Farance Inc.

subject: **C Binding of ISO 10967 (LIA-1)**
document: **WG14/N461 X3J11/95-062**
file: **`language-independent/lia-1.*`**

date: **1995-08-25**

from: **Frank Farance**
+1 212 486 4700
frank@farance.com

William Rugolsky, Jr.
+1 212 486 4700
rugolsky@farance.com

ABSTRACT

This is a preliminary proposal for the inclusion of a C binding of language independent arithmetic (LIA-1) as defined in ISO 10967-1:1994. LIA-1 specifies a parameterized model of arithmetic computation. The purpose of LIA-1 is to provide a known environment in conforming implementations across platforms and languages for applications requiring numeric computation. Overall, the C binding of LIA-1 doesn't affect existing programs but new programs will achieve a higher degree of portability on LIA-1 systems. The impact of the changes are: adding some macros, adding a handful of library functions, and requiring the implementation to document certain features of its arithmetic. This proposal is in the early stages of development. It is intended to foster discussion of these features. Assuming that there is interest in this proposal, the next revision will include detailed standards wording.

1. PROBLEM STATEMENT

The following issues draw interest in including a C binding of LIA-1.

- There is a need for a common model of arithmetic for homogeneous and heterogeneous systems.
- A parameterized model allows adaptability across implementations. The C arithmetic model (`<limits.h>`) is parameterized somewhat, but the C arithmetic model has primarily been influenced by C implementations.
- LIA-1 provides more general arithmetic model.
- A better defined model would allow more portable numeric computation applications.
- WG14 should provide language independent features that other languages will be providing, i.e., direction from SC22.

2. LIA-1 OVERVIEW

The following excerpt is taken from the Introduction in LIA-1.

The Aims

Programmers writing programs that perform a significant amount of numeric processing have often not been certain how a program will perform when run under a given language processor. Programming language standards have traditionally been somewhat weak in the area of numeric processing, seldom providing an adequate specification of the properties of arithmetic data types, particularly floating point numbers. Often they do not even require much in the way of documentation of the actual arithmetic data types by a conforming language processor.

It is the intent of this part of ISO/IEC 10967 to help to redress these shortcomings, by setting out precise definitions of integer and floating point data types, and requirements for documentation. This is done in a way that makes as few presumptions as possible about the underlying machine architectures.

It is not claimed that this part of ISO/IEC 10967 will ensure complete certain of arithmetic behavior in all circumstances; the complexity of numeric software and the difficulties of analysing and proving algorithms are too great for that to be attempted. Rather, the requirements set forth here will provide a firmer basis that hitherto for attempting such analysis.

Hence the first aim of this part of ISO/IEC 10967 is to enhance the predictability and reliability of the behavior of programs performing numeric processing.

The second aim, which helps to support the first, is to help programming language standards to express the semantics of arithmetic data types. These semantics need to be precise enough for numeric analysis, but not so restrictive as to prevent efficient implementation of the language on a wide range of platforms.

The third aim is to help enhance the portability of programs that perform numeric processing across a range of different platforms. Improved predictability of behavior will aid programs designing code intended to run on multiple platforms, and will help in predicting what will happen when such a program is moved from one conforming language processor to another.

Note that this part of ISO/IEC 10967 does not attempt to ensure bit-fot-bit identical results when programs are transferred between language processors, or translated from one language into another. Programming languages and platforms are too diverse to make that a sensible goal. However, experience shows that diverse numeric environments can yield comparable results under most circumstances, and that with careful program design significant portability is actually achievable.

The Content

This part of ISO/IEC 10967 defines the fundamental properties of integer and floating point numbers. These properties are presented in terms of a parameterized model. The parameters allow enough variation in the model so that most platforms are covered, but when a particular set of parameter values is selected, and all required documentation is supplied, the resulting information should be precise enough to permit careful numerical analysis.

The requirements of this part of ISO/IEC 10967 cover three areas. First, the programmer must be given runtime access to the parameters and functions that describe the arithmetic properties of the platform. Second, the executing program must be notified when proper results cannot be returned (e.g., when a computed result is out of range or undefined). Third, the numeric properties of conforming platforms must be publicly documented.

The part of ISO/IEC 10967 focuses on the classical integer and floating point data types. Later parts will consider common mathematical procedures (part 2), complex numbers (part 3), and possibly additional arithmetic types such as fixed point.

Relationship to Hardware

ISO/IEC 10967 is not a hardware architecture standard. It makes no sense to talk about an ‘‘LIA machine’’. Future platforms are expected either to duplicate existing architectures, or to satisfy high quality architecture standards such as IEC 559 (also known as IEEE 754). The floating point requirements of this part of ISO/IEC 10967 are compatible with (and enhance) IEC 559.

This part of ISO/IEC 10967 provides a bridge between the abstract view provided by a programming language standard and the precise details of the actual arithmetic implementation.

The Benefits

Adoption and proper use of this part of ISO/IEC 10967 can lead to the following benefits.

Language standards will be able to define their arithmetic semantics more precisely without preventing the efficient implementation of their language on a wide range of machine architectures.

Programmers of numeric software will be able to assess the portability of their programs in advance. Programmers will be able to trade off program design requirements for portability in the resulting program.

Programs will be able to determine (at run time) the crucial numerical properties of the implementation. They will be able to reject unsuitable implementations, and (possibly) to correctly characterize the accuracy of their own results. Programs will be able to extract apparently implementation dependent data (such as the exponent of a floating point number) in an implementation independent way. Programs will be able to detect (and possibly correct for) exceptions in arithmetic processing.

End users will find it easier to determine whether a (properly documented) application program is likely to execute satisfactorily on their platform. The can be done by comparing the documented requirements of the program against the documented properties of the platform.

Finally, end users of numeric applications packages will be able to rely on the correct execution of those packages. That is, for correctly programmed algorithms, the results are reliable if and only if there is no notification.

3. IMPACT TO STANDARD C

This section provides an rough draft of the wording that would be added to the Standard to support LIA-1. The definitions of the parameters (e.g., `INT_MODULO`) have been omitted for the sake of clarity at this phase of review: most of the definitions are obvious or defined in LIA-1. Of course, the definitions would be included in future proposals and in the final Standards wording.

The following wording would be added to the Standard:

3.1 Language Independent Arithmetic <lia.h>

An implementation shall conform to all the requirements of LIA-1 (ISO 10967-1:1994) unless otherwise specified in this clause.

NOTE: The operations or parameters marked † are not part of Standard C and are enhancements required by LIA-1.

3.1.1 Boolean Type

The LIA-1 data type Boolean is implemented in the C data type `int` (1 == true and 0 == false).

3.1.2 Integral Types

The integral types `int`, `long`, `unsigned int`, and `unsigned long` conform to LIA-1.

NOTE: The conformity of `short` and `char` (signed or unsigned) is not relevant since values of these types are promoted to `int` (signed or unsigned) before computations are done.

3.1.2.1 LIA-1 Parameters

The parameters for the LIA-1 integer data types can be accessed by the following:

maxint	`INT_MAX`, `LONG_MAX`, `UINT_MAX`, `ULONG_MAX`.
minint	`INT_MIN`, `LONG_MIN`.
modulo	`INT_MODULO`†, `LONG_MODULO`†.

The parameter *bounded* is always true, and is not provided. The parameter *minint* is always 0 for the unsigned types, and is not provided for those types. The parameter *modulo* is always true for the unsigned types, and is not provided for those types.

3.1.2.2 LIA-1 Operations

The integer operations are the following:

addI	`x + y`.
subI	`x - y`.
mulI	`x * y`.

divI	`x / y`.
remI	`x % y`.
modaI	`modulo(x,y)`†, `lmodulo(x,y)`†.
modpI	No binding.
negI	`- x`.
absI	`abs(x)`, `labs(x)`.
signI	`sgn(x)`†, `lsgn(x)`†.
eqI	`x == y`.
neqI	`x != y`.
lssI	`x < y`.
leqI	`x <= y`.
gtrI	`x > y`.
geqI	`x >= y`.

where `x` and `y` are expressions of the same integral type.

The C Standard permits *divI* and *remI* (`/` and `%`) to be implemented using either round toward minus infinity (*divfI*) or toward zero (*divtI/remtI*). The implementation shall choose the same rounding for both and document the choice.

3.1.3 Floating Types

The floating types `float`, `double`, and `long double` conform to LIA-1.

3.1.3.1 LIA-1 Parameters

The parameters for a floating point data type can be accessed by the following:

r	`FLT_RADIX`.
p	`FLT_MANT_DIG`, `DBL_MANT_DIG`, `LDBL_MANT_DIG`.
emax	`FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDGL_MAX_EXP`.
emin	`FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.
denorm	`FLT_DENORM`†, `DBL_DENORM`†, `LDBL_DENORM`†.
iec_559	`FLT_IEC_559`†, `DBL_IEC_559`†, `LDBL_IEC_559`†.

The `*_DENORM` macros and `*_IEC_559` macros represent booleans and have values 1 or 0.

The derived constants for the floating types are accessed by the following:

fmax	`FLT_MAX`, `DBL_MAX`, `LDBL_MAX`.

fminN	`FLT_MIN`, `DBL_MIN`, `LDBL_MIN`.
fmin	`FLT_TRUE_MIN`†, `DBL_TRUE_MIN`†, `LDBL_TRUE_MIN`†.
epsilon	`FLT_EPSILON`†, `DBL_EPSILON`†, `LDBL_EPSILON`†.
rnd_error	`FLT_RND_ERR`†, `DBL_RND_ERR`†, `LDGL_RND_ERR`†.
rnd_style	`FLT_ROUNDS`.

3.1.3.2 LIA-1 Rounding Styles

The C Standard requires all floating types use the same radix and rounding style, so that only one identifier for each is provided in the LIA-1 binding.

The `FLT_ROUNDS` parameter corresponds to the LIA-1 rounding styles:

truncate	`FLT_ROUNDS == 0`.
nearest	`FLT_ROUNDS == 1`.
other	`FLT_ROUNDS != 0 && FLT_ROUNDS != 1`.

NOTE: — The definition of `FLT_ROUNDS` has been extended to cover the rounding style used in all LIA-1 operations, not just addition.

3.1.3.3 LIA-1 Operations

The floating point operations are:

addF	`x + y`.
subF	`x - y`.
mulF	`x * y`.
divF	`x / y`.
negF	`- x`.
absF	`fabsf(x)`†, `fabs(x)`, `fabsl(x)`†.
signF	`fsgnf(x)`†, `fsgn(x)`†, `fsgnl(x)`†.
exponentF	`exponf(x)`†, `expon(x)`†, `exponl(x)`†.
fractionF	`fractf(x)`†, `fract(x)`†, `fractl(x)`†.
scaleF	`scalef(x,n)`†, `scale(x,n)`†, `scalel(x,n)`†.
succF	`succf(x)`†, `succ(x)`†, `succl(x)`†.
predF	`predf(x)`†, `pred(x)`†, `precl(x)`†.
ulpF	`ulpf(x)`†, `ulp(x)`†, `ulpl(x)`†.
truncF	`trunctof(x,n)`†, `truncto(x,n)`†, `trunctol(x,n)`†.

roundF	`roundtof(x,n)`†, `roundto(x,n)`†, `roundtol(x,n)`†.
intpartF	`intprtf(x)`†, `intprt(x)`†, `intprtl(x)`†.
fractpartF	`frcprtf(x)`†, `frcprt(x)`†, `frcprtl(x)`†.
eqF	`x == y`.
neqF	`x != y`.
lssF	`x < y`.
leqF	`x <= y`.
gtrF	`x > y`.
geqF	`x >= y`.

where `x` and `y` are expressions of the same floating point type, and `n` is of type `int`.

NOTE: *scaleF* can be computed using the `ldexp` library function, only if `FLT_RADIX==2`.

NOTE: The Standard C function `frexp` differs from *exponentF* in that no notification is raised when the argument is 0.

3.1.3.4 LIA-1 Indicators

The following indicators shall be provided a one method of notification (see LIA-1 subclause 6.1.2).

integer_overflow	`INT_OVERFLOW`†.
floating_overflow	`FLT_OVERFLOW`†.
underflow	`UNDERFLOW`†.
undefined	`UNDEFINED`†.

The values representing individual indicators shall be distinct non-negative powers of two. The empty set is denoted by `0`. Other indicator subsets are named by combining individual indicators using bit-or. For example, the LIA-1 indicator subset

```
{floating_overflow, underflow, integer_overflow}
```

would be denoted by the expression

```
FLT_OVERFLOW | UNDERFLOW | INT_OVERFLOW
```

The indicator interrogation and manipulation operations are:

set_indicators	`set_indicators(i)`†.
clear_indicators	`clear_indicators(i)`†.
test_indicators	`test_indicators(i)`†.

current_indicators `current_indicators()`†.

where `i` is an expression of type `unsigned int` representing an LIA-1 indicator subset.

The implementation shall provide an alternative of notification through termination with a ‘‘hard-to-ignore’’ message (see LIA-1 subclause 6.1.3).

3.1.4 Type Conversions

LIA-1 operations shall be provided in all floating types.

The LIA-1 type conversions are the following type casts:

cvtF→I `(int) x`, `(long) x`, `(unsigned int) x`, `(unsigned long) x`.

cvtI’→I `(int) x`, `(long) x`, `(unsigned int) x`, `(unsigned long) x`.

cvtI→F `(float) x`, `(double) x`, `(long double) x`.

cvtF’→F `(float) x`, `(double) x`, `(long double) x`.

3.1.5 Outstanding Issues

The C Standard requires that float to integer conversions round toward zero. An implementation that wishes to conform to LIA-1 must use round to nearest for conversions to a floating point type.

4. CONCLUSIONS

The LIA-1 features should be included in C9X. The following are the steps to formal inclusion in C9X.

1. Review of this paper.
2. Receive comments on the general features of LIA-1.
3. Receive comments on specific features of the C binding.
4. Receive comments on the ability of members' implementations to conform to the C binding.
5. Provide detailed definition of new macros and functions required by LIA-1.
6. Review of detailed version of this paper.

CONTENTS