PROPOSAL FOR I/O IN P1788
VERSION 2

JOHN PRYCE

1. Introduction

I put forward this proposal on what place I/O should have in the P1788 standard. The aim is to follow the spirit of the scheme for conversion between floating-point numbers and character sequences in IEEE 754-2008, called 754 henceforth.

Saying nothing about I/O runs the risk of incompatible implementations by different languages, and by different compilers for the same language—a bad and unnecessary outcome. Saying too much risks tying the specification too closely to a particular language.

So a middle road is desirable. Fortunately, 754§5.4.2 and §5.12 specify the mathematical properties of conversion while leaving details, mostly of formatting, language-defined or implementation-defined. For instance it is immaterial whether a given decimal value appears externally as `1.2345e6` or `.12345D+07`, or is written right-to-left, etc.

The extra requirement, for intervals, is that containment hold on both input and output so that, when a program computes an enclosure of some quantity given an enclosure of the data, this remains true all the way from text data to text results.

The proposed solution puts a layer of abstraction between internal intervals and a character stream, by defining a *text interval*. At level 2 (ti-datum) this is a member of a particular kind of interval format (ti-format). At level 3 (ti-object) it is an implementation-defined structure in the program which for a nonempty interval contains the bounds, exactly, in text form. The part of interval I/O that concerns P1788 then reduces to interval hull operations.

For instance a binary64 interval $\boldsymbol{x}$ holding a tight enclosure of $[1/3, 2/3]$ may be converted to a ti-datum $\boldsymbol{y} = [0.333333, 0.666667]$, represented by a ti-object $\boldsymbol{Y}$ whose bounds fields hold the strings `3.33333E-1` and `6.66667E-1`. The conversion was done by forming $\boldsymbol{y} = \mathrm{hull}_{\mathbb{T}}(\boldsymbol{x})$ where $\mathbb{T}$ is a certain ti-format: in a C-related language it may be the one defined by the conversion specifier `%.5e`. A 754-conforming arithmetic provides the outward-rounding operations needed to achieve this, as specified in 754§5.12.

Finally, depending on the language, $\boldsymbol{y}$ might appear on an output stream as `[3.33333E-1, 6.66667E-1]` or as `<3,33333E-1 6,66667E-1>`, etc., but P1788 is not concerned with this stage.

An issue discussed in the group, and in the Vienna proposal, is related because it concerns converting numbers between text and internal form. Namely, the loss of containment when a literal like `0.1` in program code is converted to a floating-point number and then used in constructing an interval. However, this seems an entirely language matter that this proposal cannot contribute to.

[*Note. Some interval output forms are more akin to mid-rad representation than to inf-sup. An example is the* `12.3_` *form, meaning* $[12.25, 12.35]$. *It would be possible, but not very convenient, to handle this form by the mechanism proposed here.*]

I acknowledge suggestions from George Corliss, which are the reason why this document has version number 2.

2. Motion

In P1788 the definition of I/O and the requirements for its support shall be as specified in the following rationale.

Date: May 15, 2010.

3. Rationale

3.1. **Definitions.**

3.1.1. *Text* means character sequences (the 754 term) generally, in some language-defined character set, and *string* means a particular character sequence.

A *text number format* (tn-format) is a set $\mathbb{T}$ of strings each of which represents a number in the extended reals $\mathbb{R}^*$ in a language-defined way. Different strings may represent the same number. A member of a tn-format is called a *text number.* The set $\mathrm{num}(\mathbb{T}) \subseteq \mathbb{R}^*$ of numbers thus represented shall contain $\pm\infty$. Hence $\mathrm{num}(\mathbb{T})$ is a number format as defined in P1788§5.1.

Following P1788 convention, a member of $\mathbb{T}$ may be identified with the number it represents, and $\mathbb{T}$ may be identified with $\mathrm{num}(\mathbb{T})$, when context makes clear.

A *decimal tn-format* is one where numbers are expressed in radix 10, or an equivalent. A *binary tn-format* is one where numbers are expressed in radix 2, or an equivalent such as octal or hexadecimal.

[*Example.* An example of a decimal tn-format is the set $\mathbb{T}$ comprising all strings that can be output by `printf()` in C using the `%.5e` conversion specifier. An example of a binary tn-format is the "hexadecimal-significand" form of 754§5.12.3, output in C by the `%a` specifier.]

The set of supported tn-formats, and the elements of each such tn-format, is language-defined, subject to the precision requirement in 3.4.2.

A *stream* is a medium for input or output of character sequences, usually external to the program but possibly an internal type: for instance in C one can read from and write to an array of `char`.

3.1.2. A *text interval* is a mathematical interval, namely either the empty set or an interval whose endpoints are text numbers of some tn-format $\mathbb{T}$. That is, it is a $\mathrm{num}(\mathbb{T})$-interval in the sense of P1788 §5.2; this is abbreviated to $\mathbb{T}$-interval. The set of all $\mathbb{T}$-intervals is the *text interval format* (ti-format) of $\mathbb{T}$, denoted by $\overline{\mathbb{I}\mathbb{T}}$.

The interval-hull operation of a ti-format $\overline{\mathbb{I}\mathbb{T}}$ is written $\mathrm{hull}_{\mathbb{T}}$ (an abbreviation of $\mathrm{hull}_{\mathrm{num}(\mathbb{T})}$).

3.1.3. A text-interval is particular kind of level 2 (bare, i.e. undecorated) interval datum. When this is to be emphasized it is called a *ti-datum.* Following P1788§6.1, an implementation may choose any means to represent a ti-datum $\boldsymbol{x}$ by a level 3 *ti-object* `X`, provided that it shall be possible to retrieve the bounds of any nonempty $\boldsymbol{x}$ exactly.

[*Note. A ti-object must have a means to represent the empty interval, as well as infinite bounds of nonempty intervals. Representations should be such that converting text intervals to (language-defined) text form on an output stream is straightforward.*]

3.2. **Input.**

3.2.1. The process of reading an interval in text form from an input stream, and storing it in a ti-object `X` that represents a ti-datum $\boldsymbol{x}$, is language-defined. The part of input conversion that concerns P1788 is done by a function

formatOf-**convertFromTextInterval**$(\boldsymbol{x})$

which returns $\mathrm{hull}_{\mathbb{F}}(\boldsymbol{x})$ where $\mathbb{F}$ is the destination format specified by *formatOf.*

3.2.2. The text intervals accepted as input belong to a language-defined *universal ti-format* for which the bounds $\underline{x}, \overline{x}$ of a nonempty $\boldsymbol{x}$ are (the values of) arbitrary floating constants (including infinities) defined by the language. These constants may be in any radix supported by the language and their values shall exactly equal those of the corresponding interval bounds on the input stream. Thus there shall be only one rounding of each bound, namely that on converting to the destination format. It is language-defined whether $\underline{x}$ can be of a different radix from $\overline{x}$.

The universal ti-format shall include a representation of the empty interval.

[*Example.* A language might specify intervals on the input stream in the form `$(3.456:3.789)$`. The language is responsible for parsing this and creating a text interval $\boldsymbol{x}$ with the exact bounds $\underline{x} = 3.456$ and $\overline{x} = 3.789$. This is converted to, say, a binary64 interval by calling

binary64convertFromTextInterval $(\boldsymbol{x})$.

]

3.3. **Output.**

3.3.1. The part of output conversion that concerns P1788 consists in replacing a interval $\boldsymbol{x}$ in the program by its hull $\boldsymbol{y} = \mathrm{hull}_{\mathbb{T}}(\boldsymbol{x})$, for some tn-format $\mathbb{T}$, by a function

convertToTextInterval ($\boldsymbol{x}$, *conversionSpecifier*)

where *conversionSpecifier* defines $\mathbb{T}$ in a language-defined way. The process of putting $\boldsymbol{y}$ on an output stream is language-defined.

[*Example.* Suppose the language is C. Let `x` be a binary64 interval-object holding a tight enclosure of $[1/3, 2/3]$. Suppose conversion specifiers are strings, whose allowed values are a subset of the specifiers supported by `fprintf()`. Suppose the chosen one is `%.5e`, and the C name of **convertToTextInterval** is `intvl2text`. Then `x` is converted by

```
Y = intvl2text(x,"%.5e");
```

which produces a ti-object `Y` whose bounds are the text numbers `3.33333E-1` and `6.66667E-1`. Suppose `Y` is a structure with fields `inf` and `sup` that hold these bounds. Then a library function that writes ti-objects to an output stream `os` might use the statement

```
fprintf(os,"[%s, %s]", Y.inf,Y.sup);
```

which outputs `[3.33333E-1, 6.66667E-1]`.]

3.4. **Constraints.**

3.4.1. For each supported (binary or decimal) interval format $\overline{\mathbb{IF}}$ belonging to a format $\mathbb{F}$, an implementation should provide sufficient decimal tn-formats, defined by *conversionSpecifier*s as in 3.3.1, to meet common user needs: for instance `%e.`d and `%f.`d type specifiers with a range of d values appropriate to the precision of $\mathbb{F}$.

3.4.2. In a 754-conforming implementation, the tn-formats shall meet the requirement of 754§5.12, that binary-to-decimal conversion shall be available to sufficient precision for the original value to be recovered exactly. Namely, for each supported interval format $\overline{\mathbb{IF}}$ belonging to a format $\mathbb{F}$ there shall be a decimal tn-format $\mathbb{T}$ with the following property. For any nonempty $\boldsymbol{x} \in \overline{\mathbb{IF}}$, the bounds of $\boldsymbol{y} = \mathrm{hull}_{\mathbb{T}}(\boldsymbol{x})$, when converted back to format $\mathbb{F}$ in round-to-nearest mode, shall give the bounds of $\boldsymbol{x}$ exactly. Because of outward rounding, this does not imply that $\boldsymbol{x}' = \mathrm{hull}_{\mathbb{F}}(\boldsymbol{y})$ equals $\boldsymbol{x}$. However: (i) $\boldsymbol{x}'$ differs from $\boldsymbol{x}$ by at most one ulp at either end; (ii) knowing that $\boldsymbol{y}$ was created as just described, one can recover $\boldsymbol{x}$ exactly from $\boldsymbol{y}$.

3.4.3. A 754-conforming implementation shall provide, for each supported interval format $\overline{\mathbb{IF}}$, a binary ti-format $\overline{\mathbb{IT}}$ that represents any member $\boldsymbol{x}$ of $\overline{\mathbb{IF}}$ exactly, and conversions between these. Conversion to $\overline{\mathbb{IT}}$ is by converting the *interchange format version* of $\boldsymbol{x}$ to give two numbers in the hexdecimal-significand form of 754§5.12.3. Thus the result is independent of the implementation's level 3 representation of intervals (inf-sup versus neginf-sup, etc.).