WG14 N3082

Title:	**CFP response to NB comments and N3071 - update**
Author, affiliation:	**C FP group**
Date:	**2023-01-19**
Reference:	**N3054, N3067, N3071, N3073, N3081**

This document updates N3081, "CFP response to NB comments and N3071". It includes: (1) response to N3073, "Updated SCC Comments for ISO/IEC CD 9899, C", (2) changes intended to address WG14 email comments, and (3) clarifications. Changes from N3081 are highlighted.

Below are the comments CFP has reviewed and our suggested response. (Editorial comments seen as clear and uncontroversial are not listed.) Also, at the end, are suggested responses to the issues raised in N3071.

Agree. Accept proposed change.

US 5-018
US 6-023
GB-063
GB-127
GB-147
GB-149
GB-152
US 39-155
GB-156
GB-157
GB-163
US 40-166
GB-173
US 56-187
US 57-189
GB-220 and duplicate US 63-216
GB-229
GB-230
GB-267
GB-268 Note also F.10.3.7 uses **`exp`** while 7.12.6.7 uses **`p`** for the same argument.
GB-269
GB-271 Wording more style-consistent than similar US 68-270.
GB-276
US 67-278

Disagree. No change needed.

GB-153
If imaginary types are not supported the formula gives Inf + (0,1)*zero = Inf + (zero,zero) = (Inf, zero). The normative formula defines **`proj`** when **`z`** has an infinite part.

GB-164

The qualification for default rounding in C18 was intended to apply to returning **HUGE_VAL**, not to reporting an error. If overflow returns a finite number, it's even more important to report it. Change not desired.

However, if preserving the default rounding qualification is deemed too valuable, the qualification could be restored for errno but not floating-point exceptions, as follows:

In 7.12.1 #5, insert before "and the integer expression **math_errhandling & MATH_ERRNO** is nonzero" the words "and default rounding is in effect".

The corresponding change for **MATHERR_EXCEPT** should not be made. Not raising an "overflow" floating-point exception when overflow occurs because a non-default rounding direction is in effect would be inconsistent with IEC 60559. In this regard, the following clarification should be made (if the change above is made):

In 7.12.1 #5, insert at the end of the paragraph, after "the 'overflow' floating-point exception is raised" the parenthetical remark "(regardless of whether default rounding is in effect)".

US 43-170

Both footnotes are implied by the Description ("return the maximum/minimum numeric value of their argument"), so are not normative. No change needed.

CA-N3073-006

Recommend no change.

The primary objection raised with the CD specification is that it requires some current implementations with double-double formats for **long double** to change macro values. This requirement comes from the CD's clarification that all normalized numbers, i.e. all numbers with a given precision (*p*) within a given exponent range (*emin* through *emax*), must be represented.

The last Proposed Technical Corrigendum (April 2017, too late for C18) for DR #467 included a change to clarify that a type must be able to represent all normalized numbers. This property is important for users, and, we believe, has been generally assumed. For example, users might reasonably expect that

```
#include <float.h>
#ifdef LDBL_MANT_DIG >= 105
long double x = 0x1.1234567890123456789012345Fp0L;
#endif
```

would store the exact value of its initializer in **x**.

The change proposed in CA-N3073-006 qualifies the definition of the ***_MANT_DIG** (and other) macros with "in relation to implementation-defined model parameters not subject to the restriction that the floating type is able to represent all normalized floating-point numbers". This change would invalidate the code (above), and the general assumption about the macros.

The change proposed in CA-N3073-006 also introduces new macros that are equivalent to the ones in the CD now. This would be a cost to users in needing to understand more specification, the differences between old and new macros, and which ones they should use. We expect almost all users would want the ones in the CD and (with the proposed change) they would need to modify their code to get them.

We appreciate the general concern that there might be unknown user consequences of interface changes, that existing program might get different results. Here, an implementation would need to change its macro definitions only if (1) the values are not currently consistent with respect to some fixed *p*, *emax*, and *emin*, or (2) the implementation doesn't represent all normalized numbers for its chosen *p*, *emax*, and *emin*. We are not aware of meaningful use of the macros that the required implementation changes would break.

CA-N3073-006 proposes changes and new macros for the ***_MANT_DIG** and ***_EPSILON** macros. But other macros are similarly dependent on the representability of normalized numbers. The proposed change (if generally accepted) would need to be expanded.

Generally agree. Modify/complete proposed change.

GB-007

Agree with first two changes. In 6.7.1 change "single precision" to "**float**" and change "32-bit single-precision IEC 60559" to "IEC 60559 binary32".

US 26-075

In 6.7.1 #5 after the second sentence, insert "An initializer of floating type shall be evaluated with the translation-time floating-point environment."

In the comment example the **fesetround** call would not affect the initialization of **h**.

GB-151

Change both footnotes to: "For a complex variable **z**, **z** and **CMPLX(creal(z), cimag(z))** are equivalent expressions. If imaginary types are supported, **z** and **creal(z) + cimag(z)*I** are equivalent expressions."

US 42-169

Add the Returns paragraph as suggested. Similar 7.24.1.3 has a Returns section.

US 71-275

Delete the entire line. It was a holdover from earlier version of IEC 60559.

GB-279

CFP response (above) to US-75 makes **constexpr** initialization done with the translation-time floating-point environment, like for static and thread storage duration. It would be independent of the **FENV_ACCESS** pragma but would be affected by the **FENV_ROUND** and **FENV_DEC_ROUND** pragmas.

In F.8.4 #1 insert after "for an object that has static or thread storage duration" the words "or that is declared with storage-class specifier **constexpr**".

In F.8.4 #2 in the example before "**float w[] =** …" insert "**constexpr double v = 0.0/0.0; // does not raise an exception**".

In F.8.4 #3 change "For the static initialization" to "For the static and **constexpr** initializations".

In F.8.5 #1 insert after "of objects that have static or thread storage duration" the words "or that are declared with storage-class specifier **constexpr**".

In F.8.5 #2 in the example before "**float u[] =** …" insert "**constexpr float t = (float)1.1e75; // does not raise an exception**".

In F.8.5 #3 change "The static initialization of **v** raises no (execution-time) floating-point exceptions because its computation is done at translation time" to "The **constexpr** initialization of **t** and the static initialization of **v** raise no (execution-time) floating-point exceptions because their computation is done at translation time".

A related problem is that the CD specification for initialization with signaling NaN macros doesn't cover **constexpr**. This could be fixed with the following change:

In 5.2.4.2.2 #21 replace "If an optional unary **+** or **-** operator followed by a signaling NaN macro is used as the initializer for initializing an object of the same type that has static or thread storage duration …" with "If an optional unary **+** or **-** operator followed by a signaling NaN macro is used as an initializer that must be evaluated at translation time …".

GB-286

The **strtof***N* functions are not like the **strfromf***N* functions whose wide character versions can easily be obtained from other functions (as shown in an example). Rather than add them at this late date …

In 7.33.20 after paragraph 1 add: "Functions with potentially reserved identifiers **wcstof***N* and **wcstod***N* are intended to be wide character analogs of the **strtof***N* and **strtod***N* functions."

GB-287

The suggested change seems too large to do now. Instead …

In 7.24.1.6#4 change the bullet "It is not a hexadecimal floating number" to "Whether the subject sequence may be a hexadecimal floating number is implementation-defined."

In 7.24.1.6, before the Returns section, insert:

> **Recommended practice**
>
> Rounding for hexadecimal input should follow the method in H.12.2.

In 7.24.1.6#4, in the "**0x1.8p+4**" example, before "(+1, 0, 0), …" insert "If hexadecimal input is accepted, (+1, 24, 0). If hexadecimal input is not accepted, "

GB-288

To H.12.2 #3 append: "The preferred quantum exponent for the result is 0 if the hexadecimal number is exactly represented in the decimal type; the preferred quantum exponent for the result is the least possible if the hexadecimal number is not exactly represented in the decimal type."

About N3071

The following proposed changes are intended to address the missing (or ambiguous) specification pointed out in N3071.

To 6.7.1 #5, append: "If the object declared has real floating type, the initializer shall have integer or real floating type. If the object declared has imaginary type, the initializer shall have imaginary type. If the initializer has decimal floating type, the object declared shall have decimal floating type and the conversion shall preserve the quantum of the initializer. If the initializer has real type and a signaling NaN value, the unqualified versions of the type of the initializer and the corresponding real type of the object declared shall be compatible."

After 6.7.1 #17 (EXAMPLE 3), insert

EXAMPLE 4 This example illustrates **constexpr** initializations involving different type domains, decimal and non-decimal floating types, NaNs and infinities, and quanta in decimal floating types.

```
#include <float.h>
#include <complex.h>
constexpr float _Complex fc1 = 1.0;                  // ok
constexpr float _Complex fc2 = 0.1;                  // constraint violation, unless double
                                                     //   has the same precision as float
                                                     //   and is evaluated with the same
                                                     //   precision
constexpr float _Complex fc3 = 3*I;                  // ok
constexpr double d1 = (double _Complex)1.0;          // constraint violation
constexpr double d2 = (double _Imaginary)0.0;        // constraint violation
constexpr float f1 = (long double)INFINITY;          // ok
constexpr float f2 = (long double)NAN;               // ok, quiet NaNs in real floating
                                                     //   types are considered the same
                                                     //   value, regardless of payloads
constexpr double d3 = DBL_SNAN;                      // ok
constexpr double d4 = FLT_SNAN;                      // constraint violation, even if float
                                                     //   and double have the same format
constexpr double _Complex dc1 = DBL_SNAN;            // ok
constexpr double _Complex dc2 = CMPLX(DBL_SNAN, 0.); // ok
constexpr double _Complex dc3 = CMPLX(0., DBL_SNAN); // ok
constexpr _Decimal32 d321 = 1.0;                     // ok
constexpr _Decimal32 d322 = 1;                       // ok
constexpr _Decimal32 d323 = INFINITY;                // ok
constexpr _Decimal32 d324 = NAN;                     // ok
constexpr _Decimal64 d641 = DEC64_SNAN;              // ok
constexpr _Decimal64 d642 = DEC32_SNAN;              // constraint violation
constexpr float f3 = 1.DF;                           // constraint violation
constexpr float f4 = DEC_INFINITY;                   // constraint violation
constexpr double d5 = DEC_NAN;                       // constraint violation
constexpr _Decimal32 d325 = DEC64_TRUE _MIN * 0;     // constraint violation, quantum not
                                                     //   preserved

#ifdef __STDC_IEC_60559_COMPLEX__
constexpr double d6 = (double _Imaginary)0.0;        // constraint violation
constexpr double _Imaginary di1 = 0.0*I;             // ok
constexpr double _Imaginary di2 = 0.0;               // constraint violation
#endif
```

The following is an additional editorial comment, with a proposal:

6.7.1 #14 (NOTE 2) uses “mantissa” which is used nowhere else in the document. Suggest changing “a diagnostic is required if a truncation of the mantissa occurs” to “a diagnostic is required if a truncation of the excess precision changes the value”.