

**Update to N2213 suggested TC for CFP CR 13
N2252**

Submitter: C FP group

Submission Date: 2018-05-10

Source: WG14

Reference Document: N2202, N2213, TS 18661-3

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

Summary

This document updates the suggested TC for TS 18661-3 CR 13 presented in N2213, which was an update to N2202.

After N2202 was posted, Joseph Myers sent the following message:

Joseph Myers

(SC22WG14.14921) Floating-point DR#13 and integer arguments to type-generic macros

To: SC22 WG14

I believe these comments all still apply to the version of the DR resolution in N2202: it still determines a type, but says nothing about what function is determined from that type (needed to cover `dadd(f, f)` which needs to call `daddl` to stay compatible with TS 18661-1, for example - the type determined is float, but what function is determined from it?).

--

Joseph S. Myers

joseph@codesourcery.com

On Thu, 23 Nov 2017, Joseph Myers wrote:

Looking at the latest proposed DR resolution

<http://wiki.edg.com/pub/CFP/WebHome/tgmath_for_narrowing_functions-20171117.pdf>:

This resolution changes text that partially determines a function called by type-generic macros such as `dadd`, to text that determines a type. Does it then result in a call to a function whose parameters have that type? I don't see anything saying so, but it's possible I've missed some text in the complicated sequence of (C11 amended by 18661-1 amended by 18661-2 amended by 18661-3 amended by DR#9 amended by DR#13 as modified by this proposed change to the resolution of DR#13).

In any case, there needs to be *something* about choosing a function whose arguments have a wider type than the one determined from the types of the arguments (subject to whatever's needed to keep things well-defined in the case of integer arguments, if desired), because of the `dadd(f, f)` case, which is clearly specified in TS 18661-1 to call the function `daddl`, and is included as an example there - as there isn't any `dadd` function with float or double arguments. A correction to TS 18661-3 should not have the effect of invalidating something that was valid with TS 18661-1.

--

Joseph S. Myers

joseph@codesourcery.com

The 23 Nov message Joseph Myers refers to had been overlooked and the valid issue it raises was not considered in the preparation of N2202. The suggested TC below revises the one in N2202 to address this issue. The changes to the suggested TC in N2202 are the additions of the last bullet and the last three examples.

With the approach suggested here, and in N2213, rounding of arguments might occur. For example, `f32xsqrt(f32x)` invokes `f32xsqrtf64x(f32x)` if `_Float64x` is supported, else `f32xsqrtf64`. Thus, if `_Float64x` is not supported and `_Float32z` is wider than `_Float64`, the argument `f32x` will be rounded to `_Float64`. We didn't see a way to avoid such roundings without unduly complicating the specification and/or breaking with the overall approach in C and the other parts of TS 18661. Note that the cases where argument rounding might occur do not represent the intended use of the macros: to round result to narrower type.

The macros that round results to narrower type differ from other `<tgmath.h>` macros in that the type of the expanded expression can be determined by the macro prefix, rather than by the argument types. We considered directly specifying that these macros produce their result with at most one rounding (after appropriately converting integer type arguments), and leaving the function to be called, or other manner of computation, to the implementation. We rejected this approach because it was inconsistent with the rest of the specification in `<tgmath.h>`.

After N2213 was posted, Joseph Myers sent the following message:

Joseph Myers
(SC22WG14.14928) Floating-point DR#13 and integer arguments to type-generic macros
To: SC22 WG14

Regarding the latest version in N2213:

This version does seem to make the chosen functions unambiguous, but it also leaves the chosen functions in some decimal floating-point cases different from what they are in TS 18661-2.

Consider the `d64add(d32, d32)` example in TS 18661-2, which is specified there as resulting in a call to `d64addd128`. Under the new proposal, because there is no `d64addd32` function, it would call `d64addd64x`. This difference isn't observable to the user, but at very least the example would need updating, to reflect that in the case where `__STDC_IEC_60559_TYPES__` is defined `d64addd64x` is called (whereas if `__STDC_IEC_60559_TYPES__` isn't defined, neither `_Decimal64x` nor `_Decimal96` is supported and so the new wording would result in the same function choice as in TS 18661-2).

If the arguments to `d64add` had a 128-bit integer type and `_Decimal64x` had less precision than `_Decimal128`, the difference in results would be user-visible. (Note, however, that I don't know if there are any existing implementations of the narrowing functions for decimal floating point, or of the associated type-generic macros, or of `_Decimal64x`, or of `_Decimal96`.)

The following suggested TC includes a minor change to N2213 so that the TS 18661-3 rules are consistent with the ones in TS 18661-2. The change was to rows 3 and 5 in the table and the `f32add` and `f64div` examples.

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 **fM**, **fMx**, **dM**, or **dMx** prefix are:

fMadd	fMxmul	dMfma
fMsub	fMxdiv	dMsqrt
fMmul	fMxfma	dMxadd
fMdiv	fMxsqrt	dMxsub
fMfma	dMadd	dMxmul
fMsqrt	dMsub	dMxdiv
fMxadd	dMmul	dMxfma
fMxsub	dMdiv	dMxsqrt

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 interchange or extended floating type results in undefined behavior. If the macro prefix is **fM**, or **fMx**, use of an argument of standard or decimal floating type results in undefined behavior. If the macro prefix is **dM** or **dMx**, 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 **double** if the macro prefix is **f** or **d**, as having type **_Float64** if the macro prefix is **fM** or **fMx**, and as having type **_Decimal64** if the macro prefix is **dM** or **dMx**.
- If the function has exactly one generic parameter, the type determined is the type of the argument.
- If the function has exactly two generic parameters, the type determined is the type determined by the usual arithmetic conversions (6.3.1.8) applied to the arguments.
- If the function has three generic parameters, the type determined is the type determined by applying the usual arithmetic conversions twice, first to the first two arguments, then to that result type and the third argument.

- If no function with the given prefix has the parameter type determined above, the parameter type is determined from the prefix as follows:

f	double
d	long double
fM	_FloatN for minimum $N > M$ if supported, else _FloatMx
fMx	_FloatNx for minimum $N > M$ if supported, else _FloatN for minimum $N > M$
dM	_DecimalN for minimum $N > M$ if supported, else _DecimalMx
dMx	_DecimalNx for minimum $N > M$ if supported, else _DecimalN for minimum $N > M$

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

fsub(d, ld)	fsubl
f32add(f64x, f64)	f32addf64x
d32xsqrt(n)	d32xsqrtd64
f32mul(f128, f32x)	f32mulf128 if _Float128 is at least as wide as _Float32x , or f32mulf32x if _Float32x is wider than _Float128
f32fma(f32x, n, f32x)	f32fmaf64 if _Float64 is at least as wide as _Float32x , or f32fmaf32x if _Float32x is wider than _Float64
ddiv(ld, f128)	undefined
f32fma(f64, d, f64)	undefined
fmul(dc, d)	undefined
f32add(f32, f32)	f32addf64(f32, f32)
f32xsqrt(f32)	f32xsqrtf64x(f32) if _Float64x is supported, else f32xsqrtf64
f64div(f32x, f32x)	f64divf128(f32x, f32x)