

**Rationale for TR 24732**  
**Extension to the programming language C**  
**Decimal Floating-Point Arithmetic**

**2007-07-05**

# Contents

<b>1</b>	<b>Introduction</b> .....	<b>1</b>
1.1	Background .....	1
1.2	The Arithmetic Model .....	3
1.3	The Formats.....	3
<b>2</b>	<b>General</b> .....	<b>4</b>
2.1	Scope.....	4
2.2	References .....	4
<b>3</b>	<b>Predefined macro name</b> .....	<b>4</b>
<b>4</b>	<b>Decimal floating types</b> .....	<b>4</b>
<b>5</b>	<b>Characteristics of decimal floating types &lt;float.h&gt;</b> .....	<b>5</b>
<b>6</b>	<b>Conversions</b> .....	<b>6</b>
6.1	Conversions between decimal floating and integer .....	6
6.2	Conversions among decimal floating types, and between decimal floating types and generic floating types .....	6
6.3	Conversions between decimal floating and complex .....	6
6.4	Usual arithmetic conversions .....	6
6.5	Default argument promotion .....	7
<b>7</b>	<b>Constants</b> .....	<b>7</b>
7.1	Unsuffix decimal floating constant .....	8
7.1.1	Translation time data type .....	8
<b>8</b>	<b>Arithmetic Operations</b> .....	<b>9</b>
8.1	Operators.....	9
8.2	Functions.....	9
8.3	Conversions.....	9
<b>9</b>	<b>Library</b> .....	<b>10</b>
9.1	Standard headers.....	10
9.2	Floating-point environment <fenv.h>.....	10
9.3	Decimal mathematics <math.h>.....	10
9.4	New <math.h> functions.....	11
9.5	Formatted input/output specifiers.....	11
9.6	strtod32, strtod64, and strtod128 functions <stdlib.h>.....	11
9.7	wctod32, wctod64, and wctod128 functions <wchar.h>.....	12
9.8	Type-generic macros <tgmath.h>.....	12
<b>Annex A</b>	.....	<b>13</b>

# 1 Introduction

## 1.1 Background

The existing floating-point types in the C language are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic; the standard does not require the floating-point types to be a specific representation or radix. For this Technical Report, the committee considered both adding decimal floating-point support without introducing additional data types, as well as the current proposal of adding three new types (as per IEEE 754-2007) to the language.

Most applications do not care how floating-point is done. Many applications would be better off using decimal floating-point. Very few applications need the better error bounds of binary floating-point. There will be applications that will need both kinds of floating-point (many will be conversion programs used to convert existing data files from binary floating-point to decimal floating-point). There will be a few applications that will need to run a mixture of third party libraries that only know about binary floating-point, and other third party libraries that only know about decimal floating-point.

Binary floating-point and decimal floating-point (as defined in IEEE 754-2007) occupy the same amount of storage, and they could be treated the same for all data movement and register usage. This means that a function call whose prototype is binary floating-point, but is called with decimal floating-point (and vice versa), can be made to work (as the same number of bytes are passed in the same manner). Hence, adding several functions to the library to convert between binary floating-point and decimal floating-point (for the same sized data) would allow applications to mix both kinds of floating-point. Of course, this means that the application needs to add explicit function calls to do the conversions.

The floating-point unit (FPU) does a binary float operation versus a decimal float operation either by a different opcode or by a switchable mode bit in some control word. In either case, code generation must be controllable by the user. Switching the mode bit at runtime could be done by a function call. But, generating different opcodes require translation time control - a pragma seems like the logical choice; this also works for switching the mode bit.

Based on the above, one might come to the conclusion that adding decimal floating-point support to the language can be done by reusing the existing floating-point types, with some combination of compiler switch, pragma, and conversion routines to enable a mixed binary/decimal floating-point operations. This approach, however, does present several problems.

Variable argument functions do **float** to **double** promotion. This will be incorrect if the hardware promotes as if the data is binary, but the data is really decimal, and vice versa. Explicit calls to some conversion routines would make it work; however, it would be cumbersome to use.

Debugging tools would have no clue if a floating-point object is decimal or binary. That is, a **float**, **double**, or **long double** declaration does not imply the base that will be used for that

object. In fact, the object could be binary floating-point some places in the program and decimal floating-point in others.

Introducing three additional floating-point data types to the language resolves some of these issues. However, adding new data types can be seen as making the language, as well as their use alongside the existing floating-point data types, unnecessarily complex. Some arguments presented for having separate decimal floating-point types are:

1. The fact that there are two sets of floating-point types in itself does not mean the language would become more complex. The complexity question should be answered from the perspective of the user's program; that is, do the new data types add complexity to the user's code? The answer is probably no except for the issues surrounding implicit conversions. For a program that uses only binary floating-point types, or uses only decimal floating-point types, the programmer is still working with three floating-point types. Having additional data types is not making the program more difficult to write, understand, or maintain.
2. Implicit conversions, other than simple assignment, return statement, and function argument passing, can be handled by simply disallowing them (except maybe for cases that involve literals). If we do this, for programs that have both binary and decimal floating-point types, the code is still clean and easy to understand.
3. If we only have one set of data types, and if we provide **STDC** pragmas to allow programs to use both representations, in a large source file with **STDC** pragmas changing the meaning of the types back and forth, the code is actually a field of land mines for the maintenance programmer, who might not immediately be aware of the context of the piece of code.

Since the effect of a pragma is a lexical region within the program, additional debugger information is needed to keep track of the changing meaning of data types.

4. Giving two meanings to one data type hurts type safety. A program may bind by mistake to the wrong library, causing runtime errors that are difficult to trace. It is always preferable to detect errors during compile time. Overloading the meaning of a data type makes the language more complicated, not simpler.
5. A related advantage of using separate types is that it facilitates the use of source checking/scanning utilities (or scripts). They can easily detect which floating-point types are used in a piece of code with just local processing. If a **STDC** pragma can change the representation of a type, the use of `grep`, for example, as an aid to understand and to search program text would become very difficult.
6. Suppose the standard only defines a library for basic arithmetic operations. A C program would have to code an expression by breaking it down into individual function calls. This coding style is error prone, and the resulting code difficult to understand and maintain. A C++ programmer would almost definitely provide his/her own overloaded operators.

Rather than having everyone to come up their own, we should define it in the standard. If C++ defines these types as classes, C should provide a set of types matching the behavior.

This is not a technical issue for an implementation, as it might seem on the surface initially - that is, it might seem easier to just provide new meaning to existing types using a compiler switch - but is an issue about usability for the programmer. The meaning of a piece of code can become obscure if we reuse the **float**, **double**, **long double** types. Also, we have a chance here to bind the C behavior directly with IEEE, reducing the number of variations among implementations. This would help programmer writing portable code, with one source tree building on multiple platforms. Using a new set of data types is the cleanest way to achieve this.

Ultimately if the goal is to be able to bind to IEEE for floating-point arithmetic, C would have to support the data types and operations as specified in IEEE. Not only does IEEE 754-2007 include both binary format and decimal format, it also defines operations (e.g. conversion) between the two radices. Having three additional decimal data types allows such required mixed operations to happen more portably and intuitively within the same translation unit, as opposed to requiring mixing only across translation units.

## 1.2 The Arithmetic Model

Based on a model of decimal arithmetic<sup>1</sup>, which is a formalization of the decimal system of numeration (Algorism), as further defined and constrained by the relevant standards: IEEE 854, ANSI X3-274, and the proposed revision of IEEE 754. The latter is also known as IEEE 754-2007.

## 1.3 The Formats

Based on the current IEEE 754-2007 proposal.

C99 specifies floating-point arithmetic using a two-layer organization. The first layer provides a specification using an abstract model. The representation of a floating-point number is specified in an abstract form where the constituent components of the representation are defined (sign, exponent, significand) but not the internals of these components. In particular, the exponent range, significand size, and the base (or radix) are implementation defined. This allows flexibility for an implementation to take advantage of its underlying hardware architecture. Furthermore, certain behaviors of operations are also implementation defined, for example in the area of handling of special numbers and in exceptions.

The reason for this approach is historical. At the time when C was first standardized, there were already various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would make most of the existing implementations at the time not conforming.

---

<sup>1</sup> A description of the arithmetic model can be found in <http://www2.hursley.ibm.com/decimal/decarith.html>.

C99 provides a binding to IEEE 754 by specifying an annex F and adopting that standard by reference. An implementation not conforming to IEEE 754 can choose to do so by not defining the macro `__STDC_IEC_559__`. This means not all implementations need to support IEEE 754, and the floating-point arithmetic need not be binary.

The technical report specifies decimal floating-point arithmetic according to the IEEE 754-2007, with the constituent components of the representation defined. This is more stringent than the existing C99 approach for the floating types. Since it is expected that all decimal floating-point hardware implementations will conform to the revised IEEE 754, binding to this standard directly benefits both implementers and programmers.

## 2 General

### 2.1 Scope

The technical report is intended to follow the IEEE 754-2007 specification; any conflict between the requirements described in the technical report and IEEE 754-2007 is unintentional. The technical report defers to IEEE 754-2007.

### 2.2 References

## 3 Predefined macro name

A macro is provided to allow users to determine if the technical report is supported by the implementation.

## 4 Decimal floating types

The three new decimal floating-point data types introduced in the technical report have names similar to and characteristics matching those defined in IEEE 754-2007. An alternative naming convention that encapsulates the base (or radix) and precision in the name had also been suggested; for example: `decfp7`, `decfp16`, and `decfp34`, which indicate decimal representation (`dec`), floating point type (`fp`), with the specified number of coefficient digits (7, 16, or 34). However, it was felt that names similar to those used in IEEE 754-2007 may be more appropriate.

Furthermore, a single token used as a type name would make it easier for C++ to implement any types as classes. A more generic single token naming convention, not unlike names like `uint32_t`, can be considered in the future; for example: `_Flt_2_24`, `_Cmplx_10_7`, `Imag_16_6`, etc. Such names can easily be introduced as typedefs to existing types.

Decimal floating types are distinct types from the real floating types **float**, **double**, and **long double**, even if an implementation chooses the same decimal representation for the real floating types.

IEEE 754-2007 specifies its 32-bit decimal format as a *storage format*; i.e. it's intended for efficient storage only and not for computation. The technical report, however, provides operations for the C `_Decimal32` type, as it was felt that 32-bit operations are widely used in the embedded community, particularly for signal processing. There are many classes of applications that have very restricted range and precision requirements.

The technical report does not specify decimal complex nor decimal imaginary types; however, this does not mean that they can not be added in the future.

## 5 Characteristics of decimal floating types `<float.h>`

The characteristics for the new decimal floating types are defined in `<float.h>` by a set of macros similar to the ones defined for real floating types. However, these macros have names that do not match reserved identifiers. To prevent this from making an implementation not conform to ISO/IEC 9899, the functions, type names, and macros added by the technical report are under the control of a macro named `__STDC_WANT_DEC_FP__`, whose name does match the pattern of reserved names in ISO/IEC 9899.

The use of evaluation formats for the new decimal floating types is characterized by the implementation-defined value of `DEC_EVAL_METHOD`, similar to how the evaluation of generic floating types are indicated by the value of `FLT_EVAL_METHOD`.

The `xxx_MAXDIG10` macros originated from a WG14 paper N1151. Although not necessarily related to decimal floating types, it was nevertheless deemed useful by the committee, and therefore added to this technical report.

IEEE 754-2007 uses a floating-point model where normal numbers have one significant digit to the left of the decimal point. This model is different from the model used in C99 for the generic floating-point types, where normal numbers have all significant digits to the right of the decimal point. To avoid possible confusion when dealing with the two floating-point types in C, the technical report adopts the C99 model for decimal floating-point types. Hence, the `DECxx_MIN_EXP` and `DECxx_MAX_EXP` macros are off by one from IEEE 754-2007.

The technical report also introduces the concept of *integer coefficient* and *quantum exponent* of finite decimal numbers, which are essential to decimal semantics. Each decimal number can be represented by a triple of integers  $(s, c, q)$ .

## 6 Conversions

### 6.1 Conversions between decimal floating and integer

When the new type is a decimal floating type, we have three choices: the most positive/negative number representable, positive/negative infinity, and quiet NaN. The first provides no indication to the program that something exceptional has happened. The second provides indication, and since other operations that produce infinity also raise exception, an exception would be raised here for consistency. The third allows the program to detect the condition and provides a way for the implementation to encode the condition (for example, where it occurs).

When the new type is an unsigned integral type, the values that create problems are those less than 0 and those greater than *Utype\_MAX*. There is no overflow/under-flow processing for unsigned arithmetic. A possible choice for the result would be *Utype\_MAX* if the original value is positive, or 0 if negative. Also, implementations are not required to raise signals for signed integer arithmetic. When the new type is a signed integral type, the values that create problems are those less than *type\_MIN* and those greater than *type\_MAX*. The result here should be *type\_MIN* or *type\_MAX* depending on whether the original value is negative or positive.

Conversions between decimal floating and integer formats follow the operation rules as defined in IEEE 754-2007.

In the case where the value being converted is outside the range of values that can be represented, the result is dependent on the rounding mode as specified in the technical report. Note that with decimal floating type, there are two flavors of round to nearest: **FE\_DEC\_TONEAREST** and **FE\_DEC\_TONEARESTFROMZERO**; the effect is the same for both modes.

### 6.2 Conversions among decimal floating types, and between decimal floating types and generic floating types

The specification is similar to the existing ones for **float**, **double** and **long double**, except that when the result cannot be represented exactly, the behavior is defined to become correctly rounded.

### 6.3 Conversions between decimal floating and complex

When a value of decimal floating type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion in [6.2](#) and the imaginary part of the complex result value is a positive zero or an unsigned zero.

### 6.4 Usual arithmetic conversions



In an application that is written using decimal arithmetic, mixed operations between decimal and other real types might not occur frequently. Situations where this might occur are when interfacing with other languages, calling an existing library written in binary floating-point arithmetic, or accessing existing data. The programmer will want to use an explicit cast to control the behavior in such cases to make the code maximally portable. One way to handle usual arithmetic conversion is therefore to disallow mixed operations. The disadvantage of this approach is usability - for example, it could be tedious to add explicit casts in assignments and in function calls when the compiler can correctly handle such situations. Allowing implicit conversions only in simple assignment, return statement, and in argument passing solves this issue.

One major difficulty of allowing mixed operation is in the determination of the common type. C99 does not specify exactly the range and precision of the generic real types. The pecking order between them and the decimal types is therefore unspecified. Given two (or more) mixed type operands, there is no simple rule to define a common type that would guarantee portability in general.

For example, we can define the common type to be the one with greater range. But since a `double` type may have different range under different implementations, a program cannot assume the resulting type of an addition, say, involving both `_Decimal64` and `double`. This imposes limitations on how to write portable programs.

If the generic real type is a type defined in IEEE 754-2007, and if we use the *greater-range rule*, the common type is easily determined. When mixing decimal and binary types of the same type size, decimal type is the common type. When mixing types of different sizes, the common type is the one with larger size. The alternate suggested change in [Annex A](#) uses this approach but does not assume the generic real type to follow IEEE 754-2007. This guarantees consistent behaviors among implementation that uses IEEE 754 in their binary floating-point arithmetic, and at the same time provides reasonable behavior for those that don't.

The committee felt that few programs will require mixed operations, and that requiring explicit cast may result in less error-prone programs.

## 6.5 Default argument promotion

There is no default argument promotion specified for the decimal floating types in the technical report.

## 7 Constants

New suffixes are introduced to denote decimal floating constants. Also, due to the introduction of translation-time data type described in 7.1, it is no longer possible to specify constants of type `double`; the `d` and `D` suffixes are added for this purpose.

Note that decimal floating-point constants that have the same numeric value may be represented differently; e.g. 1.23 (1, 123, -2) has a different representation than 1.230 (1, 1230, -3). In order to preserve the quantum exponent, which is essential to decimal semantics, translation time constant conversion must match runtime conversion.

## 7.1 Unsuffixed decimal floating constant

The proposal for a translation-time data type (TTDT) to allow for the use of unsuffixed floating-point constants originated in WG14 paper N1108. At the Lillehammer meeting, the committee felt that the idea was too important to leave out, and as a minimum it should be a recommended practice in this technical report. There were extensive discussions on whether TTDT should be made part of the rules, i.e. 'required'. In the end the committee decided to make it a separate section in the TR. Note also TTDT could apply to TR 18037.

### 7.1.1 Translation time data type

Translation time data type (TTDT) is an abstract data type which the translator uses as the type for unsuffixed floating constants. A floating constant is kept in this type and representation until an operation requires it to be converted to an actual type. The value of the constant remains exact for as long as possible during the translation process. The concept can be summarized as follows:

1. The implementation is allowed to use a type different from **double** and **long double** as the type of unsuffixed floating constant. This is an implementation defined type. The intention is that this type can represent the constant exactly if the number of decimal digits is within an implementation specified limit. For an implementation that supports decimal floating pointing, a possible choice is the widest decimal floating type.
2. The range and precision of this type are implementation defined and are fixed throughout the program.
3. TTDT is an arithmetic type. All arithmetic operations are defined for this type.
4. Usual arithmetic conversion is extended to handle mixed operations between TTDT and other types. If an operation involves both TTDT and an actual type, the TTDT is converted to an actual type before the operation. There is no "top-down" parsing context information required to process unsuffixed floating constants. Technically speaking, there is no deferring in determining the type of the constant.

Examples:

```
double f;
f = 0.1;
```

Suppose the implementation uses `_Decimal128` as the TTDT. 0.1 is represented exactly after the constant is scanned. It is then converted to **double** in the assignment operator.

```
f = 0.1 * 0.3;
```

Here, both 0.1 and 0.3 are represented in TTDT. If the compiler evaluates the expression during translation time, it would be done using TTDT, and the result would be TTDT. This is then converted to **double** before the assignment. If the compiler generates code to evaluate the expression during execution time, both 0.1 and 0.3 would be converted to **double** before the multiply. The result of the former would be different but more precise than the latter.

```
float g = 0.3f;
f = 0.1 * g;
```

When one operand is a TTDT and the other is one of **float**, **double**, **long double**, the TTDT is converted to **double** with an internal representation following the specification of **FLT\_EVAL\_METHOD** for constant of type double. Usual arithmetic conversion is then applied to the resulting operands.

```
_Decimal32 h = 0.1;
```

If one operand is a TTDT and the other a decimal floating type, the TTDT is converted to **\_Decimal64** with an internal representation specified by **DEC\_EVAL\_METHOD**. Usual arithmetic conversion is then applied.

If one operand is a TTDT and the other a fixed point type, the TTDT is converted to the fixed point type. If the implementation supports fixed point type, it is a recommended practice that the implementation chooses a representation for TTDT that can represent floating and fixed point constants exactly, subjected to a predefined limit on the number of decimal digits.

## 8 Arithmetic Operations

### 8.1 Operators

Since mixed operations between decimal and generic floating types are not allowed, any operation between a decimal float operand and a generic float operand will result in a constraint violation requiring a diagnostic. The only exceptions are simple assignment, return statement, and function argument passing.

### 8.2 Functions

### 8.3 Conversions

## 9 Library

### 9.1 Standard headers

C99 section 7.1.3 prohibits the addition of identifiers to the standard headers unless the names match certain patterns of reserved identifiers. This section of the technical report adds to existing headers many functions and macros that do not match reserved identifiers. To prevent this from making an implementation not conform to ISO/IEC 9899, the functions, type names, and macros added by the technical report are under the control of a user-defined macro named `__STDC_WANT_DEC_FP__`, whose name does match the pattern of reserved names in ISO/IEC 9899.

### 9.2 Floating-point environment `<fenv.h>`

The new, unique rounding mode `FE_DEC_TONEARESTFROMZERO` for decimal floating-point operations corresponds to the IEEE 754-2007 rounding mode “Round to Nearest, Ties Away from Zero”. The other `FE_DEC_XXX` rounding modes are similar to the existing `FE_XXX` rounding modes.

A set of get/set function is added for controlling the rounding direction of decimal floating operations, independent of the rounding direction of generic floating operations.

### 9.3 Decimal mathematics `<math.h>`

The list of elementary functions specified in the mathematics library is extended to handle decimal floating-point types.

The `frexp` function breaks a floating-point number into a normalized fraction and an integer exponent. For decimal floating-point types, the exponent should be base-10.

The `ldexp` function multiplies a floating-point number by an integer exponent. For decimal floating-point types, the exponent should be base-10.

The `remquo` function is not being extended to handle decimal floating-point types, since it is unclear whether a base-10 version is useful or required.

C99 suggests coding the classification macros (example in 7.12.3.1) using the `sizeof` operator to distinguish `float` from `double` from `long double`. However, this is no longer a sufficient solution with the introduction of decimal data types. There appears to be no C99 conforming solution for a library provider to code these macros. It has been suggested that we add a language operator like *typeof* or *radixof* to help determine if a value is decimal or binary. However, it was felt that an implementation is already given enough latitude, and the issue can be resolved by

‘compiler magic’. The committee agreed that the operator can be reconsidered in a future revision of the standard.

## 9.4 New `<math.h>` functions

IEEE 754-2007 specifies two additional decimal floating-point operations: *samequantum* and *quantize*. These are implemented as new library functions in C99. The library functions have the same semantics as the IEEE operations.

The **quantexp** function returns the quantum exponent of a decimal floating-point value. It can be used with **scalbn** to compute the signed coefficient in decimal floating format. For example, for finite **\_Decimal64** *x*, the signed coefficient is equal to **scalbnd64(x, -quantexpd64(x))**, which **printf** with **%.ODF** renders as an integer decimal character sequence. Note that integer types might not have enough precision to represent coefficients.

Note that a set of encode/decode functions to manage alternate encodings (as required by IEEE 754-2007) was considered for the technical report. However, it was felt that these functions were too esoteric and incomplete, and that serialization (using **printf/scanf**) is the only portable way to stream data in and out of storage.

## 9.5 Formatted input/output specifiers

New length modifiers are introduced for decimal floating types.

One suggestion was to simply add **D** to existing conversion specifiers to indicate decimal floating types; e.g. **Df** for **\_Decimal64** and **DLf** for **\_Decimal128**. However, there's no existing length specifier for **float**, as floats are default promoted to doubles. Therefore a specifier for **\_Decimal32** does not exist with this scheme.

The precision of the length modifiers is defaulted to 6, similar to the current specification for generic floating type. Since the precision of the decimal floating type is encoded in the representation, it has been suggested that we use the encoded precision as the default instead. The committee felt that having a default precision that's predictable and consistent with the generic floating type is important, so the quantum has no effect on e/E/f/F/g/G output conversions. The committee chose instead to define a/A conversions for decimal floating-point to reflect the quantum in a reasonably intuitive way.

## 9.6 **strtod32, strtod64, and strtod128** functions `<stdlib.h>`

The latest IEEE 754-2007 draft requires that floating-point overflow be raised for values that are too large or too small. As such, setting **errno** to **ERANGE** as currently proposed does not meet

those requirements (but does match the strtod family). Perhaps the requirements of 7.12.1#4 of math\_errhandling should be applied to the strtol\* functions.

Note that IEEE 754-2007 requires conversions from internal decimal formats to external decimal character sequences and back that preserve the representation in the internal format. The technical report supports this requirement.

## 9.7 wcstod32, wcstod64, and wcstod128 functions <wchar.h>

## 9.8 Type-generic macros <tgmath.h>

According to C99 7.22p3, “if any argument for generic parameters has type **double** or is of integer type, the type determined is **double**.” That is,

```
sqrt(9);      // expands to the double version of sqrt
```

Therefore the new rules for type-generic macro handling for decimal floating types can not change this behavior.

## Annex A

The following is an alternate suggestion to [usual arithmetic conversions](#) using the *greater-range rule*.

Insert the following to 6.3.1.8#1, after "This pattern is called the *usual arithmetic conversions*:"

6.3.1.8[1]

... This pattern is called the *usual arithmetic conversions*:

If one operand is a decimal floating type and there are no complex types in the operands:

If either operand is `_Decimal128` or `long double`, the other operand is converted to `_Decimal128`.

Otherwise, if either operand is `_Decimal64` or `double`, the other operand is converted to `_Decimal64`.

Otherwise, if either operand is `_Decimal32`, the other operand is converted to `_Decimal32`.

If one operand is a decimal floating type and the other is a complex type, the decimal floating type is converted to the first type in the following list that can represent the value range: `float`, `double`, `long double`. It is converted to `long double` if no type in the list can represent its value range. In either case, the complex type is converted to a type whose corresponding real type is this converted type. Usual arithmetic conversions is then applied to the converted operands.

During any of the above conversions, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is correctly rounded. If the value being converted is outside the range of values that can be represented, the result is dependent on the rounding mode. If the rounding mode is:

*near*, if the value being converted is less than the maximum representable value of the target type plus 0.5 ulp, there is no overflow and the result is the maximum value of the target type<sup>2</sup>; otherwise the absolute value of the result is one of `HUGE_VAL`, `HUGE_VALF`, `HUGE_VALL`, `HUGE_VAL_D64`, `HUGE_VAL_D32` or `HUGE_VAL_D128` depending on the result type and the sign is the same as the value being converted.

*zero*, the value is the most positive finite number representable if the value being converted is positive, and the most negative finite number representable otherwise.

---

<sup>2</sup> That is, the values that are between `MAX` and `MAX+10Emax*ulp/2`

*positive infinity*, the value is same as for rounding mode *zero* if the value being converted is negative; otherwise the result is one of positive **HUGE\_VAL**, **HUGE\_VALF**, **HUGE\_VALL**, **HUGE\_VAL\_D64**, **HUGE\_VAL\_D32** or **HUGE\_VAL\_D128** depending on the result type.

*negative infinity*, the value is same as for rounding mode *zero* if the value being converted is positive; otherwise the result is one of negative **HUGE\_VAL**, **HUGE\_VALF**, **HUGE\_VALL**, **HUGE\_VAL\_D64**, **HUGE\_VAL\_D32** or **HUGE\_VAL\_D128** depending on the result type.

If there are no decimal floating type in the operands:

First, if the corresponding real type of either operand is **long double**, the other operand is converted, without ... <the rest of 6.3.1.8#1 remains the same>