Floating-point <fp.h>

WG14/N467 X3J11/95-068 (Draft 8/24/95)

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

7.x Floating-point <fp.h>

The header `<fp.h>` declares several types, macros and functions to support general floating-point programming. The header `<fp.h>` is intended to supersede `<math.h>`.

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]

[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 requrired different prototypes.

architecture (see rationale 5.2.4.2.2)	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 these type definitions:

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 of modifying the semantics of the `register` storage-class specifier, when applied to a floating type, to mean that the associated value may be wider than the type, was rejected as inconsistent with existing use of `register` in Standard C.]

[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, are both `double` if `FLT_EVAL_METHOD` equals 1, and are both`long double` if `FLT_EVAL_METHOD` equals 2. Note that `float_t` is the narrowest type used by the implementation to evaluate floating expressions.

The macro

```
HUGE_VAL
```

is as defined in `<math.h>`.

The macros

```
HUGE_VALF
HUGE_VALL
```

are `float` and `long double` analogs of `HUGE_VAL`.[2] They expand to positive `float` and `long double` expressions, respectively.

The macro

```
INFINITY
```

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

The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs. It expands to a floating-point 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>`, though such is not required by this specification.]

Should macros like INFINITY and NAN be guaranteed suitable for initializations?

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 `int` 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.]

[2] Like `HUGE_VAL`, the macros `HUGE_VALF` and `HUGE_VALL` can be a positive infinity in an implementation that supports infinities.

528

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). An `fp_contract` macro can occur outside external declarations, and allows or disallows contracted expressions from its occurrence until another `fp_contract` macro is encountered, or until the end of the translation unit. The effect of one of these macros appearing inside an external declaration 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 macro

```
DECIMAL_DIG
```

expands to an `int` constant expression. Its value is an implementation-defined number of decimal digits which is supported by conversion between decimal and all internal floating-point formats.[3] Conversion from (at least) `double` to decimal with `DECIMAL_DIG` digits and back should be the identity function.[4]

[`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.]

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.]

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

[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

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

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

529

7.x.1.1 The `fpclassify` macro

Synopsis

```
#include <fp.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))
```
]

7.x.1.2 The `signbit` macro

Synopsis

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

[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`.

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

530

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 <fp.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.

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 <fp.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 **isnormal** macro returns a nonzero value if and only if its argument has a NaN value.

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

531

7.x.2 Overloading

The *overloading macros* in subsequent subclauses 7.x.3-7.x.12 are function-like macros with parameter types determined by argument types and the implementation's expression evaluation method. For each macro, the *overloading parameters*, indicated with *floating-type* in the synopsis, have the type that is the wider of

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

A return type indicated with *floating-type* in the synopsis matches the type for the overloading parameters.

Suppression of an overloading macro definition makes available an ordinary function with type `double` for the parameters corresponding to the overloading parameters and for return value.

Examples

1. The square root macro has the form

```
floating-type sqrt(floating-type x);
```

`x` is an overloading parameter.

If `FLT_EVAL_METHOD` equals 0, then `float` is the narrowest floating type used for expression evaluation. The type for `sqrt` is `float` if its argument is integral or `float`, `double` if its argument is `double`, and `long double` if its argument is `long double`.

If `FLT_EVAL_METHOD` equals 1, then `double` is the narrowest floating type used for expression evaluation. The type for `sqrt` is `long double` if its argument is `long double`, and `double` in all other cases.

If `FLT_EVAL_METHOD` equals 3, then `long double` is the narrowest floating type used for expression evaluation. The type for `sqrt` is always `long double`.

2. The `remquo` macro has the form

```
floating-type remquo(floating-type x, floating-type y, int *quo);
```

`x` and `y` 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.

532

7.x.3 Comparison macros

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.[8] This subclause provides macros that are *quite* (non exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the invalid exception.

7.x.3.1 The `isgreater` macro

Synopsis

```
#include <fp.h>
int isgreater(floating-type x, floating-type y);
```

Description

The `isgreater` macro 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` macro returns a nonzero value if and only if its first argument is greater than its second argument.

7.x.3.2 The `isgreaterequal` macro

Synopsis

```
#include <fp.h>
int isgreaterequal(floating-type x, floating-type y);
```

Description

The `isgreaterequal` macro 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` macro returns a nonzero value if and only if its first argument is greater than or equal to its second argument.

[8] ANSI/EEE 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.3.3 The `isless` macro

Synopsis

```
#include <fp.h>
int isless(floating-type x, floating-type y);
```

Description

The `isless` macro 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` macro returns a nonzero value if and only if its first argument is less than its second argument.

7.x.3.4 The `islessequal` macro

Synopsis

```
#include <fp.h>
int islessequal(floating-type x, floating-type y);
```

Description

The `islessequal` macro 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` macro returns a nonzero value if and only if its first argument is less than or equal to its second argument.

7.x.3.5 The `islessgreater` macro

Synopsis

```
#include <fp.h>
int islessgreater(floating-type x, floating-type y);
```

Description

The `islessgreater` macro determines whether its first argument is less than or greater than its second argument. The `islessgreater(x,y)` macro is similar in spirit to `(x) < (y) || (x) > (y)`; however, `islessgreater(x,y)` does not raise the invalid exception when `x` and `y` are unordered (nor does it evaluate `x` and `y` twice).

534

Returns

The **`islessgreater`** macro returns a nonzero value if and only if its first argument is less than or greater than its second argument.

7.x.3.6 The `isunordered` macro

Synopsis

```
#include <fp.h>
int isunordered(floating-type x, floating-type y);
```

Description

The **`isunordered`** macro determines whether its arguments are unordered.

Returns

The **`isunordered`** macro returns a nonzero value if and only if its arguments are unordered.

[For implementations with NaNs, the translator should recognize the comparison macros 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 macros support such programs. The arithmetic operators (+, *, ...) propagate NaNs quietly; the comparison macros 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 macros can be defined trivially:

```
#define isgreater(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

535

`!<`	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 macros 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:

				raises	previous	current
greater	less	equal	unordered	exception	old proposal	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 macro 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

536

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.]

7.x.4 Trigonometric macros

The header `<fp.h>` defines overloading macros for the trigonometric functions defined in `<math.h>`.

7.x.4.1 The `acos` macro

Synopsis

```
#include <fp.h>
floating-type acos(floating-type x);
```

7.x.4.2 The `asin` macro

Synopsis

```
#include <fp.h>
floating-type asin(floating-type x);
```

7.x.4.3 The `atan` macro

Synopsis

```
#include <fp.h>
floating-type atan(floating-type x);
```

7.x.4.4 The `atan2` macro

Synopsis

```
#include <fp.h>
floating-type atan2(floating-type y, floating-type x);
```

7.x.4.5 The `cos` macro

Synopsis

```
        #include <fp.h>
        floating-type cos(floating-type x);
```

7.x.4.6 The `sin` macro

Synopsis

```
        #include <fp.h>
        floating-type sin(floating-type x);
```

7.x.4.7 The `tan` macro

Synopsis

```
        #include <fp.h>
        floating-type tan(floating-type x);
```

7.x.5 Hyperbolic macros

The header `<fp.h>` defines overloading macros for the hyperbolic functions defined in `<math.h>`, and for their arc counterparts.

7.x.5.1 The `acosh` macro

Synopsis

```
        #include <fp.h>
        floating-type acosh(floating-type x);
```

Description

The `acosh` macro computes the (nonnegative) arc hyperbolic cosine of `x`.

Returns

The `acosh` macro returns the arc hyperbolic cosine.

7.x.5.2 The `asinh` macro

Synopsis

```
        #include <fp.h>
        floating-type asinh(floating-type x);
```

Description

The `asinh` macro computes the arc hyperbolic sine of `x`.

538

Returns

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

7.x.5.3 The atanh macro

Synopsis

```
#include <fp.h>
floating-type atanh(floating-type x);
```

Description

The **atanh** macro computes the arc hyperbolic tangent of **x**.

Returns

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

7.x.5.4 The cosh macro

Synopsis

```
#include <fp.h>
floating-type cosh(floating-type x);
```

7.x.5.5 The sinh macro

Synopsis

```
#include <fp.h>
floating-type sinh(floating-type x);
```

7.x.5.6 The tanh macro

Synopsis

```
#include <fp.h>
floating-type tanh(floating-type x);
```

7.x.6 Exponential and logarithmic macros and functions

The header **<fp.h>** defines overloading macros for the exponential and logarithmic functions defined in **<math.h>**—except for **modf** which is declared with ordinary functions, and for several related functions.

7.x.6.1 The exp macro

Synopsis

```
#include <fp.h>
floating-type exp(floating-type x);
```

539

7.x.6.2 The `exp2` macro

Synopsis

```
#include <fp.h>
floating-type exp2(floating-type x);
```

Description

The **exp2** macro computes the base-2 exponential of **x**: 2^x.

Returns

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

7.x.6.3 The `expm1` macro

Synopsis

```
#include <fp.h>
floating-type expm1(floating-type x);
```

Description

The **expm1** macro 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**.

Returns

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

7.x.6.4 The `frexp` macro

Synopsis

```
#include <fp.h>
floating-type frexp(floating-type value, int *exp);
```

7.x.6.5 The `ldexp` macro

Synopsis

```
#include <fp.h>
floating-type ldexp(floating-type x, int exp);
```

7.x.6.6 The `log` macro

Synopsis

```
#include <fp.h>
floating-type log(floating-type x);
```

7.x.6.7 The `log10` macro

Synopsis

```
#include <fp.h>
floating-type log10(floating-type x);
```

7.x.6.8 The `log1p` macro

Synopsis

```
#include <fp.h>
floating-type log1p(floating-type x);
```

Description

The `log1p` macro 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)`.

Returns

The `log1p` macro returns the base-e logarithm of 1 plus the argument.

7.x.6.9 The `log2` macro

Synopsis

```
#include <fp.h>
floating-type log2(floating-type x);
```

Description

The `log2` macro computes the base-2 logarithm of `x`.

Returns

The `log2` macro returns the base-2 logarithm.

7.x.6.10 The `logb` macro

Synopsis

```
#include <fp.h>
floating-type logb(floating-type x);
```

Description

The `logb` macro 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`,

$$1 \leq x * \mathtt{FLT\_RADIX}^{-\mathtt{logb(x)}} < \mathtt{FLT\_RADIX}$$

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

541

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` macro returns the signed exponent of its argument.

7.x.6.11 The `modf` functions

Synopsis

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

7.x.6.12 The `scalb` macro

Synopsis

```
#include <fp.h>
floating-type scalb(floating-type x, long int n);
```

Description

The `scalb` macro computes `x * FLT_RADIX`^{n} efficiently, not normally by computing `FLT_RADIX`^{n} explicitly.

Returns

The `scalb` macro 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 macros

The header `<fp.h>` defines overloading macros for the exponential and logarithmic functions defined in `<math.h>`, and for a hypotenuse function.

7.x.7.1 The `fabs` macro

Synopsis

```
#include <fp.h>
floating-type fabs(floating-type x);
```

542

7.x.7.2 The `hypot` macro

Synopsis

```
#include <fp.h>
floating-type hypot(floating-type x, floating-type y);
```

Description

The `hypot` macro computes the square root of the sum of the squares of `x` and `y`, without undue overflow or underflow.

Returns

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

7.x.7.3 The `pow` macro

Synopsis

```
#include <fp.h>
floating-type pow(floating-type x, floating-type y);
```

7.x.7.4 The `sqrt` macro

Synopsis

```
#include <fp.h>
floating-type sqrt(floating-type x);
```

7.x.8 Error and gamma macros

[See [23] regarding implementation.]

7.x.8.1 The `erf` macro

Synopsis

```
#include <fp.h>
floating-type erf(floating-type x);
```

Description

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

Returns

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

543

7.x.8.2 The `erfc` macro

Synopsis

```
#include <fp.h>
floating-type erfc(floating-type x);
```

Description

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

Returns

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

7.x.8.3 The `gamma` macro

Synopsis

```
#include <fp.h>
floating-type gamma(floating-type x);
```

Description

The `gamma` macro computes the gamma function of `x`: $\Gamma(x)$.

Returns

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

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

7.x.8.4 The `lgamma` macro

Synopsis

```
#include <fp.h>
floating-type lgamma(floating-type x);
```

Description

The `lgamma` macro computes the logarithm of the absolute value of gamma of `x`: $\log_e(|\Gamma(x)|)$.

[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 identifiers`signgam` if defined by the user.]

544

Returns

The `lgamma` macro returns $\log_e(|\Gamma(x)|)$.

7.x.9 Nearest integer macros

The header `<fp.h>` defines overloading macros for the nearest integer functions defined in `<math.h>`, for nearest integer functions specified by the ANSI/IEEE floating-point standards, and for functions similar to common Fortran nearest integer functions.

7.x.9.1 The `ceil` macro

Synopsis

```
#include <fp.h>
floating-type ceil(floating-type x);
```

7.x.9.2 The `floor` macro

Synopsis

```
#include <fp.h>
floating-type floor(floating-type x);
```

7.x.9.3 The `nearbyint` macro

Synopsis

```
#include <fp.h>
floating-type nearbyint(floating-type x);
```

Description

The `nearbyint` macro differs from the `rint` macro (7.x.9.4) only in that the `nearbyint` macro 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` macro returns the rounded integral value.

7.x.9.4 The `rint` macro

Synopsis

```
#include <fp.h>
floating-type rint(floating-type x);
```

545

Description

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

Returns

The **`rint`** macro returns the rounded integral value.

7.x.9.5 The `rinttol` macro

Synopsis

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

Description

The **`rinttol`** macro rounds its argument to the nearest **`long int`**, rounding according to the current rounding direction. If the rounded value is outside the range of **`long int`**, the numeric result is unspecified.

Returns

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

7.x.9.6 The `round` macro

Synopsis

```
#include <fp.h>
floating-type round(floating-type x);
```

Description

The **`round`** macro rounds its argument to the nearest integral value in floating-point format, using *add half to the magnitude and chop* rounding a la the Fortran **`anint`** function, regardless of the current rounding direction.

Returns

The **`round`** macro returns the rounded integral value.

7.x.9.7 The `roundtol` macro

Synopsis

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

546

Description

The `roundtol` macro returns the rounded `long int` value, using *add half to the magnitude and chop* rounding a la the Fortran `nint` function and the Pascal `round` function, regardless of the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified.

Returns

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

7.x.9.8 The `trunc` macro

Synopsis

```
#include <fp.h>
floating-type trunc(floating-type x);
```

Description

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

Returns

The `trunc` macro returns the truncated integral value.

7.x.10 Remainder macros

The header `<fp.h>` declares overloading macros for the `<math.h>` `fmod` function,and for two versions of the remainder function required by the ANSI/IEEE 754 (IEC 559) floating-point standard.

7.x.10.1 The `fmod` macro

Synopsis

```
#include <fp.h>
floating-type fmod(floating-type x, floating-type y);
```

7.x.10.2 The `remainder` macro

Synopsis

```
#include <fp.h>
floating-type remainder(floating-type x, floating-type y);
```

547

Description

The **remainder** macro computes the remainder **x** REM **y** required by the ANSI/IEEE 754 (IEC 559) floating-point standard.[9]

Returns

The **remainder** macro returns **x** REM **y**.

7.x.10.3 The remquo macro

Synopsis

```
#include <fp.h>
floating-type remquo(floating-type x, floating-type y, int *quo);
```

Description

The **remquo** macro computes the same remainder as the **remainder** macro. 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.

Returns

The **remquo** macro 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.]

7.x.11 Manipulation macros and functions

The header **<fp.h>** defines overloading macros and functions that manipulate representations in floating formats.

7.x.11.1 The copysign macro

Synopsis

```
#include <fp.h>
floating-type copysign(floating-type x, floating-type y);
```

Description

The **copysign** macro 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

[9] "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.

548

signed zero but do not treat negative zero consistently in arithmetic operations, the `copysign` macro 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

$$\sqrt{3 * (-1 - 0i)} = \sqrt{-3 + 0i} \rightarrow 0 + \sqrt{3}i$$

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

$$\sqrt{3} * \sqrt{-1 - 0i} \rightarrow \sqrt{3} * (0 - i) = 0 - \sqrt{3}i$$

]

Returns

The `copysign` macro returns a value with the magnitude of `x` and the sign of `y`.

7.x.11.2 The `nan` functions

Synopsis

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

Description

If the implementation supports quiet NaNs in the type of the function, 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)`. Similarly `nanf` and `nanl` are defined in terms of `strtof` and `strtold`. If `tagp` does not point to an *n-char-sequence* string then the result NaN's content is unspecified. A call to a `nan` function of a type for which the implementation does not support quiet NaNs is unspecified.

Returns

The `nan` functions return a quiet NaN, if available, with content indicated through `tagp`.

7.x.11.3 The `nextafter` macro and functions

Synopsis

```
#include <fp.h>
floating-type nextafter(floating-type 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);
```

549

Description

The **nextafter** macro and functions determine the next representable value, in the type of the macro or function, after **x** in the direction of **y**. The **nextafter** macro and functions return **y** if **x == y**.

Returns

The **nextafter** macro and functions return the next representable value 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** macro and 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** overloading macro 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** macro 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 macros

The header **<fp.h>** includes overloading macros corresponding to 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 macro

Synopsis

```
#include <fp.h>
floating-type fdim(floating-type x, floating-type y);
```

Description

The **fdim** macro determines the *positive difference* between its arguments:

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

550

Returns

The **fdim** macro returns the positive difference between **x** and **y**.

7.x.12.2 The fmax macro

Synopsis

```
#include <fp.h>
floating-type fmax(floating-type x, floating-type y);
```

Description

The **fmax** macro determines the maximum numeric value of its arguments.[10]

Returns

The **fmax** macro returns the maximum numeric value of its arguments.

7.x.12.3 The fmin macro

Synopsis

```
#include <fp.h>
floating-type fmin(floating-type x, floating-type y);
```

Description

The **fmin** macro determines the minimum numeric value of its arguments.[11]

Returns

The **fmin** macro returns the minimum numeric value of its arguments.

[10] NaN arguments are intended to be treated as missing data. If one argument is a NaN and the other numeric, then **fmax** choses the numeric value. See X.10.9.2.

[11] **fmin** is intended to be analogous to **fmax** in its treatment of NaNs.