TECHNICAL REPORT

ISO/IEC TR 24732

First edition
2009-01-15

Information technology — Programming languages, their environments and system software interfaces — Extension for the programming language C to support decimal floating-point arithmetic

Technologies de l'information — Langages de programmation, leur environnement et interfaces des logiciels de systèmes — Extension pour que le langage de programmation C supporte l'arithmétique du point flottant décimal

Reference number
ISO/IEC TR 24732:2009(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/IEC TR 24732, which is a Technical Report of type 2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

0 Introduction

0.1 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 in business applications, 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.

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.

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.

0.2 The arithmetic model

This Technical Report proposes to add support for the decimal formats for floating-point data specified in IEEE 754-2008, with operations and behaviors consistent with that specification. IEEE 754-2008 provides a unified specification for floating-point arithmetic using both binary radix and decimal radix representations. For binary radix, it specifies upwardly-compatible extensions to the previous version, IEEE 754-1985 (equivalently IEC 60559:1989, which is already supported by C99 implementations that define the macro **__STDC_IEC_559__**).

Those extensions are not considered in this proposal. Instead, this proposal confines itself to supporting the decimal radix formats, which are new in this revision of IEEE 754.

The model of floating-point arithmetic used in IEEE 754-2008 has three components:

— *data* - numbers and NaNs, which can be manipulated by, or be the results of, the operations it specifies

— *operations* - (addition, multiplication, conversions, etc) which can be carried out on data

— *context* - the status of operations (namely, exceptions flags), and controls to govern the results of operations (for example, rounding modes). (IEEE 754-2008 does not use a single term to refer to these collectively.)

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 data or context, except that it does define specific encodings that are to be used for data that may be exchanged between different implementations that conform to the specification.

From the perspective of the C language, *data* 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.

0.3 The formats

IEEE 754-2008 specifies *formats*, in terms of their radix, exponent range, and precision (significand length), to support general purpose decimal floating-point arithmetic. It specifies operation semantics in terms of values and abstract representations of data (format members). It also specifies bit-level encodings for formats intended for data interchange.

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.

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 IEEE 754-2008, 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.

Information technology — Programming languages, their environments and system software interfaces — Extension for the programming language C to support decimal floating-point arithmetic

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 intended to be consistent with the specification in IEEE 754-2008. Any conflict between the requirements described here and that specification is unintentional. This Technical Report defers to IEEE 754-2008.

The binary floating-point arithmetic as specified in IEEE 754-2008 is not considered in this Technical Report.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9899:1999, *Programming languages — C*

ISO/IEC 9899:1999/Cor.1:2001, *Programming languages — C — Technical Corrigendum 1*

ISO/IEC 9899:1999/Cor.2;2004, *Programming languages — C — Technical Corrigendum 2*

ISO/IEC TR 18037, *Programming languages — C — Extensions to support embedded processors*

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* [1)]

IEEE 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*

IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*

IEEE 854-1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*

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 For reference materials relating to IEEE 754-2008 see [3].

1) Previously designated IEC 559:1989.

3 Predefined macro name

The following macro name is conditionally defined by the implementation:

__STDC_DEC_FP__ The integer constant **200805L**, 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**[2)] 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.

2) The 32-bit format is a storage only format in IEEE 754-2008.

[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

Add the following paragraph after 6.5#8:

[8a] Expressions involving decimal floating-point operands are evaluated according to the semantics of IEEE 754-2008, including production of results with the preferred exponent as specified in IEEE 754-2008.

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 (0.2). The formats are specified in IEEE 754-2008 (0.3).

The three decimal formats defined in IEEE 754-2008 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 $(-1)^{sign} \times coefficient \times 10^{exponent}$. Refer to IEEE 754-2008 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})	97	385	6145
Minimum Exponent (E_{min})	-94	-382	-6142

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] Macros in **<float.h>** provide characteristics of floating types in terms of the model presented in 5.2.4.2.2. The prefixes **DEC32_**, **DEC64_**, and **DEC128_** denote the types **_Decimal32**, **_Decimal64**, and **_Decimal128** respectively.

[2] For decimal floating-point, it is often convenient to consider an alternate equivalent model where the significand is represented with integer rather than fraction digits: a floating-point number (x) is defined by the model

$$x = sb^{(e-p)} \sum_{k=1}^{p} f_k b^{(p-k)}$$

where s, b, e, p, and f_k are as defined in 5.2.4.2.2, and $b = 10$.

[3] The term *quantum exponent* refers to $q = e - p$ and *coefficient* to $c = f_1 f_2 ... f_p$, an integer between 0 and $b^p - 1$ inclusive. Thus, $x = s * c * b^q$ is represented by the triple of integers *(s, c, q)*.

[4] For binary floating-point following IEC 60559 (and IEEE 754-2008), representations in the model described in 5.2.4.2.2 that have the same numerical value are indistinguishable in the arithmetic. However, for decimal floating-point, representations that have the same numerical value but different quantum exponents, e.g., (1, 10, -1) representing 1.0 and (1, 100, -2) representing 1.00, are distinguishable. To facilitate exact fixed-point calculation, standard decimal floating-point operations and functions have a *preferred quantum exponent*, as specified in IEEE 754-2008, which is determined by the quantum exponents of the operands if they have decimal floating-point types (or by specific rules for conversions from other types), and they produce a result with that preferred quantum exponent, or as close to it as possible within the limitations of the type. For example, the preferred quantum exponent for addition is the minimum of the quantum exponents of the operands. Hence (1, 123, - 2) + (1, 4000, -3) = (1, 5230, -3) or 1.23 + 4.000 = 5.230.

[5] 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.

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

- radix of exponent representation, *b(=10)*

  For the generic floating-point types, this value is implementation-defined and is specified by the macro **`FLT_RADIX`**. For the decimal floating-point types there is no corresponding macro, since the value 10 is an inherent property of the types. Wherever **`FLT_RADIX`** appears in a description of a function that has versions that operate on decimal floating-point types, it is noted that for the decimal floating-point versions the value used is implicitly 10, rather than **`FLT_RADIX`**.

- number of digits in the coefficient

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

- minimum exponent

`DEC32_MIN_EXP`	**`-94`**
`DEC64_MIN_EXP`	**`-382`**
`DEC128_MIN_EXP`	**`-6142`**

- maximum exponent

```
DEC32_MAX_EXP          97
DEC64_MAX_EXP          385
DEC128_MAX_EXP         6145
```

- 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 positive subnormal decimal floating number


```
DEC32_SUBNORMAL_MIN    0.000001E-95DF
DEC62_SUBNORMAL_MIN    0.000000000000001E-383DD
DEC128_SUBNORMAL_MIN   0.000000000000000000000000000000001E-6143DL
```


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 and the result of the conversion is unspecified.

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 with exceptions raised as specified in IEEE 754-2008.

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 to 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, 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 with exceptions raised as specified in IEEE 754-2008.

6.3 Conversions between decimal floating and complex

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, **`return`** statement, and in argument passing involving prototyped functions.

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`**.