A Proposal to add maximum significant decimal digits macros to the C Standard Library.

Document number: JTC 1/SC22/WG14/N1171
Date: 2006-04-04, version 2
Project: Languages C (and C++)

Reply to: Paul A. Bristow, pbristow@hetp.u-net.com, J16/04-0108

References:

1 A Proposal to add a maximum significant decimal digits value
to the C++ Standard Library Numeric limits, Paul A. Bristow
Document number: JTC 1/SC22/WG21/N1822=05-0082
http://www2.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1822.pdf
Revised version 4 as Document number: JTC 1/SC22/WG21/N2005=06-0075
on 2006-04-12.
2 C ISO/IEC 9899:1999.
3 C++ ISO/IEC IS 14882:1998(E).
4 William Kahan http://http.cs.berkley.edu/~wkahan/ieee754status/ieee754.ps
5 JTC 1/SC22/WG14/N1151, version 1 of this document.

Introduction

Following favourable progress on my proposal above to add to the C++ Standard Library, I think it would be rational to add equivalent macros to the C equivalent. These values could of course be used by C++ to efficiently implement std::numeric_limits, as well as providing an compatible equivalent in purely C programs.

The case for the these values has been discussed in detail in the above paper, but a brief summary follows.

C++ provides numeric limits 18.2.1 including

```
numeric_limits<Floating-Point Type>::digits10
```

also available via (and often implemented using) C macros FLT_DIG, DBL_DIG, LDBL_DIG.

The macro stores the number of decimal digits that the type can represent without change.

In effect, it is the number of decimal digits GUARANTEED to be correct (after rounding).

While useful, this does not provide another value, often more useful, the number of **potentially significant** decimal digits that the type can represent. This number of decimal digits is necessary to avoid misleading display of two floating-point numbers which only differ by one or a few least significant bits, but are represented identically.

It is also essential to use this number of decimal digits if it required to convert and save a binary floating point value as a decimal digit string and then restore to get **exactly** the same internal binary floating point value. For example, this is necessary to use Boost.Serialization. (This assumes, of course, compatible or identical internal representations – a separate issue).

For example, if using IEEE 754/IEC559 32-bit floating-point float values, and FLT_MANT_DIG is 6,

a number declared as

```
float f = 3.145900F;
```

might be displayed using the precision(`FLT_DIG == 6`) as

“`3.14590`”

But the successor, `nextafterf(3.145900F, 1.),` a single bit different, and so definitely not equal, will also display as “`3.14590`”, so log files may display a most misleading, and unhelpful, output like:

```
“3.14590” != “3.14590”
```

Whereas if the proposed `FLT_MAXDIG10` whose value is 9 is used, the output

```
“3.14590001” != “3.14590025”
```

that is much less confusing, especially to the majority of readers whose understanding of the limitations of floating-point accuracy is incomplete.

C99 already has

```
DECIMAL_DIG defined as ceil(1+precision*log10(radix)).
```

However the precision is the **maximum** precision provided by the implementation, usually long double.

This proposal is to provide **separate macros for all precisions supported by the implementation**. This is useful is to avoid outputting low precision types with many uninformative decimal digits – a significant inefficiency and a confusing nuisance to readers.

For base 2 systems, values for these macros are usually conveniently derived from the number of significand (mantissa) binary digits, `significand_digits` defined by

```
FLT_MANT_DIG, DBL_MANT_DIG or LDBL_MANT_DIG
```

using the formula

```
max_digits10 = 2 + significand_digits * 301/1000 // if 16-bit integers
```

else

```
max_digits10 = 2 + significand_digits * 30103UL/100000UL
```

For example, for systems with 32-bit integers:

```
#define FLT_MAXDIG10 (2+(FLT_MANT_DIG * 30103UL)/100000UL)
#define DBL_MAXDIG10 (2+ (DBL_MANT_DIG * 30103UL)/100000UL)
```

```
#define LDBL_MAXDIG10 (2+ (LDBL_MANT_DIG * 30103UL)/100000UL)
```

which yield the following values on typical implementations:

```
32-bit IEEE 754 float       FLT_DIG 6, FLT_MAXDIG10 9
64-bit IEEE 754 double      DBL_DIG 15, DBL_MAXDIG10 17
80-bit IEEE 754 long double LDBL_DIG 19, LDBL_MAXDIG10 21
```

For 16-bit integer systems:
if `DBLMANT_DIG` is 53 (for IEEE 64-bit doubles) then 53 * 301 = 15953, but larger and more accurate approximations, like 3010/10000 or 30103/100000, would overflow 16-bit integers.

For 32-bit integer systems:
the more accurate ratio 30103UL/100000UL is preferred to give the correct values for well beyond 256 significand bits.

Significand bit values where .3 and .30103 produce different values:
103, 113, 123, 133, 143, 153, 163, 173, 183, 193, 196, 203, 206, 213, 216, 223, 226, 233, 236, 243, 246, 253, 256, 263, 266, 273, 276, 283, 286, 293, 296, 299, ...

showing that using 301/1000 would give an incorrect result for these significand bits but 30103UL/100000UL will be correct. Using UL further reduces the risk of overflow.

For user defined floating-point types, usually to implement very high precision not available in hardware, similar (but of course non-standard) macros can be defined.

Acknowledgements

Expert comments by Fred J. Tydeman.

C Library Proposed Text Additions

Three new macros to be inserted just after `FLT_DIG, DBL_DIG, LDBL_DIG`

"

```
FLT_MAXDIG10 for float
DBL_MAXDIG10 for double
LDBL_MAXDIG10 for type long double
```

The smallest number of base 10 digits required to ensure that values which differ by only one smallest (often binary) unit in the last place (ulp) are always differentiated.

For base 10 systems, the values are:

```
precision*log10(radix)
```

and for all other bases:

```
ceil(1+precision*log10(radix))
```

"