ISO/IEC JTC1 SC22 WG14 N1176

Date: 2006-05-24

Reference number of document: **ISO/IEC WDTR 24732**

Committee identification: ISO/IEC JTC1 SC22 WG14

SC22 Secretariat: ANSI

Information Technology —

Programming languages, their environments and system software interfaces —

Extension for the programming language C to support decimal floating-point arithmetic —

Document type: Technical Report Type 2
Document subtype: n/a
Document stage: (2) Working Draft Technical Report
Document language: E

Contents

1 Introduction

1.1 Background

Most of today's general purpose computing architectures provide binary floating-point arithmetic in hardware. Binary floating-point is an efficient representation which minimizes memory use, and is simpler to implement than floating-point arithmetic using other bases. It has therefore become the norm for scientific computations, with almost all implementations following the IEEE-754 standard for binary floating-point arithmetic.

However, human computation and communication of numeric values almost always uses decimal arithmetic and decimal notations. Laboratory notes, scientific papers, legal documents, business reports and financial statements all record numeric values in decimal form. When numeric data are given to a program or are displayed to a user, binary to-and-from decimal conversion is required. There are inherent rounding errors involved in such conversions; decimal fractions cannot, in general, be represented exactly by binary floating-point values. These errors often cause usability and efficiency problems, depending on the application.

These problems are minor when the application domain accepts, or requires results to have, associated error estimates (as is the case with scientific applications). However, in business and financial applications, computations are either required to be exact (with no rounding errors) unless explicitly rounded, or be supported by detailed analyses that are auditable to be correct. Such applications therefore have to take special care in handling any rounding errors introduced by the computations.

The most efficient way to avoid conversion error is to use decimal arithmetic. Currently, the IBM zArchitecture (and its predecessors since System/360) is a widely used system that supports built-in decimal arithmetic. This, however, provides integer arithmetic only, meaning that every number and computation has to have separate scale information preserved and computed in order to maintain the required precision and value range. Such scaling is difficult to code and is error-prone; it affects execution time significantly, and the resulting program is often difficult to maintain and enhance.

Even though the hardware may not provide decimal arithmetic operations, the support can still be emulated by software. Programming languages used for business applications either have native decimal types (such as PL/I, COBOL, C#, or Visual Basic) or provide decimal arithmetic libraries (such as the BigDecimal class in Java). The arithmetic used, nowadays, is almost invariably decimal floating-point; the COBOL 2002 ISO standard, for example, requires that all standard decimal arithmetic calculations use 32-digit decimal floating-point.

At present, most implementations use software for decimal arithmetic. Even the best packages are slow, and can be 100 times slower than a corresponding hardware implementation, and in some cases much slower. At least one processor manufacturer, therefore, is adding decimal floating-point in hardware.

Arguably, the C language hits a sweet spot within the wide range of programming languages available today – it strikes an optimal balance between usability and performance. Its simple and expressive syntax makes it easy to program; and its close-to-the-hardware semantics makes it efficient. Despite the advent of newer programming languages, C is still often used together with other languages to code the computationally intensive part of an application. In many cases, entire business applications are written in C/C++. To maintain the vitality of C, the need for decimal arithmetic by the business and financial community cannot be ignored.

The importance of this has been recognized by the IEEE. The IEEE 754 standard is currently being revised, and the major change in that revision is the addition of decimal floating-point formats and arithmetic. These decimal data types are almost as efficient as the binary types, and are especially suitable for hardware implementation; it is possible that they will become the most widely used primitive data types once hardware implementations are available.

Historically there has been a close tie between IEEE-754 and C with respect to floating-point specification. This Technical Report proposes to add decimal floating types and arithmetic to the C programming language specification.

1.2 The Arithmetic Model

The proposal of this Technical Report is based on a model of decimal arithmetic[1] 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-754R.

There are three components to the model:

- *numbers* - which represent the values which can be manipulated by, or be the results of, the core operations defined in the model
- *operations* - the core operations (such as addition, multiplication, etc.) which can be carried out on numbers
- *context* - which represents the user-selectable parameters, the status of the operations (for example, any exceptions they caused), and rules which govern the results of arithmetic operations (for example, the rounding mode to be used)

The model defines these components in the abstract. It neither defines the way in which operations are expressed (which might vary depending on the computer language or other interface being used), nor does it define the concrete representation (specific layout in storage, or in a processor's register, for example) of numbers or context.

From the perspective of the C language, *numbers* are represented by data types, *operations* are defined within expressions, and *context* is the floating environment specified in **`<fenv.h>`**. This Technical Report specifies how the C language implements these components.

---

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

1.3 The Encodings

Based on the arithmetic model, encodings have been proposed to support the general purpose floating-point decimal arithmetic described in the Decimal Arithmetic Specification[2]. The encodings are the product of discussions by a subcommittee of the IEEE committee IEEE-754R which is currently revising the IEEE 754-1985 standard.

C99 specifies floating-point arithmetic using a two-layer organization. The first layer provides a specification using an abstract model. The representation of floating-point number is specified in an abstract form where the constituent components of the representation is 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.

C99 provides a binding to IEEE-754 by specifying an Annex F, *IEC 60559 floating point arithmetic*, and adopting that standard by reference. An implementation may choose not to conform to IEEE-754 and indicates that 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.

This Technical Report specifies decimal floating-point arithmetic according to the IEEE-754R, 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

This Technical Report specifies an extension to the programming language C, specified by the international standard ISO/IEC 9899:1999. The extension provides support for decimal floating-point arithmetic that is consistent with the specification in IEEE-754R.

This Technical Report does not specify binary floating-point arithmetic.

---

[2] A description of the encodings can be found in http://www2.hursley.ibm.com/decimal/decbits.html.

2.2 References

The following standards contain provisions which, through reference in this text, constitute provisions of this Technical Report. For dated references, subsequent amendment to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred applies. Members of IEC and ISO maintain registers of current valid International Standards.

ISO/IEC 9899:1999, *Information technology - Programming languages, their environments and system software interfaces - Programming Language C.*

ISO/IEC 9899:1999/Cor 1:2001, *Information technology - Programming languages, their environments and system software interfaces - Programming Language C – Technical Corrigendum 1*.

ISO/IEC 9899:1999/Cor 2:2004, *Information technology - Programming languages, their environments and system software interfaces - Programming Language C – Technical Corrigendum 2*.

ISO/IEC TR 18037, *Information technology - Programming languages, their environments and system software interfaces – Extensions for the programming language C to support embedded processors.*

ISO/IEC 1989:2002, *Information technology - Programming languages - COBOL.*

IEC 60559:1989, *Binary floating-point arithmetic for microprocessors systems* (previously designated IEC 559:1989).

ANSI X3.274, *Information Technology - Programming Language REXX.*

ANSI/IEEE 754-1985 - *IEEE Standard for Binary Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 1985.

ANSI/IEEE 854-1987 - *IEEE Standard for Radix-Independent Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 1987.

The IEEE 754 revision working group is currently revising the specification for floating-point arithmetic:

ANSI/IEEE 754R - *IEEE Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc. Draft.

A Decimal Floating-Point Specification, Schwarz, Cowlishaw, Smith, and Webb, in the *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (Arith 15)*, IEEE, June 2001.

Note: Reference materials relating to IEEE-754R can be found in http://grouper.ieee.org/groups/754/ and http://www.validlab.com/754R/.

3 Predefined macro name

The following macro name is conditionally defined by the implementation:

`__STDC_DEC_FP__` The integer constant 1, intended to indicate conformance to this technical report.

4 Decimal floating types

This Technical Report introduces three decimal floating types, designated as **`_Decimal32`**, **`_Decimal64`** and **`_Decimal128`**. The set of values of type **`_Decimal32`** is a subset of the set of values of the type **`_Decimal64`**; the set of values of the type **`_Decimal64`** is a subset of the set of values of the type **`_Decimal128`**.

Within the type hierarchy, decimal floating types are base types, real types and arithmetic types.

The types **`float`**, **`double`**, and **`long double`** are also called generic floating types for the purpose of this Technical Report.

Note: C does not specify a radix for **`float`**, **`double`** and **`long double`**. An implementation can choose the representation of **`float`**, **`double`** and **`long double`** to be the same as the decimal floating types. In any case, the decimal floating types are distinct from **`float`**, **`double`** and **`long double`** regardless of the representation.

Note: This Technical Report does not define decimal complex types or decimal imaginary types. The three complex types remain as **`float _Complex`**, **`double _Complex`** and **`long double _Complex`**, and the three imaginary types remain as **`float _Imaginary`**, **`double _Imaginary`** and **`long double _Imaginary`**.

Suggested changes to C99:

Change the first sentence of 6.2.5#10.

[10] There are three *generic floating types*, designated as **`float`**, **`double`** and **`long double`**.

Add the following paragraphs after 6.2.5#10.

[10a] There are three *decimal floating types*, designated as **`_Decimal32`**, **`_Decimal64`** and **`_Decimal128`**. The set of values of the type **`_Decimal32`** is a subset of the set of values of the type **`_Decimal64`**; the set of values of the type **`_Decimal64`** is a subset of the set of values of the type **`_Decimal128`**. Decimal floating types are real floating types.

[10b] Together, the generic floating types and the decimal floating types comprise the *real floating types*.

Add the following to 6.7.2 Type specifiers:

```
type-specifier:
        _Decimal32
        _Decimal64
        _Decimal128
```

5 Characteristics of decimal floating types <float.h>

The characteristics of decimal floating types are defined in terms of a model specifying general decimal arithmetic (1.2). The encodings are specified in IEEE-754R (1.3).

The three decimal encoding formats defined in IEEE-754R correspond to the three decimal floating types as follows:

- **`_Decimal32`** is a *decimal32* number, which is encoded in four consecutive octets (32 bits)
- **`_Decimal64`** is a *decimal64* number, which is encoded in eight consecutive octets (64 bits)
- **`_Decimal128`** is a *decimal128* number, which is encoded in 16 consecutive octets (128 bits)

The value of a finite number is given by $(-1)^{sign}$ x coefficient x $10^{exponent}$. Refer to IEEE-754R for details of the format.

These formats are characterized by the length of the coefficient, and the maximum and minimum exponent. The coefficient is not normalized, so trailing zeros are significant; i.e., 1.0 is equal to but can be distinguished from 1.00. The table below shows these characteristics by format:

Format	**`_Decimal32`**	**`_Decimal64`**	**`_Decimal128`**
Coefficient length in digits	7	16	34
Maximum Exponent (E_{max})	96	384	6144
Minimum Exponent (E_{min})	-95	-383	-6143

If the macro **`__STDC_WANT_DEC_FP__`** is defined at the point in the source file where the header **`<float.h>`** is included, the header **`<float.h>`** shall define several macros that expand to various limits and parameters of the decimal floating types. The names and meaning of these macros are similar to the corresponding macros for generic floating types.

Suggested change to C99:

Add the following after 5.2.4.2.2:

5.2.4.2.2a Characteristics of decimal floating types `<float.h>`

[1] The characteristics of decimal floating types are defined in terms of the format described in IEEE-754R. The finite numbers are defined by a sign, an exponent (which is a power of ten), and a decimal integer coefficient. The value of a finite number is given by $(-1)^{sign}$ x coefficient x $10^{exponent}$. The macros defined in **`<float.h>`** provide the characteristics of these representations, which is defined in the Decimal Arithmetic Encoding. The prefixes **`DEC32_`**, **`DEC64_`**, and **`DEC128_`** are used to denote the types **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** respectively.

[2] Except for assignment and casts, the values of operations with decimal floating operands and values subject to the usual arithmetic conversions and of decimal floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of **`DEC_EVAL_METHOD`**:

-1 indeterminable;
0 evaluate all operations and constants just to the range and precision of the type;
1 evaluate operations and constants of type **`_Decimal32`** and **`_Decimal64`** to the range and precision of the **`_Decimal64`** type, evaluate **`_Decimal128`** operations and constants to the range and precision of the **`_Decimal128`** type;
2 evaluate all operations and constants to the range and precision of the **`_Decimal128`** type.

All other negative values for **`DEC_EVAL_METHOD`** characterize implementation-defined behavior.

[3] The values given in the following lists shall be replaced by constant expressions suitable for use in **`#if`** preprocessing directives:

- number of digits in the coefficient

```
DEC32_MANT_DIG      7
DEC64_MANT_DIG      16
DEC128_MANT_DIG     34
```

- minimum exponent

```
DEC32_MIN_EXP       -95
DEC64_MIN_EXP       -383
```

```
DEC128_MIN_EXP      -6143
```

- maximum exponent

```
DEC32_MAX_EXP       96
DEC64_MAX_EXP       384
DEC128_MAX_EXP      6144
```

- maximum representable finite decimal floating number (there are 6, 15 and 33 9's after the decimal points respectively)

```
DEC32_MAX      9.999999E96DF
DEC64_MAX      9.999999999999999E384DD
DEC128_MAX     9.999999999999999999999999999999999E6144DL
```

- the difference between 1 and the least value greater than 1 that is representable in the given floating point type

```
DEC32_EPSILON  1E-6DF
DEC64_EPSILON  1E-15DD
DEC128_EPSILON 1E-33DL
```

- minimum normalized positive decimal floating number

```
DEC32_MIN      1E-95DF
DEC64_MIN      1E-383DD
DEC128_MIN     1E-6143DL
```

- minimum denormalized positive decimal floating number

```
DEC32_DEN      0.000001E-95DF
DEC62_DEN      0.000000000000001E-383DD
DEC128_DEN     0.000000000000000000000000000000001E-6143DL
```

[4] An implementation shall define the following macros if and only if it supports[3] subnormal (also known as denormal) numbers of the respective types. Their value is the minimum denormalized positive floating-point number:

```
FLT_DEN
DBL_DEN
LDBL_DEN
```

Their values are typically, but not always, `FLT_MIN * FLT_EPSILON`, `DBL_MIN * DBL_EPSILON`, `LDBL_MIN` * `LDBL_EPSILON`, respectively.

---

[3] Support means that they are not flushed to zero when used as operands, nor, when an arithmetic operation produces them.

[5] The number of base 10 digits required to ensure that floating-point numbers with p radix b digits which differ by only one unit in the last place (ulp) are always differentiated,

$$\begin{cases} p\log_{10}b & \text{if } b \text{ is power of } 10 \\ \lceil 1+p\log_{10}b \rceil & \text{otherwise} \end{cases}$$

`FLT_MAXDIG10`
`DBL_MAXDIG10`
`LDBL_MAXDIG10`

6 Conversions

6.1 Conversions between decimal floating and integer

For conversions between real floating and integer types, C99 6.3.1.4 leaves the behavior undefined if the conversion result cannot be represented (Annex F.4 tightened up the behavior.) To help writing portable code, this Technical Report provides defined behavior for decimal floating type. Furthermore, it is useful to allow program execution to continue without interruption unless the program needs to check the condition.

Suggested changes to C99:

Change the first sentence of 6.3.1.4 paragraph 1:

[1] When a finite value of generic floating type is converted to an integer type …

Add the follow paragraph after 6.3.1.4 paragraph 1:

[1a] When a finite value of decimal floating type is converted to an integer type other than **`_Bool`**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the “invalid” floating-point exception shall be raised. If the integer type is unsigned, the result shall be the largest representable number if the decimal floating point number is positive, or 0 otherwise. If the integer type is signed, the result shall be the most negative or positive number according to the sign of the floating point number.

Change the first sentence of 6.3.1.4 paragraph 2:

[2] When a value of integer type is converted to a generic floating type, …

Add the following paragraph after 6.3.1.4 paragraph 2:

[2a] When a value of integer type is converted to a decimal floating type, 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 shall be 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 and the “overflow” floating-point exception shall be raised. 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 unit in the last place (ulp), the result is the maximum value of the target type[4]; 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.
>
> *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 *near* if the value being converted is negative; 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.

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 tightened to become correctly rounded.

Suggested change to C99:

Add after 6.3.1.5#2.

[3] When a **`_Decimal32`** is promoted to **`_Decimal64`** or **`_Decimal128`**, or a **`_Decimal64`** is promoted to **`_Decimal128`**, the value is converted to the type being promoted to. All extra precision and/or range (for the converted type) are removed.

[4] When a **`_Decimal64`** is demoted to **`_Decimal32`**, a **`_Decimal128`** is demoted to **`_Decimal64`** or **`_Decimal32`**, or conversion is performed among decimal and generic floating types other than the above, if the value being converted can be represented exactly in the new type,

---

[4] That is, the values that are between MAX and MAX+10^{Emax}*ulp/2

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, the result is the maximum value of the target type[5]; 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.
>
> *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 *near* if the value being converted is negative; 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.

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.

This is covered by C99 6.3.1.7.

6.4 Usual arithmetic conversions

In an application that is written using decimal arithmetic, mixed operations between decimal and other real types are likely to occur only when interfacing with other languages, calling existing libraries written for binary floating point arithmetic, or accessing existing data. Determining the common type for mixed operations is difficult because ranges overlap; therefore, mixed mode operations are not allowed and the programmer must use explicit casts. Implicit conversions are allowed only for simple assignment and in argument passing.

---

[5] That is, the values that are between MAX and MAX+10^{Emax}*ulp/2

Following are suggested changes to C99:

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, all other operands shall not be generic floating type, complex type or imaginary type:

> First if either operand is **`_Decimal128`**, the other operand is converted to **`_Decimal128`**.
>
> Otherwise, if either operand is **`_Decimal64`**, the other operand is converted to **`_Decimal64`**.
>
> Otherwise, if either operand is **`_Decimal32`**, the other operand is converted to **`_Decimal32`**.

If there are no decimal floating types 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>

6.5 Default argument promotion

There is no default argument promotion specified for the decimal floating types. Default argument promotion covered in C99 6.5.2.2 [6] and [7] remains unchanged, and applies to generic floating types only.

7 Constants

New suffixes are added to denote decimal floating constants: **`DF`** for **`_Decimal32`**, **`DD`** for **`_Decimal64`**, and **`DL`** for **`_Decimal128`**.

Suggested changes to C99:

Add the following to 6.4.4.2 floating-suffix.

> *floating-suffix*: one of
> **`f d l F D L df dd dl DF DD DL`**

Add the following paragraph after 6.4.4.2#2:

6.4.4.2
...
[2a] **Constraints**

The *floating-suffix* **`df`**, **`dd`**, **`dl`**, **`DF`**, **`DD`** and **`DL`** shall not be used in a *hexadecimal-floating-constant*.

Change 6.4.4.2#4 to:

[4] An unsuffixed floating constant has type **`double`**. If suffixed by the letter **`f`** or **`F`**, it has type **`float`**. If suffixed by the letter **`d`** or **`D`**, it has type **`double`**. If suffixed by the letter **`l`** or **`L`**, it has type **`long double`**.

Add the following paragraph after 6.4.4.2#4:

6.4.4.2
...
[4a] If a floating constant is suffixed by **`df`** or **`DF`**, it has type **`_Decimal32`**. If suffixed by **`dd`** or **`DD`**, it has type **`_Decimal64`**. If suffixed by **`dl`** or **`DL`**, it has type **`_Decimal128`**.

7.1 Unsuffixed floating constant

The above introduces new suffixes for the decimal floating constants. It would help usability if unsuffixed floating constant could be used. The issue can be illustrated by the following example:

```
_Decimal64 rate = 0.1;
```

The constant 0.1 has type **`double`**. In an implementation where binary representation is used for the floating types, the internal representation of 0.1 cannot be exact. The variable *rate* will get a value slightly different from 0.1. This defeats the purpose of decimal floating types. On the other hand, requiring programmers to write:

```
_Decimal64 rate = 0.1dd;
```

can be inconvenient and affect readability of the program.

7.1.1 Translation time data type

The main idea is to introduce a translation time data type (TTDT) 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 point, 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`** with an internal representation following the specification of **`FLT_EVAL_METHOD`** for constant of type **`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`** with an internal representation following the specification of **`FLT_EVAL_METHOD`** for constant of type **`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`** with an internal representation following the specification of **`FLT_EVAL_METHOD`** for constant of type **`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`**, or **`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[6], 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 an implementation defined limit on the number of decimal digits.

Suggested changes to C99:

Below are suggested changes to C99. Fixed point types are not considered in these changes.

In 6.2.5 after paragraph 28, add a paragraph:

[28a] There is an implementation defined data type called the *translation time data type*, or *TTDT*. TTDT is an arithmetic type and is used as the type for unsuffixed floating constants. It shall have at least the range and precision of the **`double`** type. There is no type specifier for TTDT.

Replace 6.4.4.2 paragraph 4 with the following:

[4] An unsuffixed floating constant has type TTDT. If suffixed by the letter **`f`** or **`F`**, it has type **`float`**. If suffixed by the letter **`d`** or **`D`**, it has type **`double`**. If suffixed by the letter **`l`** or **`L`**, it has type **`long double`**.

Recommended practice

If the implementation supports decimal floating types, TTDT should have at least the range and precision of the **`_Decimal128`** type.

Add the following paragraphs after 6.3.1.7:

6.3.1.7a Translation Time Data Type

When a TTDT is converted to **`double`**, it is converted to the internal representation specified by **`FLT_EVAL_METHOD`**.

Recommended practice

---

[6] As defined in ISO/IEC TR 18037.

The conversion of TTDT to **`double`** should match the execution-time conversion of character strings by library functions, such as **`strtod`**, given matching inputs suitable for both conversions, the same format and default execution-time rounding.

6.3.1.7b

Before the *usual arithmetic conversions* are carried out, if one operand is TTDT and the other is not, and is not a decimal floating type, the TTDT operand is converted to **`double`**. Otherwise, the behavior is implementation defined.

8 Arithmetic Operations

8.1 Operators

The operators *Add* (C99 6.5.6), *Subtract* (C99 6.5.6), *Multiply* (C99 6.5.5), *Divide* (C99 6.5.5), *Relational operators* (C99 6.5.8), *Equality operators* (C99 6.5.9), *Unary Arithmetic operators* (C99 6.5.3.3), and *Compound Assignment operators* (C99 6.5.16.2) when applied to decimal floating type operands shall follow the semantics as defined in IEEE 754R.

Suggested changes to C99:

Add the following after 6.5.5 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall have decimal floating type.

Add the following after 6.5.6 paragraph 3:

[3a] If either operand has decimal floating type, the other operand shall have decimal floating type.

Add the following after 6.5.8 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall have decimal floating type.

Add the following after 6.5.9 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall have decimal floating type.

Add the following after 6.5.16.2 paragraph 2:

[2a] If either operand has decimal floating type, the other operand shall have decimal floating type.

8.2 Functions

The decimal floating point operations square root, min, max, fused multiply-add and remainder, which are defined in IEEE 754R, are implemented as library functions.

8.3 Conversions

Conversions between different formats and to/from integer formats are covered in section 6.

9 Library

9.1 Standard headers

The functions, macros, and types declared or defined in Clause 9 and its subclauses are only declared or defined by their respective headers if the macro **`__STDC_WANT_DEC_FP__`** is defined at the point in the source file where the appropriate header is included.

9.2 Floating-point environment <fenv.h>

The floating point environment specified in C99 7.6 applies to both generic floating types and decimal floating types. This is to implement the *context* defined in IEEE 754R. The existing C99 specification gives flexibility to implementation on which part of the environment is accessible to programs. The decimal floating-point arithmetic specifies a more stringent requirement. All the rounding directions and flags are supported.

DEC Macros	Existing C99 macros for generic floating types	IEEE 754
`FE_DEC_TOWARDZERO`	**`FE_TOWARDZERO`**	Toward zero
`FE_DEC_TONEAREST`	**`FE_TONEAREST`**	To nearest, ties even
`FE_DEC_UPWARD`	**`FE_UPWARD`**	Toward plus infinity
`FE_DEC_DOWNWARD`	**`FE_DOWNWARD`**	Toward minus infinity
`FE_DEC_TONEARESTFROMZERO`	n/a	To nearest, ties away from zero

Suggested changes to C99:

Add the following after 7.6 paragraph 7:

7.6
...
[7a] Each of the macros

`FE_DEC_DOWNWARD`
`FE_DEC_TONEAREST`

FE_DEC_TONEARESTFROMZERO
FE_DEC_TOWARDZERO
FE_DEC_UPWARD

is defined and used by **`fe_dec_getround`** and **`fe_dec_setround`** functions for getting and setting the rounding direction of decimal floating-point operations. The default rounding direction for decimal floating-point operations shall be **`FE_DEC_TONEAREST`**.

Add the following after 7.6.3.2:

7.6.3.3 The fe_dec_getround function

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <fenv.h>
int fe_dec_getround(void);
```

Description

The **`fe_dec_getround`** function gets the current rounding direction for decimal floating-point operations.

Returns

The **`fe_dec_getround`** function returns the value of the rounding direction macro representing the current rounding direction for decimal floating-point operations, or a negative value if there is no such rounding macro or the current rounding direction is not determinable.

7.6.3.4 The fe_dec_setround function

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <fenv.h>
int fe_dec_setround(int round);
```

Description

The **`fe_dec_setround`** function establishes the rounding direction for decimal floating-point operations represented by its argument round. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

If **`FLT_RADIX`** is not 10, the rounding direction altered by the **`fesetround`** function is independent of the rounding direction altered by the **`fe_dec_setround`** function; otherwise if **`FLT_RADIX`** is 10, whether the **`fesetround`** and **`fe_dec_setround`** functions alter the rounding direction of both generic floating type and decimal floating type operations is implementation defined.

Returns

The **`fe_dec_setround`** function returns a zero value if and only if the argument is equal to a rounding direction macro (that is, if and only if the requested rounding direction was established).

9.3 Decimal mathematics <math.h>

The list of elementary functions specified in the mathematics library is extended to handle decimal floating-point types. These include functions specified in 7.12.4, 7.12.5, 7.12.6, 7.12.7, 7.12.8, 7.12.9, 7.12.10, 7.12.11, 7.12.12, and 7.12.13. The macros **`HUGE_VAL_D32`**, **`HUGE_VAL_D64`**, **`HUGE_VAL_D128`**, **`DEC_INFINITY`** and **`DEC_NAN`** are defined to help using these functions. With the exception of **`sqrt`**, **`max`**, and **`min`**, the accuracy of the decimal floating-point results is implementation-defined. The implementation may state that the accuracy is unknown. All classification macros specified in C99 7.12.3 are also extended to handle decimal floating-point types. The same applies to all comparison macros specified in 7.12.14.

The names of the functions are derived by adding suffixes d32, d64 and d128 to the **`double`** version of the function name.

Suggested changes to C99:

Add after 7.12 paragraph 2.

7.12

[2a] The types

```
_Decimal32_t
_Decimal64_t
```

are decimal floating types at least as wide as **`_Decimal32`** and **`_Decimal64`**, respectively, and such that **`_Decimal64_t`** is at least as wide as **`_Decimal32_t`**. If **`DEC_EVAL_METHOD`** equals 0, **`_Decimal32_t`** and **`_Decimal64_t`** are **`_Decimal32`** and **`_Decimal64`**, respectively; if **`DEC_EVAL_METHOD`** equals 1, they are both **`_Decimal64`**; if **`DEC_EVAL_METHOD`** equals 2, they are both **`_Decimal128`**; and for other values of **`DEC_EVAL_METHOD`**, they are otherwise implementation-defined.

Add at the end of 7.12 paragraph 3 the following macros.

7.12

[3] The macro

`HUGE_VAL_D64`

expands to a constant expression of type **`_Decimal64`** representing infinity. The macros

`HUGE_VAL_D32`
`HUGE_VAL_D128`

are respectively **`_Decimal32`** and **`_Decimal128`** analogs of **`HUGE_VAL_D64`**.

Add at the end of 7.12 paragraph 4 the following macro.

7.12

[4] The macro

`DEC_INFINITY`

expands to a constant expression of type **`_Decimal32`** representing infinity.

Add at the end of 7.12 paragraph 5 the following macro.

7.12

[5] The macro

`DEC_NAN`

expands to quiet decimal floating NaN for the type **`_Decimal32`**.

Add at the end of 7.12 paragraph 7 the following macro.

7.12

[7] The macro

`FP_FAST_FMAD32`
`FP_FAST_FMAD64`
`FP_FAST_FMAD128`

are, respectively, **`_Decimal32`**, **`_Decimal64`** and **`_Decimal128`** analogs of **`FP_FAST_FMA`**.

Suggested changes to C99:

Add the following list of function prototypes to the synopsis of the respective subclauses:

7.12.4 Trigonometric functions

```
_Decimal64 acosd64(_Decimal64 x);
_Decimal32 acosd32(_Decimal32 x);
_Decimal128 acosd128(_Decimal128 x);

_Decimal64 asind64(_Decimal64 x);
_Decimal32 asind32(_Decimal32 x);
_Decimal128 asind128(_Decimal128 x);

_Decimal64 atand64(_Decimal64 x);
_Decimal32 atand32(_Decimal32 x);
_Decimal128 atand128(_Decimal128 x);

_Decimal64 atan2d64(_Decimal64 y, _Decimal64 x);
_Decimal32 atan2d32(_Decimal32 y, _Decimal32 x);
_Decimal128 atan2d128(_Decimal128 y, _Decimal128 x);

_Decimal64 cosd64(_Decimal64 x);
_Decimal32 cosd32(_Decimal32 x);
_Decimal128 cosd128(_Decimal128 x);

_Decimal64 sind64(_Decimal64 x);
_Decimal32 sind32(_Decimal32 x);
_Decimal128 sind128(_Decimal128 x);

_Decimal64 tand64(_Decimal64 x);
_Decimal32 tand32(_Decimal32 x);
_Decimal128 tand128(_Decimal128 x);
```

7.12.5 Hyperbolic functions

```
_Decimal64 acoshd64(_Decimal64 x);
_Decimal32 acoshd32(_Decimal32 x);
_Decimal128 acoshd128(_Decimal128 x);

_Decimal64 asinhd64(_Decimal64 x);
_Decimal32 asinhd32(_Decimal32 x);
_Decimal128 asinhd128(_Decimal128 x);

_Decimal64 atanhd64(_Decimal64 x);
_Decimal32 atanhd32(_Decimal32 x);
```

```
_Decimal128 atanhd128(_Decimal128 x);

_Decimal64 coshd64(_Decimal64 x);
_Decimal32 coshd32(_Decimal32 x);
_Decimal128 coshd128(_Decimal128 x);

_Decimal64 sinhd64(_Decimal64 x);
_Decimal32 sinhd32(_Decimal32 x);
_Decimal128 sinhd128(_Decimal128 x);

_Decimal64 tanhd64(_Decimal64 x);
_Decimal32 tanhd32(_Decimal32 x);
_Decimal128 tanhd128(_Decimal128 x);
```

7.12.6 Exponential and logarithmic functions

```
_Decimal64 expd64(_Decimal64 x);
_Decimal32 expd32(_Decimal32 x);
_Decimal128 expd128(_Decimal128 x);

_Decimal64 exp2d64(_Decimal64 x);
_Decimal32 exp2d32(_Decimal32 x);
_Decimal128 exp2d128(_Decimal128 x);

_Decimal64 expm1d64(_Decimal64 x);
_Decimal32 expm1d32(_Decimal32 x);
_Decimal128 expm1d128(_Decimal128 x);

_Decimal64 frexpd64(_Decimal64 value, int *exp);
_Decimal32 frexpd32(_Decimal32 value, int *exp);
_Decimal128 frexpd128(_Decimal128 value, int *exp);

int ilogbd64(_Decimal64 x);
int ilogbd32(_Decimal32 x);
int ilogbd128(_Decimal128 x);

_Decimal64 ldexpd64(_Decimal64 x, int exp);
_Decimal32 ldexpd32(_Decimal32 x, int exp);
_Decimal128 ldexpd128(_Decimal128 x, int exp);

_Decimal64 logd64(_Decimal64 x);
_Decimal32 logd32(_Decimal32 x);
_Decimal128 logd128(_Decimal128 x);

_Decimal64 log10d64(_Decimal64 x);
_Decimal32 log10d32(_Decimal32 x);
```

```
_Decimal128 log10d128(_Decimal128 x);

_Decimal64 log1pd64(_Decimal64 x);
_Decimal32 log1pd32(_Decimal32 x);
_Decimal128 log1pd128(_Decimal128 x);

_Decimal64 log2d64(_Decimal64 x);
_Decimal32 log2d32(_Decimal32 x);
_Decimal128 log2d128(_Decimal128 x);

_Decimal64 logbd64(_Decimal64 x);
_Decimal32 logbd32(_Decimal32 x);
_Decimal128 logbd128(_Decimal128 x);

_Decimal64 modd64(_Decimal64 value, _Decimal64 *iptr);
_Decimal32 modfd32(_Decimal32 value, _Decimal32 *iptr);
_Decimal128 modfd128(_Decimal128 value, _Decimal128 *iptr);

_Decimal64 scalbnd64(_Decimal64 x, int n);
_Decimal32 scalbnd32(_Decimal32 x, int n);
_Decimal128 scalbnd128(_Decimal128 x, int n);

_Decimal64 scalblnd64(_Decimal64 x, long int n);
_Decimal32 scalblnd32(_Decimal32 x, long int n);
_Decimal128 scalblnd128(_Decimal128 x, long int n);
```

7.12.7 Power and absolute-value functions

```
_Decimal64 cbrtd64(_Decimal64 x);
_Decimal32 cbrtd32(_Decimal32 x);
_Decimal128 cbrtd128(_Decimal128 x);

_Decimal64 fabsd64(_Decimal64 x);
_Decimal32 fabsd32(_Decimal32 x);
_Decimal128 fabsd128(_Decimal128 x);

_Decimal64 hypotd64(_Decimal64 x, _Decimal64 y);
_Decimal32 hypotd32(_Decimal32 x, _Decimal32 y);
_Decimal128 hypotd128(_Decimal128 x, _Decimal128 y);

_Decimal64 powd64(_Decimal64 x, _Decimal64 y);
_Decimal32 powd32(_Decimal32 x, _Decimal32 y);
_Decimal128 powd128(_Decimal128 x, _Decimal128 y);

_Decimal64 sqrtd64(_Decimal64 x);
_Decimal32 sqrtd32(_Decimal32 x);
```

```
_Decimal128 sqrtd128(_Decimal128 x);
```

7.12.8 Error and gamma functions

```
_Decimal64 erfd64(_Decimal64 x);
_Decimal32 erfd32(_Decimal32 x);
_Decimal128 erfd128(_Decimal128 x);

_Decimal64 erfcd64(_Decimal64 x);
_Decimal32 erfcd32(_Decimal32 x);
_Decimal128 erfcd128(_Decimal128 x);

_Decimal64 lgammad64(_Decimal64 x);
_Decimal32 lgammad32(_Decimal32 x);
_Decimal128 lgammad128(_Decimal128 x);

_Decimal64 tgammad64(_Decimal64 x);
_Decimal32 tgammad32(_Decimal32 x);
_Decimal128 tgammad128(_Decimal128 x);
```

7.12.9 Nearest integer functions

```
_Decimal64 ceild64(_Decimal64 x);
_Decimal32 ceild32(_Decimal32 x);
_Decimal128 ceild128(_Decimal128 x);

_Decimal64 floord64(_Decimal64 x);
_Decimal32 floord32(_Decimal32 x);
_Decimal128 floord128(_Decimal128 x);

_Decimal64 nearbyintd64(_Decimal64 x);
_Decimal32 nearbyintd32(_Decimal32 x);
_Decimal128 nearbyintd128(_Decimal128 x);

_Decimal64 rintd64(_Decimal64 x);
_Decimal32 rintd32(_Decimal32 x);
_Decimal128 rintd128(_Decimal128 x);

long int lrintd64(_Decimal64 x);
long int lrintd32(_Decimal32 x);
long int lrintd128(_Decimal128 x);

long long int llrintd64(_Decimal64 x);
long long int llrintd32(_Decimal32 x);
long long int llrintd128(_Decimal128 x);
```

```
_Decimal64 roundd64(_Decimal64 x);
_Decimal32 roundd32(_Decimal32 x);
_Decimal128 roundd128(_Decimal128 x);

long int lroundd64(_Decimal64 x);
long int lroundd32(_Decimal32 x);
long int lroundd128(_Decimal128 x);

long long int llroundd64(_Decimal64 x);
long long int llroundd32(_Decimal32 x);
long long int llroundd128(_Decimal128 x);

_Decimal64 truncd64(_Decimal64 x);
_Decimal32 truncd32(_Decimal32 x);
_Decimal128 truncd128(_Decimal128 x);
```

7.12.10 Remainder functions

```
_Decimal64 fmodd64(_Decimal64 x, _Decimal64 y);
_Decimal32 fmodd32(_Decimal32 x, _Decimal32 y);
_Decimal128 fmodd128(_Decimal128 x, _Decimal128 y);

_Decimal64 remainderd64(_Decimal64 x, _Decimal64 y);
_Decimal32 remainderd32(_Decimal32 x, _Decimal32 y);
_Decimal128 remainderd128(_Decimal128 x, _Decimal128 y);

_Decimal64 remquod64(_Decimal64 x, _Decimal64 y, int *quo);
_Decimal32 remquod32(_Decimal32 x, _Decimal32 y, int *quo);
_Decimal128 remquod128(_Decimal128 x, _Decimal128 y, int *quo);
```

7.12.11 Manipulation functions

```
_Decimal64 copysignd64(_Decimal64 x, _Decimal64 y);
_Decimal32 copysignd32(_Decimal32 x, _Decimal32 y);
_Decimal128 copysignd128(_Decimal128 x, _Decimal128 y);

_Decimal64 nand64(const char *tagp);
_Decimal32 nand32(const char *tagp);
_Decimal128 nand128(const char *tagp);

_Decimal64 nextafterd64(_Decimal64 x, _Decimal64 y);
_Decimal32 nextafterd32(_Decimal32 x, _Decimal32 y);
_Decimal128 nextafterd128(_Decimal128 x, _Decimal128 y);

_Decimal64 nexttowardd64(_Decimal64 x, _Decimal128 y);
_Decimal32 nexttowardd32(_Decimal32 x, _Decimal128 y);
```

```
  _Decimal128 nexttowardd128(_Decimal128 x, _Decimal128 y);

7.12.12  Maximum, minimum, and positive difference functions

  _Decimal64 fdimd64(_Decimal64 x, _Decimal64 y);
  _Decimal32 fdimd32(_Decimal32 x, _Decimal32 y);
  _Decimal128 fdimd128(_Decimal128 x, _Decimal128 y);

  _Decimal64 fmaxd64(_Decimal64 x, _Decimal64 y);
  _Decimal32 fmaxd32(_Decimal32 x, _Decimal32 y);
  _Decimal128 fmaxd128(_Decimal128 x, _Decimal128 y);

  _Decimal64 fmind64(_Decimal64 x, _Decimal64 y);
  _Decimal32 fmind32(_Decimal32 x, _Decimal32 y);
  _Decimal128 fmind128(_Decimal128 x, _Decimal128 y);

7.12.13  Floating multiply-add

  _Decimal64 fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
  _Decimal32 fmad32(_Decimal32 x, _Decimal32 y, _Decimal32 z);
  _Decimal128 fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
```

Add to the end of 7.12.14 paragraph 1:

[1] … If either argument has decimal floating type, the other argument shall have decimal floating type as well.

9.4 New <math.h> functions

The following are new functions added to **`<math.h>`**.

Suggested addition to C99:

7.12.11.5 The quantize functions

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <math.h>
_Decimal32  quantized32 (_Decimal32 x,  _Decimal32 y);
_Decimal64  quantized64 (_Decimal64 x,  _Decimal64 y);
_Decimal128 quantized128(_Decimal128 x, _Decimal128 y);
```

Description

The quantize functions set the exponent of argument x to the exponent of argument y, while attempting to keep the value the same. If the exponent is being increased, the value shall be correctly rounded according to the current rounding mode; if the result does not have the same value as x, the "inexact" floating-point exception shall be raised. If the exponent is being decreased and the significand of the result has more digits than the type would allow, the result is NaN and the "invalid" floating-point exception shall be raised. If one or both operands are NaN the result is NaN. Otherwise if only one operand is infinity, the result is NaN and the "invalid" floating-point exception shall be raised. If both operands are infinity, the result is **`DEC_INFINITY`** and the sign is the same as x. The quantize functions do not signal underflow. Whether the quantize functions signal overflow is implementation-defined.

Returns

The quantize functions return the number which is equal in value (except for any rounding) and sign to x, and which has an exponent set to be equal to the exponent of y.

7.12.11.6 The samequantum functions

Synopsis

```
#define __STDC_WANT_DEC_FP__
#include <math.h>
_Bool  samequantumd32  (_Decimal32 x,  _Decimal32 y);
_Bool  samequantumd64  (_Decimal64 x,  _Decimal64 y);
_Bool  samequantumd128 (_Decimal128 x, _Decimal128 y);
```

Description

The samequantum functions determine if the representation exponents of the x and y are the same. If both x and y are NaN, or infinity, they have the same representation exponents; if exactly one operand is infinity or exactly one operand is NaN, they do not have the same representation exponents. The samequantum functions raise no exception.

Returns

The samequantum functions return **`true`** when x and y have the same representation exponents, **`false`** otherwise.

9.5 Formatted input/output specifiers

Suggested changes to C99:

Add the following to 7.19.6.1 paragraph 7, to 7.19.6.2 paragraph 11, to 7.24.2.1 paragraph 7, and to 7.24.2.2 paragraph 11:

`H` Specifies that a following e, E, f, F, g, or G conversion specifier applies to a **`_Decimal32`** argument.

`D` Specifies that a following e, E, f, F, g, or G conversion specifier applies to a **`_Decimal64`** argument.

`DD` Specifies that a following e, E, f, F, g, or G conversion specifier applies to a **`_Decimal128`** argument.

Change all occurrences of:

A **`double`** argument representing …

in the descriptions for the **`e`**, **`E`**, **`f`**, **`F`**, **`g`**, and **`G`** conversion specifiers in 7.19.6.1 paragraph 8 and 7.24.2.1 paragraph 8 to:

A **`double`** or decimal floating type argument representing …

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

The specifications of these functions are similar to those of **`strtod`**, **`strtof`**, and **`strtold`** as defined in C99 7.20.1.3. These functions are declared in **`<stdlib.h>`**.

Suggested addition to C99:

7.20.1.5 The strtod32, strtod64, and strtod128 functions

Synopsis

[#1]
```
#define __STDC_WANT_DEC_FP__
#include <stdlib.h>
_Decimal32  strtod32 (const char * restrict nptr, char ** restrict endptr);
_Decimal64  strtod64 (const char * restrict nptr, char ** restrict endptr);
_Decimal128 strtod128(const char * restrict nptr, char ** restrict endptr);
```

Description

[#2] The **`strtod32`**, **`strtod64`**, and **`strtod128`** functions convert the initial portion of the string pointed to by **`nptr`** to **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **`isspace`** function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

[#3] The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2;
- **`INF`** or **`INFINITY`**, ignoring case
- **`NAN`** or **`NAN`**(*n-char-sequence*$_{opt}$), ignoring case in the **`NAN`** part, where:

n-char-sequence:
    digit
    n-char-sequence digit

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

[#4] If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that it is not a hexadecimal floating number, that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A character sequence **`INF`** or **`INFINITY`** is interpreted as an infinity. A character sequence **`NAN`** or **`NAN`**(*n-char-sequence*$_{opt}$), is interpreted as a quiet NaN; the meaning of the n-char sequences is implementation-defined.[7] A pointer to the final string is stored in the object pointed to by **`endptr`**, provided that **`endptr`** is not a null pointer.

[#5] The converted value keeps the precision as the input if possible, and the value may be denormalized. Otherwise, rounding may occur and the value is converted according to F.5 [#3]. Rounding happens after any negation.

[#6] In other than the **`"C"`** locale, additional locale-specific subject sequence forms may be accepted.

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

Recommended practice

---

[7] An implementation may use the n-char sequence to determine extra information to be represented in the NaN's significand.

[#8] If the subject sequence has the decimal form and at most **`DEC128_MANT_DIG`** (defined in **`<float.h>`**) significant digits, the result should be correctly rounded. If the subject sequence *D* has more than **`DEC128_MANT_DIG`** significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having **`DEC128_MANT_DIG`** significant digits, such that the values of *L*, *D*, and *U* satisfy *L* <= *D* <= *U*. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

Returns

[#9] The functions return the converted value, if any. If no conversion could be performed, the value +0.E0dd is returned. If the correct value is outside the range of representable values, plus or minus **`HUGE_VAL_D64`**, **`HUGE_VAL_D32`**, or **`HUGE_VAL_D128`** is returned (according to the return type and sign of the value), and the value of the macro **`ERANGE`** is stored in **`errno`**. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **`errno`** acquires the value **`ERANGE`** is implementation-defined.

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

The specifications of these functions are similar to those of **`wcstod`**, **`wcstof`**, and **`wcstold`** as defined in C99 7.24.4.1.1. They are declared in **`<wchar.h>`**.

Suggested addition to C99:

7.24.4.1.3 The wcstod32, wcstod64, and wcstod128 functions

Synopsis

[#1]
```
#define __STDC_WANT_DEC_FP__
#include <wchar.h>
_Decimal32  wcstod32 (const wchar_t * restrict nptr, wchar_t ** restrict endptr);
_Decimal64  wcstod64 (const wchar_t * restrict nptr, wchar_t ** restrict endptr);
_Decimal128 wcstod128(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Description

[#2] The **`wcstod32`**, **`wcstod64`**, and **`wcstod128`** functions convert the initial portion of the wide string pointed to by **`nptr`** to **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the **`iswspace`** function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

[#3] The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined in 6.4.4.2;
- **`INF`** or **`INFINITY`**, ignoring case
- **`NAN`** or **`NAN`**(*n-wchar-sequence*$_{opt}$), ignoring case in the **`NAN`** part, where:

n-wchar-sequence:
digit
n-wchar-sequence digit

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

[#4] If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that it is not a hexadecimal floating number, that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide character sequence **`INF`** or **`INFINITY`** is interpreted as an infinity. A wide character sequence **`NAN`** or **`NAN`**(*n-wchar-sequence*$_{opt}$), is interpreted as a quiet NaN; the meaning of the n-wchar sequences is implementation-defined.[8] A pointer to the final wide string is stored in the object pointed to by **`endptr`**, provided that **`endptr`** is not a null pointer.

[#5] The converted value keeps the precision as the input if possible, and the value may be denormalized. Otherwise, rounding may occur and the value is converted according to F.5 [#3]. Rounding happens after any negation.

[#6] In other than the **`"C"`** locale, additional locale-specific subject sequence forms may be accepted.

---

[8] An implementation may use the n-char sequence to determine extra information to be represented in the NaN's significand.

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

Recommended practice

[#8] If the subject sequence has the decimal form and at most **`DEC128_MANT_DIG`** (defined in **`<float.h>`**) significant digits, the result should be correctly rounded. If the subject sequence *D* has more than **`DEC128_MANT_DIG`** significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having **`DEC128_MANT_DIG`** significant digits, such that the values of *L*, *D*, and *U* satisfy *L* <= *D* <= *U*. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

Returns

[#9] The functions return the converted value, if any. If no conversion could be performed, the value +0.E0dd is returned. If the correct value is outside the range of representable values, plus or minus **`HUGE_VAL_D64`**, **`HUGE_VAL_D32`**, or **`HUGE_VAL_D128`** is returned (according to the return type and sign of the value), and the value of the macro **`ERANGE`** is stored in **`errno`**. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **`errno`** acquires the value **`ERANGE`** is implementation-defined.

9.8 Type-generic macros <tgmath.h>

All new functions added to **`<math.h>`** are subjected to the same requirement as specified in C99 7.22 to provide support for *type-generic* macro expansion. When one of the arguments is a decimal floating type, use of the type-generic macro invokes a function whose parameters have the types determined as follows:

If there is more than one real floating type arguments, usual arithmetic conversions are applied to the real floating type arguments so that they have compatible types. Then,

- If any argument has type **`_Decimal128`**, the type determined is **`_Decimal128`**.
- Otherwise, if any argument has type **`_Decimal64`**, the type determined is **`_Decimal64`**.
- Otherwise, if any argument has type **`_Decimal32`**, the type determined is **`_Decimal32`**.
- Otherwise, the specification in C99 7.22 paragraph 3 applies.

EXAMPLE

```
pow(2,3.f)     // expands to pow(2.f,3.f), the double version of pow
pow(2,3DF)     // expands to powd32(2DF,3DF), the _Decimal32 version of pow
```

Index