TS 18661-4,5
REVISIONS

N3165
WG 14 – virtual meeting
October 16 – 20, 2023

C FP group
Latest drafts

- Nnnnn  draft ISO/IEC TS 18661-4, 2\textsuperscript{nd} ed
- Nnnnn  draft ISO/IEC TS 18661-5, 2\textsuperscript{nd} ed
Background (1)

- C99 was first C to support IEEE 754/IEC 60559.
- 754-2008 was a major revision.
- C17 still based on 754-1985.
- C23 supports 754-2008 and minor revision in 2019, mostly.
Background (2)

- Developed ISO/IEC TS 18661, in five parts:
  - 1: Binary floating-point arithmetic (2014)
  - 2: Decimal floating-point arithmetic (2015)
  - 3: Interchange and extended types (2015)
  - 4: Supplementary functions (2015)
  - 5: Supplementary attributes (2016)
Background (3)

Incorporation into C23

* Not reduction functions

** Not augmented arithmetic
TS-4 & TS-5 revisions - purpose

Specify C extensions for IEC 60559 features not in C23.

TS-4 Supplementary functions
• Reduction functions from TS-4 v1
• Augmented arithmetic from IEC 60559: 2020

TS-5 Supplementary attributes
• All of TS-5 v1
Changes from 1st versions

- Written in the style of an annex for C extensions, not as changes to C.
- Based on C23.
- Based on IEC 60559-2020.
- Includes new augmented arithmetic feature from IEC 60559-2020.
- Offers conformance to separate features in the TSes.
- Changes most pragma prefixes from `FENV_` to `FP_` and headers from `<fenv.h>` to `<math.h>`.
- Includes new evaluation method macros that reflect the effective evaluation method.
- Includes examples for use of scaled products and augmented arithmetic.
- Changes scaled product output type from `long long int` to `long int` to work with `scalbln`. 
TS-4 V2
SUPPLEMENTARY FUNCTIONS
Two features, with separate feature macros ...

1. Reduction functions
   - \texttt{__STDC_IEC_60559_FUNCS_REDUCTION__}
   - Sum reductions
   - Scaled products

2. Augmented arithmetic
   - \texttt{__STDC_IEC_60559_FUNCS_AUGMENTED_ARITHMETIC__}

Both are \texttt{<math.h>} extensions, interfaces guarded by \texttt{__STDC_WANT_IEC_60559_FUNCS_EXT__}
Sum reductions

IEC 60559:2011 specifies and recommends sum reduction operations on vectors \( p \) and \( q \) of length \( n \):

- sum\((p, n)\) \(=\sum_{i=1}^{n} p_i\)
- dot\((p, q, n)\) \(=\sum_{i=1}^{n} p_i \times q_i\)
- sumSquare\((p, n)\) \(=\sum_{i=1}^{n} p_i^2\)
- sumAbs\((p, n)\) \(=\sum_{i=1}^{n} |p_i|\)
Scaled products

IEC 60559 specifies and recommends scaled product reduction operations: compute without over/underflow

\[ pr = \text{scaled product} \quad \text{and} \quad sf = \text{scale factor} \]

such that

\[ \text{result product} = pr \times \text{radix}^{sf} \]

\[
\text{scaledProd}(p, n) = \prod_{i=1}^{n} p_i
\]

\[
\text{scaledProdSum}(p, q, n) = \prod_{i=1}^{n} (p_i + q_i)
\]

\[
\text{scaledProdDiff}(p, q, n) = \prod_{i=1}^{n} (p_i - q_i)
\]
<table>
<thead>
<tr>
<th>IEC 60559</th>
<th>TS 16881-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>reduc_sum</td>
</tr>
<tr>
<td>dot</td>
<td>reduc_sumprod</td>
</tr>
<tr>
<td>sumSquare</td>
<td>reduc_sumsq</td>
</tr>
<tr>
<td>sumAbs</td>
<td>reduc_sumabs</td>
</tr>
<tr>
<td>scaledProd</td>
<td>scaled_prod</td>
</tr>
<tr>
<td>scaledProdSum</td>
<td>scaled_prodsum</td>
</tr>
<tr>
<td>scaledProdDiff</td>
<td>scaled_proddiff</td>
</tr>
</tbody>
</table>
Reduction function interfaces

```c
#define __STDC_WANT_IEC_60559_FUNCS_EXT__
#include <math.h>

#ifdef __STDC_IEC_60559_BFP__
#ifdef __STDC_IEC_60559_FUNCS_REDUCTION__
double reduc_sum(size_t n, const double p[static n]);
...
double scaled_prod (size_t n, const double p[static n],
    long int * restrict sfptr );
...

Arrays indexed 0 to n - 1
```
IEC 60559 reductions

- Result values not fully specified like other IEC 60559 operations.
- Implementation can (re)order operations and use extra range and precision, for speed and accuracy.
- Must avoid over/underflow, except if final result of a sum reduction deserves over/underflow.
- Scaled products allow computing quotients of huge products whose numerator and denominator products would overflow.
Reduction special cases (1)

Follows general principles for special cases, e.g.

\texttt{reduc\_sum(n, p)}

- Returns a NaN if any member of array \texttt{p} is a NaN.
- Returns a NaN and raises the “invalid” floating-point exception if any two members of array \texttt{p} are infinities with different signs.
- Otherwise, returns $\pm\infty$ if the members of \texttt{p} include one or more infinities $\pm\infty$ (with the same sign).
Reduction special cases (2)

\texttt{scaled\_prod(n, p, sfptr)}

- Returns a NaN if any member of array \( p \) is a NaN.
- Returns a NaN and raises the “invalid” floating-point exception if any two members of array \( p \) are a zero and an infinity.
- Otherwise, returns an infinity if any member of array \( p \) is an infinity.
- Otherwise, returns a zero if any member of array \( p \) is a zero.
- Otherwise, returns a NaN and raises the “invalid” floating-point exception if the scale factor is outside the range of the \texttt{long int} type.
TS-4 V2
SUPPLEMENTARY ATTRIBUTES
TS-5 revision (1)

Four features, with separate feature macros ... 

Evaluation formats
• __STDC_IEC_60559_ATTRIB_EVALUATION_FORMAT__

Optimization controls
• __STDC_IEC_60559_ATTRIB_OPTIMIZATION__

Reproducibility
• __STDC_IEC_60559_ATTRIB_REPRODUCIBLE__

Alternate exception handling
• __STDC_IEC_60559_ATTRIB_ALTERNATE_EXCEPTION_HANDLING__

Interfaces guarded by
• __STDC_WANT_IEC_60559_ATTRIBS_EXT__
TS-5 revision (2)

- IEC 60559 recommends that language standards provide block-scope attributes to control expression evaluation, value-changing optimizations, reproducibility, and alternate exception handling.
- TS 18661-5 provides these attributes as standard pragmas, like existing FP pragmas.
- The attributes are intended to address four problem areas in FP programming …
Problem area 1

Porting floating-point code between platforms and tool sets, including debugging ported code

• Program development tools typically provide controls to manage optimizations and evaluation methods.
• These controls are implementation specific, both in syntax and semantics, and are often vaguely defined.
• It’s difficult to impossible to map controls on one system to equivalent ones on another.
• Standard pragmas for evaluation methods and optimizations are intended to address this problem.
Problem area 2

*Balancing performance against precision and reliability*

- Current implementation-specific controls are usually compiler options that apply to the whole translation unit.
- However, many programs need aggressive optimizations only for relatively small performance-critical blocks.
- Applying value-changing optimizations where they aren’t needed unnecessarily risks floating-point anomalies.
- Similarly, extra precision might be needed only in relatively small precision-critical blocks.
- Using extra precision throughout the program might unnecessarily degrade performance.
- The block-scope semantics of the pragmas address this problem.
The following pragmas provide the preferred `width` attributes recommended for language standards by IEC 60559:

```c
#pragma STDC_FPFLT_EVAL_METHOD width
```

- `width` indicates a supported evaluation method for which macro `FLT_EVAL_METHOD` has the value `width`.
- Requires support for `width` equal -1 (indeterminable), 0 (evaluate to wider of `float` and type), and `DEFAULT`.
- Allows support for other values of `width`. 
Evaluation methods (2)

#pragma STDC   FP_DEC_EVAL_METHOD width

- Like FPFLT_EVAL_METHOD, but for decimal.
- width indicates a supported evaluation method for which macro
  DEC_EVAL_METHOD has the value width.
- Requires support for width equal -1 (indeterminant), 1 (evaluate
to wider of _Decimal64 and type), and DEFAULT.
- Allows support for other values of width.

TS-5 also specifies a user definable macro

___STDC_TGMATH_OPERATOR_EVALUATION___
to have tgmath macros follow the evaluation method like
operators do -- to allow wide evaluation that is consistent for all
FP operations.
Evaluation methods (3)

TS-5 clarifies that the macros `FLT_EVAL_METHOD` and `DEC_EVAL_METHOD`
- Characterize the default evaluation methods
- Are not affected by evaluation method pragmas
- Can be used in `#if/elif` directives

 Adds similar macros `FLT_EVAL_METHOD_EFFECTIVE` and `DEC_EVAL_METHOD_EFFECTIVE` that
- Characterize the effective evaluation methods
- Are affected by the evaluation method pragmas
- Cannot be used in `#if/elif` directives
Optimizations (1)

The following pragmas provide value-changing-optimization attributes recommended for language standards by IEC 60559:

```c
#pragma STDC FP_ALLOW_ASSOCIATIVE_LAW on-off-switch
• x + (y + z) = (x + y) + z
• x * (y * z) = (x * y) * z
```

```c
#pragma STDC FP_ALLOW_DISTRIBUTIVE_LAW on-off-switch
• x *(y + z) = (x * y) + (x * z)
• x *(y − z) = (x * y) − (x * z)
• (x + y) / z = (x / z) + (y / z)
• (x − y) / z = (x / z) − (y / z)
```
Optimizations (2)

#pragma STDC FP_ALLOW_MULTIPLY_BY_RECIPROCAL on-off-switch
• \( \frac{x}{y} = x \cdot \left( \frac{1}{y} \right) \)

#pragma STDC FP_ALLOW_CONTRACT_FMA on-off-switch
• Contract (compute with just one rounding) floating-point multiply and add or subtract (with the result of the multiply).
  • \( x \cdot y + z \)  \( x \cdot y - z \)
  • \( x + y \cdot z \)  \( x - y \cdot z \)

#pragma STDC FP_ALLOW_CONTRACT_OPERATION_CONVERSION on-off-switch
• Contract a floating-point operation and a conversion (of the result of the operation), e.g., \( \text{flt}_\text{var} = \text{dbl}_\text{var} \cdot \text{dbl}_\text{var} \).
Optimizations (3)

#pragma STDC FP_ALLOW_CONTRACT on-off-switch
• Includes effects of two “contract” pragmas above.
• Equivalent to C’s FP_CONTRACT pragma.

#pragma STDC FP_ALLOW_ZERO_SUBNORMAL on-off-switch
• Replace subnormal operands and results by zero.

#pragma STDC FP_ALLOW_VALUE_CHANGING_OPTIMIZATION on-off-switch
• Equivalent to all the optimization pragmas above.

➢ Optimization pragmas allow but do not require the optimizations.
Problem area 3

Obtaining reproducible results (on same or different platforms)

• Some users want results that are the same on different platforms and that remain the same after tool set updates.
• Variations in floating-point results are usually harmless, but not always. The cost to determine whether a difference is the result of insignificant roundoff errors or the result of a serious instability or bug can be great.
• Potential causes of differences in floating-point results are many, and difficult for most programmers to avoid.
• A pragma and guidance for reproducible results is intended to help with this problem.
Reproducibility (1)

The following pragma provides the reproducible-results attribute recommended for language standards by IEC 60559:

```c
#pragma STDC FP_REPRODUCIBLE on-off-switch
```

Implies effects of

- `#pragma STDC FENV_ACCESS ON`
- `#pragma STDC FP_ALLOW_VALUE_CHANGING_OPTIMIZATION OFF`

and if `__STDC_IEC_60559_BFP__` is defined
- `#pragma STDC FP_FLT_EVAL_METHOD 0`

and if `__STDC_IEC_60559_DFP__` is defined
- `#pragma STDC FP_DEC_EVAL_METHOD 1`
Reproducibility (2)

- Recommends a diagnostic message if the source code uses a language or library feature whose results may not be reproducible.
- Includes guidelines for code intended to be reproducible, e.g.
  - The code does not contain any use that may result in undefined behavior. The code does not depend on any behavior that is unspecified, implementation-defined, or locale-specific.
  - The code does not use the `long double` type.
  - The code does not depend on the payloads (F.10.13) or sign bits of quiet NaNs.
  - The code does not use signaling NaNs.
  etc.
A low-quality or initial implementation of the features for evaluation methods, optimizations and reproducibility could have a conformance mode where only \texttt{FLT_EVAL_METHOD} equal 0 is supported, optimizations are disabled, and pragmas are ignored.
Problem area 4

Where *default exception handling is not what a user wants*

- Floating-point exceptions occur when there is no generally best result

- **IEC 60559 default exception handling:**
  - recognizes five kinds of exceptions (invalid operation, division by zero, overflow, underflow, and inexact)
  - sets an exception flag
  - provides well-defined results
  - provides results that are intended to be at least as generally useful as any others
  - does not stop or change the flow of execution

- A *pragma for alternate exception handling* provides other ways to handle exceptions
Exceptions

They're called exceptions because no matter what default you choose, somebody will take exception to it.

~ W. Kahan
Alternate exception handling (1)

#pragma STDC FENV_EXCEPT except-list action

except-list  a comma-separated list of

   exception macro names:

       FE_DIVBYZERO, FE_INVALID, ...

and FE_ALL_EXCEPT

and optional sub-exception designations:

       FE_INVALID_ADD       +∞ + (−∞)
       FE_INVALID_MUL       ∞ * 0
       FE_INVALID_SNAN      signaling NaN operand
       FE_DIVBYZERO_LOG     log(0)

etc.
Alternate exception handling (2)

action one of

• DEFAULT
  IEC 60559 default exception handling.

• NOEXCEPT
  like default but no flags set.

• OPTEXCEPT
  like default but flags may be set.

• ABRUPT
  only for “underflow”, IEC 60559-defined abrupt underflow shall occur, unlike ALLOW_ZERO_SUBNORMAL where zeroing may occur.
Alternate exception handling (3)

The following actions change flow of control

\textit{action} one of (cont.)

- \textbf{BREAK}

  terminate compound statement associated with pragma, ASAP*.

*ASAP – for performance, the objects, flags, dynamic modes, and library states that would be changed at any point if the compound statement ran to completion are indeterminate or unspecified.
Alternate exception handling (4)

action one of (cont.)

These work together

• **TRY**
  
  A designated exception may be handled (ASAP) by a compound statement associated with a **CATCH** action.

• **CATCH**
  
  Code to handle designated exceptions.
Alternate exception handling (5)

*action* one of (cont.)

These work together

- **DELAYED_TRY**
  
  After associated compound statement completes, a designated exception may be handled by a compound statement associated with a `DELAYED_CATCH` action.

- **DELAYED_CATCH**
  
  Code to handle designated exceptions.
Alternate exception handling (6)

double d[n]; float f[n];
...
#pragma STDC FENV_EXCEPT TRY FE_DIVBYZERO, FE_OVERFLOW
{
    for (i=0; i<n; i++) {
        f[i] = 1.0 / d[i];
    }
}
#pragma STDC FENV_EXCEPT CATCH FE_DIVBYZERO
{
    printf("divide-by-zero\n");
}
#pragma STDC FENV_EXCEPT CATCH FE_OVERFLOW
{
    printf("overflow\n");
}
Questions?
Thank you!