Document Number:	P2614R2
Date:	2022-11-08
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	LWG
Target:	C++23

Deprecate `numeric_limits::has_denorm`

ABSTRACT

Since C is intent on obsoleting the `*_HAS_SUBNORM` macros, we should consider the analogue change in C++: the deprecation of `numeric_limits::has_denorm`. In general, compile-time constants that describe floating-point *behavior* are problematic, since behavior might change at runtime. Let's also deprecate `numeric_limits::has_denorm_loss` while we're at it.

CONTENTS

1 CHANGELOG

1.1 CHANGES FROM REVISION 0

Previous revision: P2614R0

- Added Wording.

2 STRAW POLLS

2.1 LEWG TELECON 2022-09-13

Poll: Deprecate `numeric_limits::has_denorm`

SF	F	N	A	SA
5	6	0	0	0

Poll: Deprecate `numeric_limits::has_denorm_loss`

SF	F	N	A	SA
5	6	0	0	0

3 INTRODUCTION

`numeric_limits` has a member called `has_denorm` of type `float_denorm_style`:

```
enum float_denorm_style {
  denorm_indeterminate = -1,
  denorm_absent        = 0,
  denorm_present       = 1
};
```

As Tydeman [N2993] states:

> There are several ways subnormals are "supported" in the field:
>
> - Partial support - hardware has encodings, but operations "fail".
>   - Operands are flushed to zero; results are kept.
>   - Operands are kept; results are flushed to zero.
>   - Some operations flush, others do not flush.
>   - Results suffer double rounding.
>   - Support can be changed at runtime (not by means in Stardard C).
> - Not at all. There are no hardware encodings of subnormals.

- Full support as per IEEE-754.

Since hardware can change in future iterations, an implementation that does not want to risk an ABI break via `numeric_limits` will never set `has_denorm` to `denorm_absent` or `denorm_present`. The only ABI-safe sensible value is `denorm_indeterminate`. I.e. implementations cannot give a compile-time guarantee about *run-time behavior*.

The `has_denorm` value is not helping C++ users. Worst case, it is misleading users, resulting in incorrect assumptions and possibly breaking algorithms at some point.

3.1 `has_denorm_loss`

The subsequent member in `numeric_limits`, `has_denorm_loss` is also calling for deprecation. Who can tell me the meaning of: "true if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.[1]" cppreference.com explains [1]:

> No implementation of denormalization loss mechanism exists (accuracy loss is detected after rounding, as inexact result), and this option was removed in the 2008 revision of IEEE Std 754.
>
> libstdc++, libc++, libCstd, and stlport4 define this constant as false for all floating-point types. Microsoft Visual Studio defines it as true for all floating-point types.

I don't own a IEEE 754 revision older than the 2008 revision, so it's hard to check. But at least the 2008 revision has no occurence of the word "loss" and no relevant occurence of "accuracy". The footnote's reference to the IEEE 754 standard is impossible to follow.

4 PROPOSED SOLUTION

The `has_denorm` and `has_denorm_loss` values should not be used.

A shallow code search[2] suggests that no code actually relies on `has_denorm`. However, a removal of the value would be a major compatiblity break. We can deprecate it, but without an actual intent of removal (since it would break too much). As an alternative to deprecation, we could change paragraph 46 "Meaningful for all floating-point types." to state that it's not even meaningful for floating-point types. Thus, user-defined types could still define a meaning for `has_denorm`.

The preference of SG6 after discussing [N2993] was deprecation of `has_denorm`.

`has_denorm_loss` should simply be deprecated (without actual intent of removal, though). The reference to IEEE 754 should be removed in any case.

1 See ISO/IEC/IEEE 60559.

2 `https://codesearch.isocpp.org/cgi-bin/cgi_ppsearch?q=has_denorm&search=Search`

5 WORDING

This wording is relative to N4917.

Modify 17.3.3 [limits.syn] as follows:

[limits.syn]

```
// 17.3.4, floating-point type properties
enum float_round_style;
enum float_denorm_style;
```

Remove 17.3.4.2 [denorm.style].

Modify 17.3.5.1 [numeric.limits.general] as follows:

[numeric.limits.general]

```
static constexpr bool has_quiet_NaN = false;
static constexpr bool has_signaling_NaN = false;
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
static constexpr T infinity() noexcept { return T(); }
static constexpr T quiet_NaN() noexcept { return T(); }
```

Modify 17.3.5.2 [numeric.limits.members] as follows:

[numeric.limits.members]

`static constexpr bool has_signaling_NaN;`

42 `true` if the type has a representation for a signaling “Not a Number”.

43 Meaningful for all floating-point types.

44 Shall be `true` for all specializations in which `is_iec559` $\neq$ `false`.

`static constexpr float_denorm_style has_denorm;`

45 `denorm_present` if the type allows subnormal values (variable number of exponent bits), `denorm_absent` if the type does not allow subnormal values, and `denorm_indeterminate` if it is indeterminate at compile time whether the type allows subnormal values.

46 Meaningful for all floating-point types.

`static constexpr bool has_denorm_loss;`

47 `true` if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.

`static constexpr T infinity() noexcept;`

48 Representation of positive infinity, if available.

49 Meaningful for all specializations for which `has_infinity` $\neq$ `false`. Required in specializations for which `is_iec559` $\neq$ `false`.

`static constexpr T quiet_NaN() noexcept;`

Representation of a quiet "Not a Number", if available.

Meaningful for all specializations for which `has_quiet_NaN` $\neq$ `false`. Required in specializations for which `is_iec559` $\neq$ `false`.

`static constexpr T signaling_NaN() noexcept;`

Representation of a signaling "Not a Number", if available.

Meaningful for all specializations for which `has_signaling_NaN` $\neq$ `false`. Required in specializations for which `is_iec559` $\neq$ `false`.

`static constexpr T denorm_min() noexcept;`

Minimum positive subnormal value, if available. Otherwise minimum positive normalized value.

Meaningful for all floating-point types.

~~In specializations for which `has_denorm` == `false`, returns the minimum positive normalized value.~~

Modify 17.3.5.3 [numeric.special] paragraph 2 as follows:

[numeric.special] p2

```
static constexpr bool has_infinity = true;
static constexpr bool has_quiet_NaN = true;
static constexpr bool has_signaling_NaN = true;
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
```

Modify 17.3.5.3 [numeric.special] paragraph 3 as follows:

[numeric.special] p3

```
static constexpr bool has_infinity = false;
static constexpr bool has_quiet_NaN = false;
static constexpr bool has_signaling_NaN = false;
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
```

Add a new subclause in Annex D after D.10 [depr.res.on.required]:

[depr.numeric.limits.has.denorm]

(5.0.1) ## D.11 `has_denorm` members in `numeric_limits` [depr.numeric.limits.has.denorm]

1 The following type is defined in addition to those specified in 17.3.4 [limits.syn]:

```
namespace std {
  enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
  };
}
```

2 The following members are defined in addition to those specified in 17.3.5.1 [numeric.limits.general]:

```
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
```

3 The values of `has_denorm` and `has_denorm_loss` of specializations of `numeric_limits` are unspecified.

4 The following members are defined in addition to those specified in 17.3.5.3 [numeric.special] paragraph 3:

```
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
```

A Acknowledgments

Thanks to WG14 and specifically Fred Tydeman for their work on `*_HAS_SUBNORM` and presenting in SG6. Thanks to Dietmar Kühl, Fred Tydeman, Jens Maurer, John McFarlane, and Mark Hoemmen for the discussion in SG6 that motivated this paper. Thanks to Mark Hoemmen for pointing out that we should deprecate `has_denorm_loss`.

B Bibliography

[N2993] Fred Tydeman. *N2993: Make *_HAS_SUBNORM be obsolescent*. ISO/IEC C Standards Committee Paper. 2022. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2993.htm.

[1] User:Cubbi. *std::numeric_limits<T>::has_denorm_loss - cppreference.com*. 2021. URL: https://en.cppreference.com/w/cpp/types/numeric_limits/has_denorm_loss (visited on 07/05/2022).