the percent sign character (%). A non-white-space character causes fscanf(), scanf(), and sscanf() to read, but not to store, a matching non-white-space character. If the next character in the input stream does not match, the function ends.
- Conversion specifications which are introduced by the percent sign (%) or the sequence (%n$) where n is a decimal integer in the range [1,NL_ARGMAX]. A conversion specification causes fscanf(), scanf(), and sscanf() to read and convert characters in the input into values of a conversion specifier. The value is assigned to an argument in the argument list.

All three functions read *format-string* from left to right. Characters outside of conversion specifications are expected to match the sequence of characters in the input stream; the matched characters in the input stream are scanned but not stored. If a character in the input stream conflicts with *format-string*, the function ends, terminating with a “matching” failure. The conflicting character is left in the input stream as if it had not been read.

When the first conversion specification is found, the value of the first *input field* is converted according to the conversion specification and stored in the location specified by the first entry in the argument list. The second conversion specification converts the second input field and stores it in the second entry in the argument list, and so on through the end of *format-string*.

Special Behavior for XPG4.2

- When the %n$ conversion specification is found, the value of the *input field* is converted according to the conversion specification and stored in the location specified by the nth argument in the argument list. Numbered arguments in the argument list can only be referenced once from *format-string*.
- The *format-string* can contain either form of the conversion specification, that is, % or %n$ but the two forms cannot be mixed within a single *format-string* except that %% or %* can be mixed with the %n$ form.

An *input field* is defined as:

- All characters until a white-space character (space, tab, or new line) is encountered
- All characters until a character is encountered that cannot be converted according to the conversion specification
- All characters until the field *width* is reached.

If there are too many arguments for the conversion specifications, the extra arguments are evaluated but otherwise ignored. The results are undefined if there are not enough arguments for the conversion specifications.

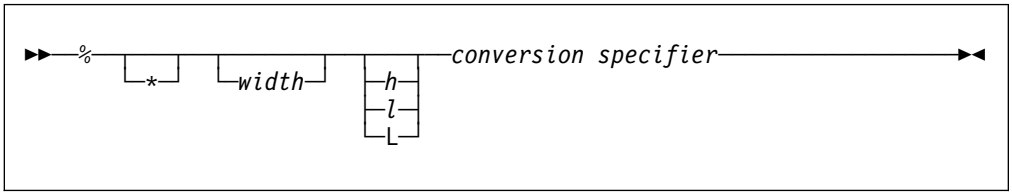


Figure 2. Syntax of Conversion Specification for fscanf(), scanf(), and sscanf()

Each field of the conversion specification is a single character or a number signifying a particular format option. The *conversion specifier*, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest conversion specification contains only the percent sign and a *conversion specifier* (for example, `%s`).

Each field of the format specification is discussed in detail below.

Other than conversion specifiers, you should avoid using the percent sign (`%`), except to specify the percent sign: `%%`. Currently, the percent sign is treated as the start of a conversion specifier. Any unrecognized specifier is treated as an ordinary sequence of characters. If, in the future, OS/390 C/C++ permits a new conversion specifier, it could match a section of your format string, be interpreted incorrectly, and result in undefined behavior. See Table 4 for a list of conversion specifiers.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *conversion specifier*. The field is scanned but not stored.

width is a positive decimal integer controlling the maximum number of characters to be read. No more than *width* characters are converted and stored at the corresponding *argument*.

Fewer than *width* characters are read if a white-space character (space, tab, or new line), or a character that cannot be converted according to the given format occurs before *width* is reached.

The optional prefix `l` shows that you use the `long` version of the following *conversion specifier*, while the prefix `h` indicates that the `short` version is to be used. The corresponding *argument* should point to a `long` or `double` object (for the `l` character), a `long double` object (for the `L` character), or a `short` object (with the `h` character). The `l` and `h` modifiers can be used with the `d`, `i`, `o`, `x`, and `u` *conversion specifiers*. The `l` modifier can also be used with the `e`, `f`, and `g` *conversion specifiers*. The `L` modifier can be used with the `e`, `f` and `g` *conversion specifiers*. Note that the `l` modifier is also used with the `c` and `s` conversion specifiers to indicate a multibyte character or string. The `l` and `h` modifiers are ignored if specified for any other *conversion specifier*.

The *type* characters and their meanings are in Table 4.

Table 4 (Page 1 of 4). Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to `int`
o	Octal integer	Pointer to `unsigned int`
x X	Hexadecimal integer	Pointer to `unsigned int`
i	Decimal, hexadecimal, or octal integer	Pointer to `int`
u	Unsigned decimal integer	Pointer to `unsigned int`

Table 4 (Page 2 of 4). Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
e f g E G	Floating-point value consisting of an optional sign (+ or –); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent (e or E) followed by a possibly signed integer value.	Pointer to `float`

fscanf Family of Formatted Input Functions

fscanf family functions match e, E, f, g or G conversion specifiers to floating-point number substrings in the input stream. *fscanf* family functions convert each input substring matched by an e, E, f, g or G conversion specifier to a `float`, `double` or `long double` value depending on a size modifier preceeding the e, E, f, g or G conversion specifier.

The floating-point value produced is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking the *fscanf* family function. The *fscanf* family functions use __isBFP() to determine the floating-point mode of invoking threads.

Many OS/390 (C/C++) formatted input functions, including the *fscanf* family, recognize special infinity and NaN floating-point number input sequences when the invoking thread is in IEEE floating-point mode as determined by __isBFP().

- The special sequence for infinity input is an optional plus or minus sign, then the character sequence INF, where the individual characters may be upper or lower case, and then a white-space character (space, tab, or new line), a null character (`\0`) or EOF.
- The special sequence for NaN input is an optional plus or minus sign, then the character sequence NANS for a signalling NaN or NANQ for a quiet NaN, where the individual characters may be upper or lower case, then an optional NaN ordinal sequence, and then a a white-space character (space, tab, or new line), a null character (`\0`) or EOF.

  A NaN ordinal sequence is a left-parenthesis character, “(”, followed by a digit sequence representing an integer n, where 1 <= n <= INT_MAX–1, followed by a right-parenthesis character, “)”. If the NaN ordinal sequence is omitted, NaN ordinal sequence (1) is assumed. The integer value, n, corresponding to a NaN ordinal sequence determines what IEEE floating-point NaN fraction bits are produced by formatted input functions.

  For a signalling NaN, these functions produce NaN fraction bits (left to right) by reversing the bits (right to left) of the even integer value 2*n.

  For a quiet NaN they produce NaN fraction bits (left to right) by reversing the bits (right to left) of the odd integer value 2*n–1.

D(n,p)	Fixed–point value consisting of an optional sign (+ or –); a series of one or more decimal digits possibly containing a decimal point.	Pointer to decimal

Table 4 (Page 3 of 4). Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
c	(Can be used with the "l" modifier as `lc`). Character; white-space characters that are ordinarily skipped are read when `c` is specified.	Pointer to `char` large enough for input field.
C or lc	The input matches the number of multibyte characters specified by the field width. Each multibyte character in the sequence is converted to a wide character as if by a call to the mbstowcs() function. The conversion state described by mbstate_t object is initialized to zero before the first multibyte character is converted. The number of wide characters matched is specified by the field width (1 if no field width is present in the directive). The corresponding argument is a pointer to the initial element of an array of `wchar_t` large enough to accept the resulting sequence of wide characters. No null wide character is added. `C` or `lc` is the multibyte character constant.	`C` or `lc` uses a pointer to `wchar_t`.
s S or ls	(Can be used with the "l" modifier as `ls`). String, which matches a sequence of multibyte characters that begins and ends in the initial shift state. None of the multibyte characters in the sequence are also single-byte white-space characters (as specified by the isspace() function). Each multibyte character in the sequence is converted to a wide character as if by a call to the mbrtowc() function, with the conversion state described by mbstate_t object initialized to zero before the first multibyte character is converted. The corresponding argument is a pointer to the initial array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which is added automatically. `S` or `ls` expects a multibyte string constant.	Pointer to character array large enough for input field, plus a terminating null character (`\0`) that is automatically appended. `S` or `ls` uses a pointer to `wchar_t` string.
n	No input read from *stream* or buffer.	Pointer to `int`, into which is stored the number of characters successfully read from the *stream* or buffer up to that point in the call to either fscanf() or to scanf().
p	Pointer to `void` converted to series of characters. For the specific format of the input, see the individual system reference guides.	Pointer to `void`.

Table 4 (Page 4 of 4). Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
[ ]	A non-empty sequence of bytes from a set of expected bytes (the *scanset*), which form the conversion specification. The conversion continues reading bytes until a failure to match or until an input failure. Consider the following situations: [^*bytes*]. In this case, the scanset contains all bytes that do not appear between the circumflex and the right square bracket. []abc] *or* [^]abc.] In both these cases the right square bracket is included in the scanset (in the first case:]abc and in the second case, *not*]abc) [a–z] The – is in the scanset; the characters b through y are *not* in the scanset. The code point for the square brackets ([and]) and the caret (^) vary among the EBCDIC encoded character sets. The default C locale expects these characters to use the code points for encoded character set Latin-1 / Open Systems 1047. Conversion proceeds one byte at a time: there is no conversion to wide characters.	Pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence and a terminating byte, which will be added automatically.

When the LC_SYNTAX category is set using setlocale(), the format strings passed to the fscanf(), scanf(), or sscanf() functions must use the same encoded character set as is specified for the LC_SYNTAX category.

To read strings not delimited by space characters, substitute a set of characters in square brackets ([ ]) for the *s* (string) conversion specifier. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a logical not (¬), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification %*a*c, where *a* is a decimal integer. In this instance, the c conversion specifier means that the argument is a pointer to a character array. The next *a* characters are read from the input stream into the specified location, and no null character is added.

The input for a %*x* conversion specifier is interpreted as a hexadecimal number.

All three functions, fscanf(), scanf(), and sscanf() scan each input field character by character. It might stop reading a particular input field either before it reaches a space character, when the specified *width* is reached, or when the next character cannot be converted as specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

Returned Value

All three functions, fscanf(), scanf(), and sscanf() return the number of input items that were successfully matched and assigned. The returned value does not include conversions that were performed but not assigned (for example, suppressed assignments). The functions return `EOF` if there is an input failure before any conversion, or if EOF is reached before any conversion. Thus a returned value of 0 means that no fields were assigned: there was a matching failure before any conversion. Also, if there is an input failure, then the file error indicator is set, which is not the case for a matching failure.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Examples

CBC3BF42

```
/* This example scans various types of data  */
#include <stdio.h>

int main(void)
{
   int i;
   float fp;
   char c, s[81];

   printf("Enter an integer, a real number, a character "
          "and a string : \n");
   if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
      printf("Not all of the fields were assigned\n");
   else
   {
      printf("integer = %d\n", i);
      printf("real number = %f\n", fp);
      printf("character = %c\n", c);
      printf("string = %s\n",s);
   }
}
```

Output

If input is: `12 2.5 a yes`, then output would be:

```
Enter an integer, a real number, a character and a string:
integer = 12
real number = 2.500000
character = a
string = yes
```

CBC3BF43

```
/* CBC3BF43
   This example converts a hexadecimal integer to a decimal integer.
   The while loop ends if the input value is not a hexadecimal integer.
 */
#include <stdio.h>

int main(void)
{
   int number;
```

```
   printf("Enter a hexadecimal number or anything else to quit:\n");
   while (scanf("%x",&number))
      {
      printf("Hexadecimal Number = %x\n",number);
      printf("Decimal Number     = %d\n",number);
      }
}
```

Output

If input is: `0x231 0xf5e 0x1 q`, then output would be:

```
Enter a hexadecimal number or anything else to quit:
Hexadecimal Number = 231
Decimal Number     = 561
Hexadecimal Number = f5e
Decimal Number     = 3934
Hexadecimal Number = 1
Decimal Number     = 1
```

CBC3BF44

```
/* CBC3BF44
   The next example illustrates the use of scanf() to input fixed-point
   decimal data types. This example works under C only, not C++.
 */
#include <stdio.h>
#include <decimal.h>

decimal(15,4) pd01;
decimal(10,2) pd02;
decimal(5,5) pd03;

int main(void) {
  printf("\nFirst time :-------------------------------\n");
  printf("Enter three fixed-point decimal number\n");
  printf("  (15,4) (10,2) (5,5)\n");
  if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
    printf("Error found in scanf\n");
  } else {
    printf("pd01 = %D(15,4)\n", pd01);
    printf("pd02 = %D(10,2)\n", pd02);
    printf("pd03 = %D(5,5)\n", pd03);
  }
  printf("\nSecond time :-------------------------------\n");
  printf("Enter three fixed-point decimal number\n");
  printf("  (15,4) (10,2) (5,5)\n");
  if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
    printf("Error found in scanf\n");
  } else {
    printf("pd01 = %D(15,4)\n", pd01);
    printf("pd02 = %D(10,2)\n", pd02);
    printf("pd03 = %D(5,5)\n", pd03);
  }
  return(0);
}
```

Output

```
First time :-------------------------------
Enter three fixed-point decimal number
  (15,4) (10,2) (5,5)
12345678901.2345 -987.6 .24680
pd01 = 12345678901.2345
pd02 = -987.60
pd03 = 0.24680

Second time :------------------------------
Enter three fixed-point decimal number
  (15,4) (10,2) (5,5)
123456789013579.24680 123.4567890 987
pd01 = 12345678901.3579
pd02 = 123.45
pd03 = 0.98700
```

CBC3BF46

```
/* CBC3BF46
   The next example opens the file myfile.dat for reading and then scans
   this file for a string, a long integer value, a character, and a
   floating-point value.
 */
#include <stdio.h>
#define  MAX_LEN  80

int main(void)
{
   FILE *stream;
   long l;
   float fp;
   char s[MAX_LEN + 1];
   char c;

   stream = fopen("myfile.dat", "r");

   /* Put in various data. */
   fscanf(stream, "%s", &s[0]);
   fscanf(stream, "%ld", &l);
   fscanf(stream, "%c", &c);
   fscanf(stream, "%f", &fp);

   printf("string = %s\n", s);
   printf("long double = %ld\n", l);
   printf("char = %c\n", c);
   printf("float = %f\n", fp);
}
```

Output

If `myfile.dat` contains `abcdefghijklmnopqrstuvwxyz 343.2`, then the expected output is:

```
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
```

CBC3BS32

```
/* CBC3BS32
   This example uses sscanf() to read various data from the string
   tokenstring, and then displays the data.
 */
#include <stdio.h>
#define  SIZE  81

int main(void)
{
char *tokenstring = "15 12 14";
int i;
float fp;
char s[SIZE];
char c;

   /* Input various data                              */
   printf("No. of conversions=%d\n",
      sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp));

   /* If there were no space between %s and %c,       */
   /* sscanf would read the first character following */
   /* the string, which is a blank space.            */

   /* Display the data */
   printf("string = %s\n",s);
   printf("character = %c\n",c);
   printf("integer = %d\n",i);
   printf("floating-point number = %f\n",fp);
}
```

Output

You would see this output from example CBC3BS32.

```
No. of conversions = 4
string = 15
character = 1
integer = 2
floating-point number = 14.000000
```

Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h”
- “stdio.h”
- “fprintf() - printf() - sprintf() — Format and Write Data”
- “localtime() — Convert Time and Correct for Local Time”
- “setlocale() — Set Locale”

gcvt() — Convert Double to String

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *gcvt(double x, int ndigit,
           char *buf);
```

General Description

The gcvt() function converts double floating-point argument values to floating-point output strings. The gcvt() function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of double argument values by using __isBFP().

OS/390 (C/C++) formatted output functions, including the gcvt() function, convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See “*fprintf* Family of of Formatted Output Functions” on page 28 for a description of the special infinity and Nan output sequences.

The gcvt() function converts *x* to a null-terminated string (similar to the %g format of “fprintf() - printf() - sprintf() — Format and Write Data” on page 23) in the array pointed to by *buf* and returns *buf*. It produces *ndigit* significant digits (limited to an unspecified value determined by the precision of a double) in %f if possible, or %e (scientific notation) otherwise. A minus sign is included in the returned string if *value* is less than 0. A radix character is included in the returned string if *value* is not a whole number. Trailing zeros are suppressed where *value* is not a whole number. The radix character is determined by the current locale. If “setlocale() —Set Locale” has not been called successfully, the default locale, “POSIX”, is used. The default locale specifies a period (.) as the radix character. The LC_NUMERIC category determines the value of the radix character within the current locale.

Returned Value

If it succeeds, gcvt() returns the character equivalent of *x* as specified above.

If the conversion fails, gcvt() returns NULL.

Related Information

- “stdlib.h”
- “ecvt() —Convert Double to String”
- “fcvt() —Convert Double to String”

strfmon() — Convert Monetary Value to String

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#include <monetary.h>

int strfmon(char *s, size_t maxsize, const char *format, ...);
```

General Description

| strfmon() produces a formatted monetary output string from a double argument. It
| has been extended to determine floating-point argument format (hexadecimal
| floating-point or IEEE floating-point) using the __isBFP() function.

| **Note:** In IEEE floating-point mode, denormal, infinity and NaN argument values
| are out of range.

Places characters into the array pointed to by **s* as controlled by the string pointed to by *format*. No more than *maxsize* characters are placed into the array.

The character string *format* contains two types of objects: plain characters, which are copied to the output array, and directives, each of which results in the fetching of zero or more arguments that are converted and formatted. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are simply ignored. If objects pointed to by *s* and *format* overlap, the behavior is undefined.

The directive (conversion specification) consists of the following sequence:

1. A % character
2. Optional flags: =f, ^, !, then +, C, or (
3. Optional field width (may be preceded by w
4. Optional left precision: #n
5. Optional right precision: .p
6. Required conversion character to indicate what conversion should be performed: i or n

Each directive is replaced by the appropriate characters, as described in the following list:

%i The double argument is formatted according to the locale's international currency format (for example, in USA: USD 1,234.56).

%n The double argument is formatted according to the locale's national currency format (for example, in USA: $1,234.56).

%% is replaced by %. No argument is converted.

The following optional conversion specifications may immediately follow the initial % of a directive:

=f
: A flag, used in conjunction with the maximum digits specification *#n* (see below), specifies that the character *f* should be used as the numeric fill character. The default numeric fill character is the space character. This option does not affect the other fill operations that always use space as the fill character.

^
: A flag. Do not format the currency amount with thousands grouping characters. The default is to insert the grouping characters if defined for the current locale.

  Note: The code point for the ^ character will be determined according to the current LC_SYNTAX category.

+ | C | (
: A flag, specifies the style of representing positive and negative currency amounts. Only one of `+`, `C`, or `(` may be specified. If `+` is specified, the locale's equivalent of + and – are used (for example, in USA: the empty (null) string if positive and - if negative). If `C` is specified, the locale's equivalent of DB for negative and CR for positive are used. If `(` is specified, the locale's equivalent of enclosing negative amounts within parentheses is used. If this option is not included, a default specified by the current locale is used.

[–]w
: The field width. The decimal digit string *w* specifies a minimum field width in which the result of the conversion is right-justified (or left-justified if the optional flag "–" is specified).

#n
: The left precision. The decimal digit string *n* specifies the maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the strfmon() aligned in the same columns. It can also be used to fill unused positions with a special character as in $***123.45. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more digit positions are required than the number specified, conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character. (See the `=f` specification above.)

  If the thousands grouping is enabled, the behavior is:

  1. Format the number as if it is an *n* digit number.
  2. Insert fill characters to the left of the leftmost digit (for example, $0001234.56 or $***1234.56)
  3. Insert the separator character (for example, $0,001,234.56 or $*,**1,234.56)
  4. If the fill character is not the digit zero, the separators are replaced by the fill character (for example, $****1,234.56).

  To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with space characters to make their positive and negative formats an equal length.

  Note: The code point for the # character (in `#n`) will be determined according to the current LC_SYNTAX category.

.p
: The right precision. The decimal digit string *p* specifies the number of digits after the radix character. If the value of the precision *p* is zero, no radix character appears. If this option is not included, a default

specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

! A flag used to suppress the currency symbol from the output conversion.

Note: The code point for the ! character is determined according to the current LC_SYNTAX category.

The LC_MONETARY category of the program's locale affects the behavior of this function including the monetary radix character (which is different from the numeric radix character affected by the LC_NUMERIC category), the thousands (or alternative grouping) separator, the currency symbols and formats. The international currency symbol must be in accordance with those specified in ISO 4217 Codes for the representation of currencies and funds.

Returned Value

If the total number of resulting bytes including the terminating null character is not more than *maxsize*, the strfmon() function returns the number of bytes placed into the array pointed to by **s*, not including the terminating null character. Otherwise, -1 is returned , the contents of the array are indeterminate and errno is set to indicate the error.

E2BIG Conversion stopped due to lack of space in the buffer

Example

CBC3BS41

```
/* CBC3BS41 */
#include <localdef.h>
#include <monetary.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char   string[100];     /* hold the string returned from strfmon() */
   double money = 1234.56;

   if (setlocale(LC_ALL, "En_US") == NULL) {
      printf("Unable to setlocale().\n");
      exit(1);
   }

   strfmon(string, 100, "%i", money);
   printf("%s\n", string);
   strfmon(string, 100, "%n", money);
   printf("%s\n", string);
}
```

Related Information

- “monetary.h”

strtod() — Convert Character String to Double

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

Format

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr)
```

General Description

Converts a part of a character string, pointed to by *nptr*, to a double. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type *double*.

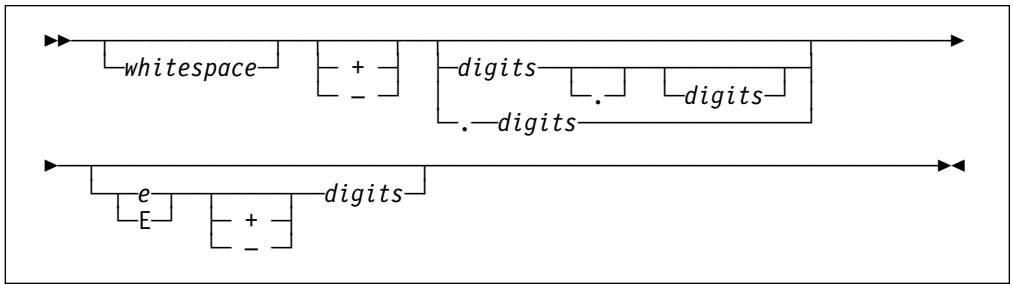
The double value is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking the strtod() function. This function uses __isBFP() to determine the floating-point mode of the invoking thread.

See the "*fscanf* Family of Formatted Input Functions" on page 35 for a description of special infinity and NaN sequences recognized by OS/390 formatted input functions, including atof() and strtod() in IEEE floating-point mode.

The strtod() function breaks the string into three parts:

1. A sequence of white-space characters (as specified for the current locale, see isspace())
2. The sequence of characters of the floating-point constant format (the *subject string*)
3. A sequence of unrecognized characters (including a null character).

The function then attempts to convert the subject string into the floating-point number. The format of the expected string is as follows:



The subject string is the longest string that matches the expected form.

The pointer to the last string successfully converted is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer. If the subject string is

empty or it does not have the expected form, then no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

Returned Value

Returns the value of the floating-point number, except when the representation causes an underflow or overflow. In an overflow, it returns -HUGE_VAL or +HUGE_VAL. In an underflow, it returns the value 0. If no conversion is performed, strtod() returns the value 0. In both cases, errno is set to ERANGE, depending on the base of the value.

Example

CBC3BS53

```
/* CBC3BS53
   This example converts a string to a double value.  It prints out the
   converted value and the substring that stopped the conversion.
 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string, *stopstring;
   double x;

   string = "3.1415926This stopped it";
   x = strtod(string, &stopstring);
   printf("string = %s\n", string);
   printf("   strtod = %f\n", x);
   printf("   Stopped scan at %s\n\n", stopstring)

   string = "100ergs";
   x = strtod(string, &stopstring);
   printf("string = \"%s\"\n", string);
   printf("   strtod = %f\n", x);
   printf("   Stopped scan at \"%s\"\n\n", stopstring);
}
```

Output

```
string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at This stopped it

string = 100ergs
   strtod = 100.000000
   Stopped scan at ergs
```

Related Information

- “stdlib.h”
- “atof() —Convert Character String to Double”
- “atoi() —Convert Character String to Integer”
- “atol() —Convert Character String to Long”
- “fscanf() - scanf() - sscanf() —Read and Format Data”
- “strtol(0) —Convert Character String to Long”
- “strtoul() —Convert String to Unsigned Integer”

wcstod() — Convert Wide-Character String to a Double Floating-Point

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

Format

```
#include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

General Description

The wcstod() function converts a wchar_t * type floating-point number input string to a double value. The double value is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking wcstod(). The wcstod() function uses __isBFP() to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of the invoking thread.

See the "*fscanf* Family of Formatted Input Functions" on page 35 for a description of special infinity and NaN sequences recognized by OS/390 formatted input functions, including wcstod() in IEEE floating-point mode.

Converts the initial portion of the wide-character string pointed to by *nptr* to `double` representation. First it decomposes the input string into three parts:

1. An initial, possibly empty, sequence of white space characters (as specified by the iswspace() function)
2. A subject sequence resembling a floating-point constant.
3. A final string of one or more unrecognized characters, including the terminating NULL character of the input string.

Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a non-empty sequence of digits optionally containing a decimal-point wide character, then an optional exponent part as defined in ISO/IEC 9899: subclause 6.1.3.1, but with no floating suffix. The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space wide characters, or if the first non-white space wide character is other than a sign, a digit, or a decimal-point wide character.

If the subject sequence has the expected form, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of ISO/IEC 9899: subclause 6.1.3.1, except the decimal-point wide character is used in place of a period, and if neither an exponent part nor a decimal-point wide character appears, a decimal point is assumed to follow the last digit in the wide-character string. If the subject sequence begins with a minus sign, the value resulting from the conversion

is negated. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In a locale other than the C locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned Value

Returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus HUGE_VAL is returned—according to the sign of the value—and the value of the macro ERANGE is stored in errno. If the correct value would cause underflow, zero is returned and the value of the macro ERANGE is stored in errno.

Example

CBC3BW21

```
/* CBC3BW21 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs;
   wchar_t *stopwcs;
   double   d;

   wcs = L"3.1415926This stopped it";
   d = wcstod(wcs, &stopwcs);
   printf("wcs =  %ls \n", wcs);
   printf("   wcstod = %f\n", d);
   printf("   Stopped scan at  %ls \n", stopwcs);
}
```

Related Information

- “wchar.h”

Communicating Your Comments to IBM

OS/390
OS/390 C/C++ Run-Time Library Reference
IEEE Floating-Point Supplement

Publication No.

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use this network ID:
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: http://www.s390.ibm.com/os390

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

OS/390
OS/390 C/C++ Run-Time Library Reference
IEEE Floating-Point Supplement

Publication No.

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: _____

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ]	As an introduction	[ ]	As a text (student)
[ ]	As a reference manual	[ ]	As a text (instructor)
[ ]	For another purpose (explain)		

_____

_____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

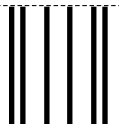
Page Number: Comment:

Name	Address
Company or Organization	
Phone No.	

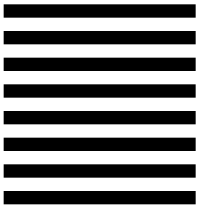
Reader's Comments — We'd Like to Hear from You

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie NY 12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5647-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.