Draft DRs for TS 18661

WG 14 N2125

2017-03-04

DDR #1

==========

Reference Document: TS 18661-1

Subject: Zero payloads and **`setpayload`** function

Summary

This is about an issue raised by Joseph Myers in SC22WG14.14450:

> The specification for **`setpayload`** (and likewise **`setpayloadsig`**) says "If **`pl`** is not a positive floating-point integer representing a valid payload, **`*res`** is set to positive zero."
>
> Does "positive" as applied to "floating-point integer" here mean "with sign bit 0" (the list of definitions in IEEE 754 doesn't include "positive")? In the preferred encodings for binary interchange formats, 0 is a valid payload for quiet NaNs. So should +0.0 as an argument to **`setpayload`** result in a quiet NaN with payload 0, while -0.0 results in **`*res`** being set to +0.0 because -0.0 isn't positive (and for **`setpayloadsig`**, both result in **`*res`** set to +0.0 because a payload for a signaling NaN has to be nonzero to avoid all mantissa bits being zero)?

A “positive floating-point integer” is a positive integer in the floating-point format, hence it is greater than zero. So, the current specification for **`setpayload`** and **`setpayloadsig`** is flawed in that it doesn’t allow setting the payload to zero.

A more basic problem is that TS 18661-1 assumes IEC 60559 interprets payloads as integers. This is true for decimal formats. IEC 60559 says:

> The payload corresponds to the significand of finite numbers, interpreted as an integer with a maximum value of 10^(3×J)−1, …

The significand c interpreted as an integer is assumed throughout to be non-negative, while the *s* field in (*s*, *q*, *c*) provides the sign. For decimal, interpreting the bits in the encodings allows the two encoding schemes to have the same payloads and the payloads to fit conceptually with their encoding schemes.

However, for binary formats, IEC 60559 says:

> For binary formats, the payload is encoded in the $p-2$ least significant bits of the trailing significand field.

Nowhere does it actually interpret the payload for binary formats as an integer.

However, the payload for binary formats has a natural interpretation as an unsigned integer, so it is reasonable for TS 1866-1 to interpret payloads (for binary and decimal formats) as such.

The suggested Technical Corrigendum below addresses these problems.

Suggested Technical Corrigendum

In 14.10, replace the first sentence:

> IEC 60559 defines the payload of a NaN to be a certain part of the NaN's significand interpreted as an integer.

with:

> IEC 60559 defines the payload of a NaN to be a certain part of the NaN's significand. The payload can be interpreted as an unsigned integer.

In 14.10, in the new C subclause F.10.13, replace:

> IEC 60559 defines the *payload* of a quiet or signaling NaN as an integer value encoded in the significand.

with:

> IEC 60559 defines the *payload* of a quiet or signaling NaN as information encoded in part of the NaN significand. The payload can be interpreted as an unsigned integer.

In 14.10, in the new C subclauses F.10.13.2#2 and F.10.13.3#2, change:

> If **`pl`** is not a positive floating-point integer representing a valid payload, **`*res`** is set to positive zero.

to:

> If **`pl`** is not a floating-point integer representing a valid payload, **`*res`** is set to positive zero.

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

Reference Document: TS 18661-3

Subject: Type-generic macros for functions that round result to narrower type

Summary

This is about an issue raised by Joseph Myers in SC22WG14.14561:

> TS 18661-1 and -2 define type-generic macros for the functions that round result to a narrower type. In part 1 these are, for example, fadd and dadd for addition; in part 2, for example, d32add and d64add.
>
> Part 3 does not seem to make any changes or additions to those macros, and consequences of that seem nonobvious. It defines new functions for the new types: fMaddfN, fMaddfNx, fMxaddfN, fMxaddfNx (where M < N, or M <= N in the fMaddfNx case), and likewise for decimal types. But the type-generic macros remain as defined in 7.25#6a after the changes from parts 1 and 2 are applied (part 3 does not contain the string "6a").
>
> That is, it's valid to pass the _FloatN and _FloatNx types to the fadd and dadd macros, and valid to pass the new _DecimalN and _DecimalNx types from part 3 to the d32add and d64add types.
>
> (a) 7.25#6a says "If the macro prefix is d32 or d64, use of an argument of standard floating type results in undefined behavior.". Other places get amended in part 3 to say "floating type of radix 2" in addition to "standard floating type". But it appears it fails to make it undefined to pass _FloatN or _FloatNx arguments to d32add, d64add etc. type-generic macros - although clearly it should be undefined.
>
> (b) Passing _Decimal128 to d32add would result in the d32addd128 function being called, as expected. But say you pass a _Decimal128x argument. A function d32addd128x exists but the specification would seem to result in d32addd64 being called, which seems unintuitive. Similar issues apply with _FloatN and _FloatNx types - calling fadd on them would always call the fadd function not faddl. (But in that case there *are* no functions defined that take _FloatN / _FloatNx arguments and return float or double. So the right thing to do is less obvious.)

The following addresses these issues by filling in the missing specification in part 3.

Suggested Technical Corrigendum

In clause 15, after the change to 7.25#6, add:

Change 7.25#6a from:

> [6a] The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any suffix from the function names. Thus, the macros with **f** or **d** prefix are:
>
> | | | |
> |---|---|---|
> | **fadd** | **fmul** | **ffma** |
> | **dadd** | **dmul** | **dfma** |
> | **fsub** | **fdiv** | **fsqrt** |
> | **dsub** | **ddiv** | **dsqrt** |
>
> and the macros with **d32** or **d64** prefix are:
>
> | | | |
> |---|---|---|
> | **d32add** | **d32mul** | **d32fma** |
> | **d64add** | **d64mul** | **d64fma** |
> | **d32sub** | **d32div** | **d32sqrt** |
> | **d64sub** | **d64div** | **d64sqrt** |
>
> All arguments are generic. If any argument is not real, use of the macro results in undefined behavior. If the macro prefix is **f** or **d**, use of an argument of decimal floating type results in undefined behavior. If the macro prefix is **d32** or **d64**, use of an argument of standard floating type results in undefined behavior. The function invoked is determined as follows:
>
> — If any argument has type **_Decimal128**, or if the macro prefix is **d64**, the function invoked has the name of the macro, with a **d128** suffix.
>
> — Otherwise, if the macro prefix is **d32**, the function invoked has the name of the macro, with a **d64** suffix.
>
> — Otherwise, if any argument has type **long double**, or if the macro prefix is **d**, the function invoked has the name of the macro, with an **l** suffix.
>
> — Otherwise, the function invoked has the name of the macro (with no suffix).

to:

> [6a] The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any suffix from the function

names. Thus, the macros with **f** or **d** prefix are:

fadd	**fmul**	**ffma**
dadd	**dmul**	**dfma**
fsub	**fdiv**	**fsqrt**
dsub	**ddiv**	**dsqrt**

and the macros with **f***M*, **f***M***x**, **d***M*, or **d***M***x** prefix are:

f*M***add**	**f***M***xmul**	**d***M***fma**
f*M***sub**	**f***M***xdiv**	**d***M***sqrt**
f*M***mul**	**f***M***xfma**	**d***M***xadd**
f*M***div**	**f***M***xsqrt**	**d***M***xsub**
f*M***fma**	**d***M***add**	**d***M***xmul**
f*M***sqrt**	**d***M***sub**	**d***M***xdiv**
f*M***xadd**	**d***M***mul**	**d***M***xfma**
f*M***xsub**	**d***M***div**	**d***M***xsqrt**

All arguments are generic. If any argument is not real, use of the macro results in undefined behavior. If the macro prefix is **f**, **d**, **f***M*, or **f***M***x**, use of an argument of decimal floating type results in undefined behavior. If the macro prefix is **d**M or **d***M***x**, use of an argument of standard or binary floating type results in undefined behavior. The function invoked is determined as follows:

— Arguments that have integer type are regarded as having type **_Decimal64** if any argument has decimal floating type, and as having type **double** otherwise.

— The unsuffixed name of the function is the name of the macro, and its suffix, if any, corresponds to the parameter type which may be any type with at least the range and precision of the argument types.

In clause 15, at the end of the text appended to the table in 7.25#7, further append:

f32xadd(d, f32x)	any **f32xaddf***N* or **f32xaddf***N***x** such that *N* > 32 and the suffix type, **_Float***N* or **_Float***N***x**, is at least as wide as **double** and **_Float32x**

DDR #3
==========

Reference Document: TS 18661-2

Subject: Effect of **%a** vs **%A** conversion specifiers

Summary

The specification in TS 18661-2 for **a**,**A** conversion specifiers for decimal describes the behavior in terms of **f** and **e** formatting. The intention was that the **A** conversion specifier would have the effects of **F** and **E** formatting. The following Technical Corrigendum corrects this oversight, using wording similar to that in C11 for the **g**,**G** conversion specifiers.

Suggested Technical Corrigendum

In 12.5, in the text added to 7.21.6.1#8 and 7.29.2.1#8, under **a**,**A** conversion specifiers, replace the bullets:

— if $-(n+5) \le q \le 0$, use style **f** formatting with formatting precision equal to $-q$,

— otherwise, use style **e** formatting with …

with:

— if $-(n+5) \le q \le 0$, use style **f** (or style **F** in the case of an **A** conversion specifier) with formatting precision equal to $-q$,

— otherwise, use style **e** (or style **E** in the case of an **A** conversion specifier) with …