Mathematics (Enhanced) <math.h>

WG14/N512 X3J11/95-113 (Draft 12/21/95)

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
jim_thomas@taligent.com

This is a C9X proposal to merge the FPCE <fp.h> header into <math.h>.

7.x Mathematics <math.h>

The header <math.h> declares two types and several mathematical functions and defines several macros. Integer arithmetic functions and conversion functions are discussed later.

> ["Floating-Point C Extensions" in the X3J11 Numerical C Extensions Technical Report called this header <fp.h>. For integrating the features into the standard, merging them into <math.h> seemed to provide the least troublesome change to the standard.]

The typedefs

```
float_t
double_t
```

are defined to be the implementation's most efficient floating types at least as wide as `float` and `double`, respectively.[1]

> [A facility to use wider types is needed for writing portable efficient code. Previously, Standard C gave no way of asking for the most efficient floating type with at least a given width. Efficiency on different floating-point architectures required different prototypes.
>
> | architecture | most efficient prototype |
> |---|---|
> | extended-based | `long double f(long double)` |
> | double-based | `double f(double)` |
> | single/double | `float f(float)` |
> | single/double/extended | `float f(float)` |
>
> Differences may involve whether values can be kept in registers, hence are substantial. Implementations for the various floating-point architectures might use the following type definitions:

[1] It is intended that `float_t` and `double_t` fit the implementation's (default) expression evaluation method: `float_t` and `double_t` are `float` and `double`, respectively, if `FLT_EVAL_METHOD` equals 0; they are both `double` if `FLT_EVAL_METHOD` equals 1; and they are both `long double` if `FLT_EVAL_METHOD` equals 2. The type `float_t` is the narrowest type used by the implementation to evaluate floating expressions.

95

architecture	`float_t`	`double_t`
extended-based	`long double`	`long double`
double-based	`double`	`double`
single/double	`float`	`double`
single/double/extended	`float`	`double`

An alternate approach would have been to modify the semantics of the `register` storage-class specifier. Applied to a floating type, `register` would have meant the associated value may be wider than the type. This was rejected as inconsistent with existing use of `register` in Standard C.]

The macro

`HUGE_VAL`

expands to a positive `double` expression, not necessarily representable as a `float`. The macros

`HUGE_VALF`
`HUGE_VALL`

are respectively `float` and `long double` analogs of `HUGE_VAL`.[2]

The macro

`INFINITY`

expands to an expression of type `float_t` representing an implementation-defined positive or unsigned infinity, if available, else to a positive constant of type `float_t` that overflows at translation time.

The macro

`NAN`

is defined if and only if the implementation supports quiet NaNs for the `float_t` type. It expands to an expression of type `float_t` representing an implementation-defined quiet NaN.

[Ideally, the `INFINITY` and `NAN` macros would be suitable for static and aggregate initialization, as would similar macros in `<float.h>`. However, this is not required.]

Should macros like INFINITY and NAN be guaranteed suitable for initializations? Given hex constants, could the floating-valued macros in <float.h> be required to be constant expressions?

[2] `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL` can be positive infinities in an implementation that supports infinities.

CONSTANTS @118 96

The macros

```
FP_NAN
FP_INFINITE
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

are for number classification. They represent the mutually exclusive kinds of floating-point values. They expand to integral constant expressions with distinct values.

> [Some prior art uses a finer classification: `FP_POS_INFINITE`, `FP_NEG_INFINITE`, etc. The consensus was that those specified, in conjunction with the `signbit` macro, are generally preferable.]

The macros

```
fp_contract_on
fp_contract_off
fp_contract_default
```

can be used to allow (if the state is *on*) or disallow (if the state is *off*) the implementation to contract expressions (6.3). Each macro can occur either outside external declarations or preceding a compound statement. When outside external declarations, the macro takes effect from its occurrence until another `fp_contract` macro is encountered, or until the end of the translation unit. When preceding a compound statement, the macro takes effect from its occurrence until another `fp_contract` macro is encountered, or until the end of the compound statement. The effect of one of these macros in any other context is undefined. The default state (*on* or *off*) for the macros is implementation-defined.

> [Previous versions of this specifications used pragmas instead of macros for this mechanism. Macros were preferred because of general limitations with pragmas and because of the wish not to require standard pragmas.
>
> The ability to apply the macros to compound statements was added to avoid unnecessary restrictions on optimization and to support locality in source code.]

The macro

```
DECIMAL_DIG
```

expands to an integral constant expression whose value is implementation-defined. It represents a number of decimal digits supported by conversion between decimal and all internal floating-point formats.[3]

> [`DECIMAL_DIG` is distinct from `DBL_DIG`, which is defined in terms of conversion from decimal to `double` and back.
>
> `DECIMAL_DIG` was deemed more useful than `FP_CONV_DIG`, which previous versions of this specification defined as the number of decimal digits for which the implementation guaranteed correctly rounded conversion.]

[3] `DECIMAL_DIG` is intended to give an appropriate number of digits to carry in canonical decimal representations.

Recommended practice

Conversion from (at least) `double` to decimal with `DECIMAL_DIG` digits and back is the identity function.[4]

7.x.1 Classification macros

In the synopses in this subclause, *floating-type* indicates a parameter of the same floating type as the argument. The result is undefined if an argument is not of floating type.

[Requiring the arguments to be of floating type allows efficient implementation.]

7.x.1.1 The `fpclassify` macro

Synopsis

```
#include <math.h>
int fpclassify(floating-type x);
```

Description

The `fpclassify` macro classifies its argument value as NaN, infinite, normal, subnormal, or zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.[5]

Return

The `fpclassify` macro returns the value of the number classification macro appropriate to the value of its argument.

[`fpclassify` might be implemented as

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

]

---

[4] In order that correctly rounded conversion from an internal floating-point format with precision m to decimal with `DECIMAL_DIG` digits and back be the identity function, `DECIMAL_DIG` should be a positive integer n satisfying the inequality:

$n \geq m$ if `FLT_RADIX` is 10

$10^{n-1} > \texttt{FLT\_RADIX}^m$ otherwise

[5] Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal `long double` value might become subnormal when converted to `double`, and zero when converted to `float`.

7.x.1.2 The `signbit` macro

Synopsis

```
#include <math.h>
int signbit(floating-type x);
```

Description

The `signbit` macro determines whether the sign of its argument value is negative.[6]

Return

The `signbit` macro returns a nonzero value if and only if the sign of its argument value is negative.

7.x.1.3 The `isfinite` macro

Synopsis

```
#include <math.h>
int isfinite(floating-type x);
```

Description

The `isfinite` macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Return

The `isfinite` macro returns a nonzero value if and only if its argument has a finite value.

7.x.1.4 The `isnormal` macro

Synopsis

```
#include <math.h>
int isnormal(floating-type x);
```

Description

The `isnormal` macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

---

[6] The `signbit` macro faithfully reports the sign of all values, including infinities, zeros, and NaNs.

Return

The `isnormal` macro returns a nonzero value if and only if its argument has a normal value.

7.x.1.5 The `isnan` macro

Synopsis

```
#include <math.h>
int isnan(floating-type x);
```

Description

The `isnan` macro determines whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.[7]

Return

The `isnan` macro returns a nonzero value if and only if its argument has a NaN value.

7.x.2 Overloading

The functions in `<math.h>` generally have one or more `double` parameters and return `double` or `int` results. Expect where otherwise indicated, each function is additionally implemented as an *overloading macro*[8]. The parameters whose type is `double` in the function synopsis are *overloading parameters*. Use of the macro invokes a function whose type for the overloading parameters is the wider of

— the types of floating arguments for overloading parameters
— the narrowest floating type used for expression evaluation

If the function synopsis indicates a `double` return type, then the return type for the function invoked by the overloading macro matches the type for the overloading parameters.

Examples

1. The synopsis for the square root function shows the prototype

```
double sqrt(double x);
```

Parameter `x` has type `double` and hence is an overloading parameter.

For `FLT_EVAL_METHOD` equal 0, `float` is the narrowest floating type used for expression evaluation. If the argument is `float` or integral then the type deployed by the `sqrt` macro is `float`. For example, the macro might invoke

---

[7] The type for determination doesn't matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.

[8] Like other function-like macros in the Standard libraries, each overloading macro can be suppressed to make available the corresponding ordinary function.

```
float _Sqrtf(float);
```

If the argument is **double** or **long double** then the type deployed by the **sqrt** macro is **double** or **long double**, respectively.

For **FLT_EVAL_METHOD** equal 1, **double** is the narrowest floating type used for expression evaluation. The type deployed by the **sqrt** macro is **long double** for a **long double** argument, and is **double** for other argument types.

For **FLT_EVAL_METHOD** equal 2, **long double** is the narrowest floating type used for expression evaluation. The type deployed by the **sqrt** macro is always **long double**.

2. The synopsis for the **remquo** function shows the prototype

```
double remquo(double x, double y, int *quo);
```

Parameters **x** and **y** have type **double** and hence are the overloading parameters. In the following fragment **remquo** has type **float**, **double**, or **long double**, according as **FLT_EVAL_METHOD** equals 0, 1, or 2, respectively:

```
float a, b, r;
long n;
int q;
/*...*/
r = remquo(n, a * b, &q);
```

Rationale on overloading still needs to be moved over from the TR and updated.

7.x.3 Treatment of error conditions

The behavior of each of the functions in **<math.h>** is defined for all representable values of its input arguments.

For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.[9] On a domain error, the function returns an implementation-defined value; whether the integer expression **errno** acquires the value **EDOM** is implementation-defined.

Similarly, a *range error* occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude. The result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result overflows and default rounding is in effect, or if the mathematical result is an exact infinity (for example **log(0.0)**), then the function returns the value of the macro **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** appropriate to the

[9] In an implementation that supports infinities, this allows an infinity as an argument to be a domain error if the mathematical domain of the function does not include the infinity.

specified result type, with the same sign as the correct value of the function; whether **errno** acquires the value **ERANGE** is implementation-defined. The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type.[10] If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number in the specified type and is otherwise implementation-defined; whether **errno** acquires the value **ERANGE** is implementation-defined.

> [The setting of **errno** is no longer required because doing so is prohibitively expensive on many systems and because many implementations, including all IEEE ones, have more useful means of detecting domain and range errors.]

7.x.4 Trigonometric functions

No changes here (except for renumbering).

7.x.5 Hyperbolic functions

Merge in these subclauses:

7.x.5.1 The **acosh** function

Synopsis

```
#include <math.h>
double acosh(double x);
```

Description

The **acosh** function computes the (nonnegative) arc hyperbolic cosine of **x**. A domain error occurs for arguments less than 1. A range error occurs if **x** is too large.

Returns

The **acosh** function returns the arc hyperbolic cosine in the range [0, +∞].

7.x.5.2 The **asinh** function

Synopsis

```
#include <math.h>
double asinh(double x);
```

Description

The **asinh** function computes the arc hyperbolic sine of **x**. A range error occurs if the magnitude of **x** is too large.

---

[10] The term underflow here is intended to encompass both *gradual underflow* as in ANSI/IEEE 754 (IEC 559) and also *flush-to-zero* underflow.

Returns

The **asinh** function returns the arc hyperbolic sine.

7.x.5.3 The atanh function

Synopsis

```
#include <math.h>
double atanh(double x);
```

Description

The **atanh** function computes the arc hyperbolic tangent of **x**. A domain error occurs for arguments not in the range [-1, +1].

Returns

The **atanh** function returns the arc hyperbolic tangent.

7.x.6 Exponential and logarithmic functions

Merge in these subclauses, replacing the modf subclause:

7.x.6.2 The exp2 function

Synopsis

```
#include <math.h>
double exp2(double x);
```

Description

The **exp2** function computes the base-2 exponential of **x**: 2^{x}. A range error occurs if the magnitude of **x** is too large.

Returns

The **exp2** function returns the base-2 exponential.

7.x.6.3 The expm1 function

Synopsis

```
#include <math.h>
double expm1(double x);
```

Description

The **expm1** function computes the base-e exponential of the argument, minus 1: $e^{x} - 1$. For small magnitude **x**, **expm1(x)** is expected to be more accurate than **exp(x) - 1**. A range error occurs if the **x** is too large.

Returns

The **expm1** function returns $e^x - 1$.

7.x.6.8 The log1p function

Synopsis

```
#include <math.h>
double log1p(double x);
```

Description

The **log1p** function computes the base-e logarithm of 1 plus the argument. For small magnitude **x**, **log1p(x)** is expected to be more accurate than **log(1 + x)**. A domain error occurs if the argument is less than -1. A range error may occur if the argument equals -1.

Returns

The **log1p** function returns the base-e logarithm of 1 plus the argument.

7.x.6.9 The log2 function

Synopsis

```
#include <math.h>
double log2(double x);
```

Description

The **log2** function computes the base-2 logarithm of **x**. A domain error occurs if the argument is less than zero. A range error may occur if the argument is zero.

Returns

The **log2** function returns the base-2 logarithm.

7.x.6.10 The logb function

Synopsis

```
#include <math.h>
double logb(double x);
```

Description

The **logb** function extracts the exponent of **x**, as a signed integral value in the format of **x**. If **x** is subnormal it is treated as though it were normalized; thus for positive finite **x**,

$$1 \le x * \mathrm{FLT\_RADIX}^{-\mathrm{logb}(x)} < \mathrm{FLT\_RADIX}$$

A range error may occur if the argument is zero.

[The treatment of subnormal `x` follows the recommendation in IEEE standard 854, which differs from IEEE standard 754 on this point. Even 754 implementations are intended to follow this definition rather than the one recommended (not required) by 754.

Particularly on machines whose radix is not 2, `logb` can be expected to obtain the exponent more accurately and quickly than `frexp`.]

Returns

The `logb` function returns the signed exponent of its argument.

7.x.6.11 The `modf` functions

Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

Description

The `modf` function breaks the argument `value` into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `double` in the object pointed to by `iptr`. It has no overloading macro.

The `modff` function is similar to the `modf` function, expect that the first parameter and the returned value each has type `float` and the second parameter has type *pointer to* `float`. The `modfl` function is similar to the `modf` function, expect that the first parameter and the returned value each has type `long double` and the second parameter has type *pointer to* `long double`.

Returns

The `modf`, `modff`, and `modfl` functions each returns the signed fractional part of `value`.

7.x.6.12 The `scalb` function

Synopsis

```
#include <math.h>
double scalb(double x, long int n);
```

Description

The `scalb` function computes `x * FLT_RADIX`^{n} efficiently, not normally by computing `FLT_RADIX`^{n} explicitly. A range error may occur.

Returns

The `scalb` function returns `x * FLT_RADIX`^{n}.

[On machines whose radix is not 2, `scalb`, compared with `ldexp`, can be expected to have better accuracy, speed, and overflow and underflow behavior.

The second parameter has type `long int`, unlike the corresponding `int` parameter for `ldexp`, because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width `int` format allowed by Standard C.]

7.x.7 Power and absolute value functions

Rename the "Power functions" subclause as shown above. Move fabs into this subclause. In the first sentence of the description of pow, replace "is not an integral value" with "is finite and not an integral value". Merge in the following subclause:

7.x.7.2 The `hypot` function

Synopsis

```
#include <math.h>
double hypot(double x, double y);
```

Description

The `hypot` function computes the square root of the sum of the squares of `x` and `y`, without undue overflow or underflow. A range error may occur.

Returns

The `hypot` function returns the square root of the sum of the squares of `x` and `y`.

Add the following subclause:

7.x.8 Error and gamma functions

[See [23] regarding implementation.]

7.x.8.1 The `erf` function

Synopsis

```
#include <math.h>
double erf(double x);
```

Description

The `erf` function computes the error function of `x`.

Returns

The `erf` function returns the error function of `x`.

7.x.8.2 The `erfc` function

Synopsis

```
#include <math.h>
double erfc(double x);
```

Description

The `erfc` function computes the complementary error function of `x`.

Returns

The `erfc` function returns the complementary error function of `x`.

7.x.8.3 The `gamma` function

Synopsis

```
#include <math.h>
double gamma(double x);
```

Description

The `gamma` function computes the gamma function of `x`: $\Gamma(x)$. A domain error occurs if `x` is a negative integer or zero. A range error may occur.

Returns

The `gamma` function returns $\Gamma(x)$.

[In UNIX System V [10], both the `gamma` and `lgamma` functions compute $\log(|\Gamma(x)|)$.]

7.x.8.4 The `lgamma` function

Synopsis

```
#include <math.h>
double lgamma(double x);
```

Description

The `lgamma` function computes the logarithm of the absolute value of gamma of `x`: $\log_e(|\Gamma(x)|)$. A range error occurs if `x` is too large.

[In UNIX System V [10], a call to `lgamma` sets an external variable `signgam` to the sign of `gamma(x)`, which is -1 if

```
x < 0 && remainder(floor(x), 2) != 0
```

Note that this specification does not remove the external identifier `signgam` from the user's name space. An implementation that supports, as an extension, `lgamma`'s setting of `signgam` must still protect the external identifier `signgam` if defined by the user.]

Returns

The **lgamma** function returns $\log_e(|\Gamma(x)|)$.

Split up the subclause "Nearest integer, absolute value, and remainder functions" into two subclauses, "Nearest integer functions" which will include ceil and floor and "Remainder functions" which will include fmod. Merge in the subclauses below.

7.x.9 Nearest integer functions

7.x.9.3 The nearbyint function

Synopsis

```
#include <math.h>
double nearbyint(double x);
```

Description

The **nearbyint** function differs from the **rint** function (7.x.9.4) only in that the **nearbyint** function does not raise the inexact exception. (See X.10.6.3-4.)

[For implementations that do not support the inexact exception, **nearbyint** and **rint** are equivalent.]

Returns

The **nearbyint** function returns the rounded integral value.

7.x.9.4 The rint function

Synopsis

```
#include <math.h>
double rint(double x);
```

Description

The **rint** function rounds its argument to an integral value in floating-point format, using the current rounding direction.

Returns

The **rint** function returns the rounded integral value.

7.x.9.5 The rinttol function

Synopsis

```
#include <math.h>
long int rinttol(long double x);
```

Description

The `rinttol` function rounds its argument to the nearest integral value, rounding according to the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified. (The `rinttol` function has no overloading macro.)

Returns

The `rinttol` function returns the rounded `long int` value, using the current rounding direction.

7.x.9.6 The `round` function

Synopsis

```
#include <math.h>
double round(double x);
```

Description

The `round` function rounds its argument to the nearest integral value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

[This function rounds like the Fortran `anint` function.]

Returns

The `round` function returns the rounded integral value.

7.x.9.7 The `roundtol` function

Synopsis

```
#include <math.h>
long int roundtol(long double x);
```

Description

The `roundtol` function rounds its argument to the nearest integral value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified. (The `roundtol` function has no overloading macro.)

[This function rounds like the Fortran `nint` function and the Pascal `round` function.]

Returns

The `roundtol` function returns the rounded `long int` value.

7.x.9.8 The `trunc` function

Synopsis

```
#include <math.h>
double trunc(double x);
```

Description

The `trunc` function rounds its argument to the integral value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

The `trunc` function returns the truncated integral value.

7.x.10 Remainder functions

7.x.10.2 The `remainder` function

Synopsis

```
#include <math.h>
double remainder(double x, double y);
```

Description

The `remainder` function computes the remainder `x` REM `y` required by ANSI/IEEE 754 (IEC 559).[11]

Returns

The `remainder` function returns `x` REM `y`.

7.x.10.3 The `remquo` function

Synopsis

```
#include <math.h>
double remquo(double x, double y, int *quo);
```

Description

The `remquo` function computes the same remainder as the `remainder` function. In the object pointed to by `quo` it stores a value whose sign is the sign of `x/y` and whose magnitude is congruent mod 2^n to the magnitude of the integral quotient of `x/y`, where `n` is an implementation-defined integer at least 3.

---

11 "When $y \neq 0$, the remainder $r = x$ *REM* y is defined regardless of the rounding mode by the mathematical relation $r = x - y * n$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

Returns

The **remquo** function returns **x** REM **y**.

[The **remquo** function is intended for implementing argument reductions, which can exploit a few low-order bits of the quotient. Note that **x** may be so large in magnitude relative to **y** that an exact representation of the quotient is not practical.]

Add the clause:

7.x.11 Manipulation functions

7.x.11.1 The copysign function

Synopsis

```
#include <math.h>
double copysign(double x, double y);
```

Description

The **copysign** function produces a value with the magnitude of **x** and the sign of **y**. It produces a NaN (with the sign of **y**) if **x** is a NaN. On implementations that represent a *signed* zero but do not treat negative zero consistently in arithmetic operations, the **copysign** function regards the sign of zero as positive.

[The requirement that **copysign** regard a negative sign of zero as positive if the arithmetic treats negative zero like positive zero is justified in order to preserve more identities. For example, to preserve the identity, *the square root of the product is the product of the square roots*, the algorithm in [22] for the complex square root depends on consistency of **copysign** with the rest of the arithmetic: if -0 behaves like +0 then the square root of the product would yield

$$\sqrt{3} * (-1 - i0) = \sqrt{-3 + i0} \text{ --> } 0 - i\sqrt{3}$$

but if **copysign** were to treat the sign of -0 as negative then the product of the square roots would yield

$$\sqrt{3} * \sqrt{-1 - i0} \text{ --> } \sqrt{3} * (0 - i) = 0 - i\sqrt{3}$$

]

Returns

The **copysign** function returns a value with the magnitude of **x** and the sign of **y**.

7.x.11.2 The nan functions

Synopsis

```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Description

If the implementation supports quiet NaNs for the **double** type, then the call **nan("***n-char-sequence***")** is equivalent to **strtod("NAN(***n-char-sequence***)", (char**) NULL)**; the call **nan("")** is equivalent to **strtod("NAN()", (char**) NULL)**. If **tagp** does not point to an *n-char-sequence* string then the result NaN's content is unspecified. If the implementation does not support quiet NaNs for the **double** type, a call to the **nan** function is unspecified. The **nan** function has no overloading macro.

Similarly **nanf** and **nanl** are defined in terms of **strtof** and **strtold**, respectively.

Returns

The functions **nan**, **nanf**, and **nanl** each returns a quiet NaN, if available, with content indicated through **tagp**.

7.x.11.3 The nextafter functions

Synopsis

```
#include <math.h>
double nextafter(double x, long double y);
float nextafterf(float x, float y);
double nextafterd(double x, double y);
long double nextafterl(long double x, long double y);
```

Description

The **nextafter** function determines the next representable value, in the type of the function (which may be the function invoked by an overloading macro), after **x** in the direction of **y**. The **nextafter** function returns **y** if **x == y**.

The functions **nextafterf**, **nextafterd**, and **nextafterl** each is similar to the **nextafter** function except the type for both parameters and the returned value is **float**, **double**, or **long double**, respectively, and each function has no overloading macro.

Returns

The functions **nextafter**, **nextafterf**, **nextafterd**, and **nextafterl** each returns the next representable value in the specified format after **x** in the direction of **y**.

[It's sometimes desirable to find the next representation after a value in the direction of a previously computed value—maybe smaller, maybe larger. The **nextafter** functions have a second floating argument so that the program will not have to include floating-point tests for determining the direction in such situations. And, on some machines these tests may fail due to overflow, underflow, or roundoff.

The **nextafter** function depends substantially on the expression evaluation method—which is appropriate for certain uses but not for others. The explicitly typed functions can be employed to obtain next values in a particular format. For example,

```
nextafterf(x, y)
```

will return the next **float** value after **(float) x** in the direction of **(float) y** regardless of the evaluation method.

The second parameter of the `nextafter` function has type `long double` primarily to keep the overloading scheme simple. Promotion of the second argument to `long double` is harmless but unnecessary.

For the case `x == y`, the IEEE standard recommends that `x` be returned. This specification differs in order that `nextafter(-0.0, +0.0)` return `+0.0` and `nextafter(+0.0, -0.0)` return `-0.0`.]

7.x.12 Maximum, minimum, and positive difference functions

[The functions in this clause correspond to the standard Fortran functions `dim`, `max`, and `min`.

Their names have `f` prefixes to allow for integer versions—following the example of `fabs` and `abs`.]

7.x.12.1 The `fdim` function

Synopsis

```
#include <math.h>
double fdim(double x, double y);
```

Description

The `fdim` function determines the *positive difference* between its arguments:

```
x - y    if x > y
+0       if x ≤ y
```

A range error may occur.

Returns

The `fdim` function returns the positive difference between `x` and `y`.

7.x.12.2 The `fmax` function

Synopsis

```
#include <math.h>
double fmax(double x, double y);
```

Description

The `fmax` function determines the maximum numeric value of its arguments.[12]

Returns

The `fmax` function returns the maximum numeric value of its arguments.

---

[12] NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then `fmax` chooses the numeric value. See X.10.9.2.

7.x.12.3 The `fmin` function

Synopsis

```
#include <math.h>
double fmin(double x, double y);
```

Description

The `fmin` function determines the minimum numeric value of its arguments.[13]

Returns

The `fmin` function returns the minimum numeric value of its arguments.

7.x.13 Comparison functions

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships—*less*, *greater*, and *equal*—is true. Relational operators may raise the invalid exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.[14] The following subclauses provide functions that are *quiet* (non exception raising) versions of the relational operators, and other comparison functions that facilitate writing efficient code that accounts for NaNs without suffering the invalid exception.

7.x.13.1 The `isgreater` function

Synopsis

```
#include <math.h>
int isgreater(double x, double y);
```

Description

The `isgreater` function determines whether its first argument is greater than its second argument. The value of `isgreater(x,y)` is always equal to `x > y`; however, unlike `x > y`, `isgreater(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

Returns

The `isgreater` function returns the value of `x > y`.

---

[13] `fmin` is analogous to `fmax` in its treatment of NaNs.

[14] ANSI/IEEE 754 (IEC 559) requires that the built-in relational operators raise the invalid exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs.

7.x.13.2 The `isgreaterequal` function

Synopsis

```
#include <math.h>
int isgreaterequal(double x, double y);
```

Description

The `isgreaterequal` function determines whether its first argument is greater than or equal to its second argument. The value of `isgreaterequal(x,y)` is always equal to `x >= y`; however, unlike `x >= y`, `isgreaterequal(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

Returns

The `isgreaterequal` function returns the value of `x >= y`.

7.x.13.3 The `isless` function

Synopsis

```
#include <math.h>
int isless(double x, double y);
```

Description

The `isless` function determines whether its first argument is less than its second argument. The value of `isless(x,y)` is always equal to `x < y`; however, unlike `x < y`, `isless(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

Returns

The `isless` function returns the value of `x < y`.

7.x.13.4 The `islessequal` function

Synopsis

```
#include <math.h>
int islessequal(double x, double y);
```

Description

The `islessequal` function determines whether its first argument is less than or equal to its second argument. The value of `islessequal(x,y)` is always equal to `x <= y`; however, unlike `x <= y`, `islessequal(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

Returns

The `islessequal` function returns the value of `x <= y`.

7.x.13.5 The `islessgreater` function

Synopsis

```
#include <math.h>
int islessgreater(double x, double y);
```

Description

The `islessgreater` function determines whether its first argument is less than or greater than its second argument. The `islessgreater(x,y)` function is always equal to `x < y || x > y`; however, `islessgreater(x,y)` does not raise the invalid exception when `x` and `y` are unordered.

Returns

The `islessgreater` function returns the value of `x < y || x > y`.

7.x.13.6 The `isunordered` function

Synopsis

```
#include <math.h>
int isunordered(double x, double y);
```

Description

The `isunordered` function determines whether its arguments are unordered.

Returns

The `isunordered` function returns 1 if its arguments are unordered and 0 otherwise.

[For implementations with NaNs, the translator should recognize the comparison functions in order to provide efficient implementation. Typical hardware offers efficient quiet comparisons. The semantically correct implementation

```
int isless(long double x, long double y)
{
   return ! (isnan(x) || isnan(y) || x >= y);
}
```

is unsuitable for efficiency reasons.

Programs written for (or ported to) systems with NaNs will be expected to handle invalid and NaN input in reasonable ways. The comparison functions support such programs. The arithmetic operators (+, *, ...) propagate NaNs quietly; the comparison functions facilitate directing NaNs through branches quietly. Thus NaNs can flow through many computations without the need for inefficient or unduly obfuscating code, and without raising inappropriate exceptions.

For implementations without NaNs, the overloading macros can be defined trivially:

```
#define isgreater(x,y) ((x)>(y))
...
#define islessgreater(x,y) ((x)!=(y))
#define isunordered(x,y) 0
```

Several previous versions of this specification proposed extending the relational operators:

Symbol	Relation
`<`	less
`>`	greater
`<=`	less or equal
`>=`	greater or equal
`==`	equal
`!=`	unordered, less, or greater
`!<>=`	unordered
`<>`	less or greater
`<>=`	less, equal, or greater
`!<=`	unordered or greater
`!<`	unordered, greater, or equal
`!>=`	unordered or less
`!>`	unordered, less, or equal
`!<>`	unordered or equal

The additional operators were to be analogous to, and have the same precedence as, the Standard C relational operators. The `!` symbol was to indicate awareness of NaNs, so operators including the `!` symbol would not raise the invalid exception for unordered operands. Where the operands have types and values suitable for relational operators, the semantics detailed in 6.3.8 were to apply. The *operator* syntax in 6.1.5 was to be augmented to include the additional operators. This approach would have had the advantages of brevity and clearer promise of efficiency, at no greater implementation cost for systems that support NaNs (the large majority of systems do support NaNs). However, it was rejected because of reluctance to extend the language definition for functionality which could be provided with a library interface, and because continuing contentiousness might discourage implementation. Also, in some cases, the functions provide a more straightforward articulation, e.g. `isless(x,y)` instead of `! (x !< y)`.

The IEEE standard enumerates 26 functionally distinct comparison predicates, from combinations of the four comparison results and whether invalid is raised. The following table shows how the previous and current specifications cover all important cases:

greater	less	equal	unordered	raises exception	previous old proposal	current current specification
		√			`x == y`	`x == y`
√	√		√		`x != y`	`x != y`
√				√	`x > y`	`x > y`
√		√		√	`x >= y`	`x >= y`
	√			√	`x < y`	`x < y`
	√	√		√	`x <= y`	`x <= y`
			√		`x !<>= y`	`isunordered(x,y)`
√	√			√	`x <> y`	`N/A`
√	√	√		√	`x <>= y`	`N/A`
√			√		`x !<= y`	`! islessequal(x,y)`
√		√	√		`x !< y`	`! isless(x,y)`
	√		√		`x !>= y`	`! isgreaterequal(x,y)`
	√	√	√		`x !> y`	`! isgreater(x,y)`
		√	√		`x !<> y`	`! islessgreater(x,y)`
	√	√	√	√	`! (x > y)`	`! (x > y)`
	√		√	√	`! (x >= y)`	`! (x >= y)`
√		√	√	√	`! (x < y)`	`! (x < y)`
√			√	√	`! (x <= y)`	`! (x <= y)`
√	√	√			`! (x !<>= y)`	`! isunordered(x,y)`
		√	√	√	`! (x <> y)`	`N/A`
			√	√	`! (x <>= y)`	`N/A`
	√	√			`! (x !<= y)`	`islessequal(x,y)`
	√				`! (x !< y)`	`isless(x,y)`
√		√			`! (x !>= y)`	`isgreaterequal(x,y)`
√					`! (x !> y)`	`isgreater(x,y)`
√	√				`! (x !<> y)`	`islessgreater(x,y)`

The previous proposal would have naturally covered the four N/A cases not covered by the current proposal. (The current proposal covers them, except for the invalid exception.) However, covering these cases per se is unimportant, because the facility would provide no additional capability except more ways to write NaN-unaware code.

In the interest of efficiency, note that each quiet combination of less, greater, equal, and unordered can be tested with a single comparison function or equality or relational operator.

The proposal for `!` operators supplanted an earlier proposal that would have augmented the set of relations by using the `?` symbol to denote unordered, for example `a ?>= b` instead of `a !< b`. Use of the `?` relationals would have had the advantage that the unordered case would have been dealt with explicitly. However, the `!` relationals seemed a more natural language extension, particularly from the point of view of programmers for (non-IEEE) implementations not detecting unordered. Also, using `??` as proposed for the unordered operator would have conflicted with trigraphs.

Other macro approaches, such as

```
isrelation(x, FP_UNORDERED | FP_LESS | FP_EQUAL, y)
```

seemed more cumbersome.

Without any language or library support `isgreater(a,b)` might be implemented by the programmer as

```
! (a != a || b != b || a <= b)
```

However, even more awkward code would be required if `a` or `b` had side effects. The programmer would have to remember to put the NaN tests first, and trust the compiler not to replace `a != a || b != b` by *false*. Also, special optimization would be necessary to generate efficient code. Use of `isnan` helps only a little.]