Draft DRs for TS 18661

WG 14 N2094

2016-10-06

This document updates N2077. The only changes are (1) different type faces, (2) the addition of the change to the example in the suggest technical corrigendum in DDR #9.

DDR #1

==========

Reference Document: C11

Subject: Ambiguous specification for **FLT_EVAL_METHOD**

Summary

5.2.4.2.2#9:

> Except for assignment and cast (which remove all extra range and precision), the values yielded by operators with floating operands and values subject to the usual arithmetic conversions and of 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 **`FLT_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 **`float`** and **`double`** to the range and precision of the **`double`** type, evaluate **`long double`** operations and constants to the range and precision of the **`long double`** type;
>
> **`2`** evaluate all operations and constants to the range and precision of the **`long double`** type.
>
> All other negative values for **`FLT_EVAL_METHOD`** characterize implementation-defined behavior

Do the words:

> the values yielded by operators with floating operands and values subject to the usual arithmetic conversions

in the first sentence mean the same as:

> Interpretation 1: the values yielded by operators with: (a) floating operands and (b) values subject to the usual arithmetic conversions

or:

> Interpretation 2: (a) the values yielded by operators with floating operands and (b) the values subject to the usual arithmetic conversions?

Interpretation 2 is problematic because the evaluation methods pertain only to operators that return a value of floating type, not to, for example, the relational operators with floating operands. Nor do they apply to all values subject to the usual arithmetic conversions, and so (b) doesn't add anything. Thus, reasonableness suggests Interpretation 1. However, the mention of assignment and cast (which are not subject to the usual arithmetic conversions) suggests Interpretation 2.

Interpretation 2, unlike Interpretation 1, implies that values yielded by unary operators are widened to the evaluation format. In some cases whether a unary operator is widened matters. Widening a signaling NaN operand raises the "invalid" floating-point exception. Widening an operand with a non-canonical encoding canonicalizes the encoding.

The IEC 60559 *copy* and *negate* operations are bit manipulation operations that affect at most the sign. C operations bound to these IEC 60559 operations are expected to behave accordingly, but won't if they entail widening.

Widening unary operators would introduce conversions that might affect performance but which have no benefit.

According to personal notes, this issue came up at the WG14 meeting in Chicago in 2013, but was not resolved and did not result in an action item.

Recently, this issue came up again as underlying the issue raised by Joseph Myers in email SC22WG14.14278:

> Suppose that with an implementation of C11 + TS 18661-1, that defines **`FLT_EVAL_METHOD`** to **`2`**, you have:
>
> ```
> static volatile double x = SNAN;
> (void) x;
> ```
>
> Suppose also that the implementation defines the "**`(void) x;`**" statement to constitute an access to volatile-qualified **`x`**.
>
> May the implementation define that access to convert **`x`** from the format of **`double`** to the format of **`long double`**, with greater range and precision, that format being used to represent **`double`** operands in accordance with the setting of **`FLT_EVAL_METHOD`**, and thereby to raise the "invalid" exception?
>
> That is, may a convertFormat operation be applied as part of

lvalue-to-rvalue conversion where **FLT_EVAL_METHOD** implies that a wider evaluation format is in use?

Even without signaling NaNs, the issue can apply to the case of exact underflow, which can be detected using pragmas from TS 18661-5, if the wider format has extra precision but not extra range and so exact underflow occurs on converting a subnormal value to the wider format.

The following suggested Technical Corrigendum is intended to clarify the wording in favor of Interpretation 1, which excludes widening unary operators to the evaluation format.

Suggested Technical Corrigendum

In 5.2.4.2.2#9, replace:

> Except for assignment and cast (which remove all extra range and precision), the values yielded by operators with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type.

with:

> The values of floating type yielded by operators subject to the usual arithmetic conversions and the values of floating constants are evaluated to a format whose range and precision may be greater than required by the type. In all cases, assignment and cast remove all extra range and precision.

DDR #2
==========

Reference Document: C11

Subject: Can **DECIMAL_DIG** be larger than necessary?

Summary

This is about the issue raised by Joseph Myers in email SC22WG14.14285:

> C11 defines **DECIMAL_DIG** as "number of decimal digits, *n*, such that any floating-point number in the widest supported floating type with p_{max} radix *b* digits can be rounded to a floating-point number with *n* decimal digits and back again without change to the value," and then gives a formula.
>
> Is it OK for the value of **DECIMAL_DIG** to be larger than given by the formula? Such a value would still seem to meet the textual description, though being suboptimal.
>
> This is an issue for implementing TS 18661-3 when that involves types wider than **long double**. In C11, "real floating type" means **float**, **double** or **long**

double (6.2.5#10) (and then those types plus the three complex types are defined to be the floating types). TS 18661-3 is supposed to be compatible with C11, so that an implementation can conform to both simultaneously. The definition of **DECIMAL_DIG** in TS 18661-3 covers all supported floating types and non-arithmetic encodings. And that's not conditional on **__STDC_WANT_IEC_60559_TYPES_EXT__**. So in an implementation of TS 18661-3 that supports **_Float128**, **DECIMAL_DIG** must be big enough for **_Float128**, even if **__STDC_WANT_IEC_60559_TYPES_EXT__** is not defined when **<float.h>** is included. And that's only compatible with C11 (if **long double** is narrower than **_Float128**) if C11 allows **DECIMAL_DIG** to be larger than given by the formula.

Agreed. The current specification for **DECIMAL_DIG** in TS 18661-3 is incompatible with C11, as explained.

The suggested Technical Corrigendum below allows **DECIMAL_DIG** to be larger than the value of the given formula. Thus an implementation that supports a floating type wider than **long double**, for example a wide type in TS 18661-3, could define **DECIMAL_DIG** to be large enough for its widest type and still conform as a C implementation without extensions.

Where **DECIMAL_DIG** is used to determine a sufficient number of digits, this change might lead to conversions with more digits than needed and with more digits than would have been used without the change. However, programs wishing the minimal sufficient number of digit are better served by the type-specific macros **FLT_DECIMAL_DIG**, etc.

We considered the alternative of changing TS 198661-3 to make **DECIMAL_DIG** dependent on **__STDC_WANT_IEC_60559_TYPES_EXT__**. But this could lead to errors resulting from separately compiled parts of a program using inconsistent values of **DECIMAL_DIG**.

Suggested Technical Corrigendum

In 5.2.4.2.2#11, change the bullet defining **DECIMAL_DIG** from:

— number of decimal digits, *n*, such that any floating-point number in the widest supported floating type with p_{max} radix *b* digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

< … formula … >

to:

— number of decimal digits, *n*, such that any floating-point number in the widest supported floating type with p_{max} radix *b* digits can be rounded to a floating-point number with n decimal digits and back again without change to the value, at least

< … formula … >

DDR #3

==========

Reference Document: Floating Point Extensions, Part 1

Subject: Is return of same type convertFormat or copy?

Summary

This is about the issue raised by Joseph Myers in email SC22WG14.14280:

> TS 18661-1 says "Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation is implementation-defined, even if **<fenv.h>** defines the macro **FE_SNANS_ALWAYS_SIGNAL** (F.2.1).".
>
> Does this apply to function return, where the return type of the function is the same as the type of the expression passed to the return statement and no wider evaluation format is in use - that is, may this act as either convertFormat or copy? C11 F.6 clearly envisages that such a return statement may do a conversion to the same type in the case of wider evaluation formats. But 6.8.6.4#3 only refers to conversions "If the expression has a type different from the return type of the function in which it appears".

The specification, from F.3#3, quoted above is incomplete in that it doesn't cover function returns, which are not assignments or conversions as if by assignment. As currently written, C11 + TS18661-1 might be read to exclude the possibility of using convertFormat in this case. A statement should be added to say that the implementation has the option to apply convertFormat to the return value. The change does not break existing implementations.

The effect of convertFormat would be that signaling NaNs would signal and noncanonical representations would be canonicalized. It is extremely unlikely that a program would depend on convertFormat not being used.

Suggested Technical Corrigendum

In Clause 8, to the text for C F.3#3:

> [3] Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation is implementation-defined, even if **<fenv.h>** defines the macro **FE_SNANS_ALWAYS_SIGNAL** (F.2.1).

append the sentence:

> If the return expression of a **return** statement is evaluated to the floating-point format of the return type, it is implementation-defined whether a convertFormat operation is applied to the result of the return expression."

At the end of Clause 8, add:

> In F.3#3, attach a footnote to the wording:
>
> > Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation
>
> where the footnote is:
>
> > *) Where the source and destination formats are the same, convertFormat operations differ from copy operations in that convertFormat operations raise the “invalid” floating-point exception on signaling NaN inputs and do not propagate non-canonical encodings.

DDR #4

==========

Reference Document: Floating Point Extensions, Part 1

Subject: **`fetestexceptflag`** and exceptions passed to **`fegetexceptflag`**

Summary

This is about the issue raised by Joseph Myers in email SC22WG14.14328:

> TS 18661-1 says, for **`fetestexceptflag`**, "The value of **`*flagp`** shall have been set by a previous call to **`fegetexceptflag`**.".
>
> This contrasts with the C11 wording for **`fesetexceptflag`**, "The value of **`*flagp`** shall have been set by a previous call to **`fegetexceptflag`** whose second argument represented at least those floating-point exceptions represented by the argument **`excepts`**.". So what happens if more exceptions are specified in the call to **`fetestexceptflag`** than were specified in the call to **`fegetexceptflag`**? Then **`fegetexceptflag`** may or may not have stored any meaningful representation of the state of the extra exceptions being tested.
>
> I think **`fetestexceptflag`** should have the same wording for this issue as **`fesetexceptflag`**: "whose second argument represented at least those floating-point exceptions represented by the argument **`excepts`**".

`fesetexceptflag` sets global state, typically a hardware register, whereas **`fetestexceptflag`** just reads a variable. It seems more important to avoid spurious data in the former.

Still, there’s no utility in testing spurious flag settings, and placing the same restrictions on **`fetestexceptflag`** as on **`fesetexceptflag`** might be less error prone.

Suggested Technical Corrigendum

In 15.2, in the new text for C 7.6.2.4a#2, change:

> The value of **`*flagp`** shall have been set by a previous call to **`fegetexceptflag`**.

to:

> The value of **`*flagp`** shall have been set by a previous call to **`fegetexceptflag`** whose second argument represented at least those floating-point exceptions represented by the argument **`excepts`**.

DDR #5

==========

Reference Document: Floating Point Extensions, Part 1

Subject: Editorial changes

Summary

In CFP email, Fred Tydeman noted:

> Searching for "infinite precision" in part 1, most of them have "(as if) to" before it. Except, **`ffma`**, **`ffmal`**, **`dfmal`** which is missing the "(as if)".

Right. In particular, all the functions that round result to narrower type have “(as if)”, except for the **`fma`** family.

Suggested Technical Corrigendum

In 14.5, in the new text for C 7.12.13a.5#2, insert “(as if)” before “to infinite precision”.

DDR #6

==========

Reference Document: Floating Point Extensions, Part 2

Subject: Editorial clarification about number digits in the coefficient

Summary

In 12.5, n is defined to be “the number of digits in the coefficient *c*”, where the decimal floating-point argument is represented by the triple (*s*, *c*, *q*). The intention is that *n* is the number of digits in the coefficient of the particular argument, i.e., the number of significant digits, not the maximum number of digits in the coefficient for the type. This might be

misread, particularly since 5.2.4.2.2a says

— number of digits in the coefficient

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

This part of 5.2.4.2.2a is in the context of characterizing the type, so clearly refers to the type and not any particular representation.

Suggested Technical Corrigendum

In 12.5, change:

> where *n* is the number of digits in the coefficient *c*

to:

> where *n* is the number of significant digits in the coefficient *c*

DDR #7

==========

Reference Document: Floating Point Extensions, Part 3

Subject: Missing specification for usual arithmetic conversions, tgmath

Summary

This is about the issue raised by Joseph Myers in email SC22WG14.14282:

> C11 specifies that the usual arithmetic conversions on the pair of types (**`long double`**, **`double`**) produces a result of type **`long double`**.
>
> Suppose **`long double`** and **`double`** have the same set of values. TS 18661-3 rewrites the rules for usual arithmetic conversions so that the case "if both operands are floating types and the sets of values of their corresponding real types are equivalent" prefers interchange types to standard types to extended types. But this leaves the case of (**`long double`**, **`double`**) unspecified as to which type is chosen, unlike in C11, as those are both standard types.
>
> I think this is a defect in TS 18661-3, and it should say that if both are standard types with the same set of values then **`long double`** is preferred to **`double`** which is preferred to **`float`**, as in C11.
>
> A similar issue could arise if two of the extended types have equivalent sets of values. I'm not aware of anything to prohibit that, although it seems less likely in

practice. I think the natural fix would be to say that **_Float128x** is preferred to **_Float64x** which is preferred to **_Float32x**.

I think such an issue would also arise for **<tgmath.h>** (if **_Float64x** and **_Float128x** have the same set of values, the choice doesn't seem to be specified). It also seems possible for the **<tgmath.h>** rules for purely floating-point arguments to produce a different result from the usual arithmetic conversions (consider the case where **_Float32x** is wider than **long double**, and **<tgmath.h>** chooses **long double**), and since rules that are the same in most cases but subtly different in obscure cases tend to be confusing, I wonder if it might be better to specify much simpler rules for **<tgmath.h>**: take the type resulting from the usual arithmetic conversions[*], where integer arguments are replaced by **_Decimal64** if there are any decimal arguments and **double** otherwise. (That's different from the present rules for e.g. (**_Float32x**, **int**), but it's a lot simpler, and seems unlikely in practice to choose a type with a different set of values from the present choice.)

[*] Meaningful for more than two arguments as long as the usual arithmetic conversions are commutative and associative as an operation on pairs of types.

Though substantive, the suggested change to the usual arithmetic conversions is consistent with the intention in TS 18661-3 to specify all the cases (except where neither format is a subset of the other and the formats are not the same). The missing cases were an oversight. The suggested preferences of **long double** over **double** over **float** and **_Float128x** over **_Float64x** over **_Float32x** are the obvious choices.

Joseph Myers notes that the **<tgmath.h>** specification is incomplete in the same way as the usual arithmetic conversions. He argues for simplifying the specification by referring to the usual arithmetic conversions specification, rather than mostly repeating it, as the current specification does. The suggested Technical Corrigendum below follows this new approach. Though a substantive change to TS 18661-3, the effects on implementations and users are expected to be minimal – worth the simplification.

The suggested Technical Corrigendum below also restores footnote number 62, which is lost in the current TS 18661-3.

Suggested Technical Corrigendum

In clause 8, change the replacement text for 6.3.1.8#1:

> If one operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.
>
> If both operands have floating types and neither of the sets of values of their corresponding real types is a subset of (or equivalent to) the other, the behavior is undefined.

Otherwise, if both operands are floating types and the sets of values of their corresponding real types are equivalent, then the following rules are applied:

> If both operands have the same corresponding real type, no further conversion is needed.
>
> Otherwise, if the corresponding real type of either operand is an interchange floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same interchange floating type.
>
> Otherwise, if the corresponding real type of either operand is a standard floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same standard floating type.

Otherwise, if both operands have floating types, the operand, whose set of values of its corresponding real type is a (proper) subset of the set of values of the corresponding real type of the other operand, is converted, without change of type domain, to a type with the corresponding real type of that other operand.

Otherwise, if one operand has a floating type, the other operand is converted to the corresponding real type of the operand of floating type.

Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:

> . . .

to:

If one operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

If both operands have floating types and neither of the sets of values of their corresponding real types is a subset of (or equivalent to) the other, the behavior is undefined.

If both operands have the same corresponding real type, no further conversion is needed.

Otherwise, if both operands are floating types and the sets of values of their corresponding real types are equivalent, then the following rules are applied:

> If the corresponding real type of either operand is an interchange floating type, the other operand is converted, without change of type

domain, to a type whose corresponding real type is that same interchange floating type.

Otherwise, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`long double`**.

Otherwise, if the corresponding real type of either operand is **`double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`double`**.

(All cases where **`float`** might have the same format as another type are covered above.)

Otherwise, if the corresponding real type of either operand is **`_Float128x`** or **`_Decimal128x`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`_Float128x`** or **`_Decimal128x`**, respectively.

Otherwise, if the corresponding real type of either operand is **`_Float64x`** or **`_Decimal64x`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`_Float64x`** or **`_Decimal64x`**, respectively.

Otherwise, if both operands have floating types, the operand, whose set of values of its corresponding real type is a (proper) subset of the set of values of the corresponding real type of the other operand, is converted, without change of type domain62), to a type with the corresponding real type of that other operand.

Otherwise, if one operand has a floating type, the other operand is converted to the corresponding real type of the operand of floating type.

Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:

. . .

In clause 15, replace:

In 7.25#3c, replace the bullets:

... bullets ...

with:

— If two arguments have floating types and neither of the sets of values of their corresponding real types is a subset of (or equivalent to) the other, the behavior is undefined.

— If any arguments for generic parameters have type **_Decimal***M* where *M* ≥ 64 or **_Decimal***N***x** where *N* ≥ 32, the type determined is the widest of the types of these arguments. If **_Decimal***M* and **_Decimal***N***x** are both widest types (with equivalent sets of values) of these arguments, the type determined is **_Decimal***M*.

— Otherwise, if any argument for generic parameters is of integer type and another argument for generic parameters has type **_Decimal32**, the type determined is **_Decimal64**.

— Otherwise, if any argument for generic parameters has type **_Decimal32**, the type determined is **_Decimal32**.

— Otherwise, if the corresponding real type of any argument for generic parameters has type **long double**, **_Float***M* where *M* ≥ 128, or **_Float***N***x** where *N* ≥ 64, the type determined is the widest of the corresponding real types of these arguments. If **_Float***M* and either **long double** or **_Float***N***x** are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is **_Float***M*. Otherwise, if **long double** and **_Float***N***x** are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is **long double**.

— Otherwise, if the corresponding real type of any argument for generic parameters has type **double**, **_Float64**, or **_Float32x**, the type determined is the widest of the corresponding real types of these arguments. If **_Float64** and either **double** or **_Float32x** are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is **_Float64**. Otherwise, if **double** and **_Float32x** are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is **double**.

— Otherwise, if any argument for generic parameters is of integer type, the type determined is **double**.

— Otherwise, if the corresponding real type of any argument for generic parameters has type **_Float32**, the type determined is **_Float32**.

— Otherwise, the type determined is **float**.

In the second bullet 7.25#3c, attach a footnote to the wording:

the type determined is the widest

where the footnote is:

*) The term widest here refers to a type whose set of values is a superset of (or equivalent to) the sets of values of the other types.

with:

In 7.25#3c, replace the first sentence and bullets:

> [3c] Except for the macros for functions that round result to a narrower type (7.12.13a), use of a type-generic macro invokes a function whose generic parameters have the corresponding real type determined by the corresponding real types of the arguments as follows:
>
> — First, if any argument for generic parameters has type **_Decimal128**, the type determined is **_Decimal128**.
>
> — Otherwise, if any argument for generic parameters has type **_Decimal64**, or if any argument for generic parameters is of integer type and another argument for generic parameters has type **_Decimal32**, the type determined is **_Decimal64**.
>
> — Otherwise, if any argument for generic parameters has type **_Decimal32**, the type determined is **_Decimal32**.
>
> — Otherwise, if the corresponding real type of any argument for generic parameters is **long double**, the type determined is **long double**.
>
> — Otherwise, if the corresponding real type of any argument for generic parameters is **double** or is of integer type, the type determined is **double**.
>
> — Otherwise, if any argument for generic parameters is of integer type, the type determined is **double**.
>
> — Otherwise, the type determined is **float**.

with:

> [3c] Except for the macros for functions that round result to a narrower type (7.12.13a), use of a type-generic macro invokes a function whose generic parameters have the corresponding real type determined by the types of the arguments for the generic parameters as follows:
>
> — Arguments of integer type are regarded as having type **_Decimal64** if any argument has decimal floating type, and as having type **double** otherwise.
>
> — If the function has exactly one generic parameter, the type determined is the corresponding real type of the argument for the generic parameter.
>
> — If the function has exactly two generic parameters, the type determined is the corresponding real type determined by the usual arithmetic conversions (6.3.1.8) applied to the arguments for the generic parameters.

— If the function has more than two generic parameters, the type determined is the corresponding real type determined by repeatedly applying the usual arithmetic conversions, first to the first two arguments for generic parameters, then to that result type and the next argument for a generic parameter, and so forth until the usual arithmetic conversions have been applied to the last argument for a generic parameter.

DDR #8

==========

Reference Document: Floating Point Extensions, Part 1

Subject: wrong type for **`fesetmode`** parameter

Summary

This is about the issue raised by Joseph Myers in email SC22WG14.14358:

> TS 18661-1 gives the declaration of **`fesetmode`** as:
>
> **`int fesetmode(const fenv_t *modep);`**
>
> The argument should be of type **`const femode_t *`**, not **`const fenv_t *`**.
>
> --

This was an editorial cut-and-past error. The Description says the argument **`modep`** shall point to an objet set by a call to **`fegetmode`**, which sets objects of type **`femode_t`**. It's unlikely the function would be implemented with the erroneous type.

Suggested Technical Corrigendum

In 15.3, in the new text for C 7.6.3.1a#1, change:

```
int fesetmode(const fenv_t *modep);
```

to:

```
int fesetmode(const femode_t *modep);
```

DDR #9

==========

Reference Document: Floating Point Extensions, Part 2

Subject: **`a`**-style formatting not IEC 60559 conformant

Summary

The **`a`**-style formatting specified in subclause 12.5 of TS 18661-2 is not an IEC 60559 conversion for cases where the formatting precision is less than the length of the coefficient of the input. The specification entails an intermediate rounding to the floating type of the input, which might overflow resulting in a character sequence representation of infinity. IEC 60559 conversions to character sequences do not overflow, unless the language over-restricts the exponent range for character sequence output, which C does not.

Another undesirable aspect of the current specification is that in certain cases it produces results with more precision than given by a width modifier.

Here are some examples, showing the result of the intermediate conversion, with different behaviors for the current spec ("old") and the spec in the suggested Technical Corrigendum below ("new"):

For **`_Decimal32`** input *x* with representation (1, 9512345, 90) and specifier ...

`%.3Ha`

old:	*x*	->	(1, 9510000, 90)	->	**`9.510000e96`**
new:	*x*	->	(1, 951, 94)	->	**`9.51e96`**

`%.2Ha`

old:	*x*	->	(1, 9500000, 90)	->	**`9.500000e96`**
new:	*x*	->	(1, 95, 95)	->	**`9.5e96`**

`%.1Ha`

old:	*x*	->	Inf	->	**`inf`**
new:	*x*	->	(1, 1, 97)	->	**`1e97`**

Here's another example:

For **`_Decimal32`** input x with representation (1, 9512345, 86) and specifier ...

`%.2Ha`

old:	*x*	->	(1, 950, 90)	->	**`9.50e92`**
new:	*x*	->	(1, 95, 91)	->	**`9.5e92`**

The examples use a to-nearest rounding.

As the examples illustrate, the problematic cases for the current "old" spec occur because of the exponent range limitation of the format used for the intermediate conversion.

The suggested Technical Corrigendum below specifies formatting that is IEC 60559 conformant and which honors a width modifier. It does not change the numerical value of the result, except in overflow cases.

Suggested Technical Corrigendum

In 12.5, in the addition to 7.21.6.1#8 and 7.29.2.1#8, under **a**,**A** conversion specifiers, change:

> If the precision is present (in the conversion specification) and is zero or at least as large as the precision *p* (5.2.4.2.2) of the decimal floating type, the conversion is as if the precision were missing. If the precision is present (and nonzero) and less than the precision *p* of the decimal floating type, the conversion first obtains an intermediate result by rounding the input in the type, according to the current rounding direction for decimal floating-point operations, to the number of digits specified by the precision, then converts the intermediate result as if the precision were missing. The length of the coefficient of the intermediate result is the smallest number, at least as large as the formatting precision, for which the quantum exponent is within the quantum exponent range of the type (see 5.2.4.2.2a). The intermediate rounding may overflow.

to:

> If the precision *P* is present (in the conversion specification) and is zero or at least as large as the precision *p* (5.2.4.2.2) of the decimal floating type, the conversion is as if the precision were missing. If the precision *P* is present (and nonzero) and less than the precision *p* of the decimal floating type, the conversion first obtains an intermediate result as follows, where *n* is the number of significant digits in the coefficient:
>
> > If *n* <= *P*, set the intermediate result to the input.
> >
> > If *n* > *P*, round the input value, according to the current rounding direction for decimal floating-point operations, to *P* decimal digits, with unbounded exponent range, representing the result with a *P*-digit integer coefficient when in the form (*s*, *c*, *q*).
>
> Convert the intermediate result in the manner described above for the case where the precision is missing.

In 12.5, in the addition to 7.21.6.1#8 and 7.29.2.1#8, in EXAMPLE 3, change the results:

```
9.54321e+93
9.5432e+93
9.543e+93
9.540e+93
9.500e+93
1.0000e+94
inf
```

to:

```
9.54321e+93
9.5432e+93
9.543e+93
9.54e+93
9.5e+93
1e+94
1e+97
```