Document: WG14/N1215
Date: 2007/03/26
Project: TR 24732
Authors: Jim Thomas, Rich Peterson
Reply to: Rich Peterson <Rich.Peterson@hp.com>

Subject: strtodxx, scanf, and printf specification to support 754R

754R requires conversions from internal decimal formats to external decimal character sequences and back that preserve the representation in the internal format. This document proposes changes for the WG14/N1201 (TR 24732 draft of 2006/11/10)specification of strtodxx, scanf, and printf to support this requirement.

Also, 754R requires conversion that correctly rounds all external decimal character sequence numbers to internal decimal formats. The proposal here covers this requirement too.

Background: The finite decimal floating-point numbers are represented by triples (s, c, q), where s is the sign (1 or -1), c is the non-negative integer coefficient, q is the quantum exponent, and the value of the representation is s * c * 10^q. Note that multiple representations may have the same numeric value, e.g. (1, 1200, -2) = (1, 120, -1) = (1, 12, 0), but they are treated differently in the arithmetic: the representation of operands affects determination of the representation of the result. Another paper proposes that the TR include a change to C99 that describes this view of the model.

To support 754R there needs to be a reasonably convenient way to print a finite decimal floating-point number as a decimal character sequence and then scan that character sequence back into an object of the same decimal floating type so that the original representation (s, c, q) is preserved.

Here is a specification for strtodxx (employed by scanf) to determine a representation in a straightforward way. It applies to all floating-point specifiers and requires no syntax beyond what the TR currently proposes. Specifying correct rounding as required by 754R eliminates the need for recommended practice for cases of imperfect rounding.

Suggested TR changes for strtodxx and scanf: In 9.6, delete the proposed recommended practice and replace the proposed 7.20.1.5[#5] with the text below. Make the similar change in 9.7, replacing proposed 7.24.4.1.3 [#5].

If the subject sequence has the expected form for a floating-point number, then the result shall be correctly rounded as specified in 754R.

If the sequence is negated, the sign s is set to -1, else s is set to 1.

The coefficient c and the quantum exponent q of a finite converted floating-point number are determined from the subject sequence as follows:

-- The fractional-constant or digit-sequence and the exponent-part (if any) are extracted from the subject sequence. If there is an exponent-part, then q is set to the value of sign[opt] digit-sequence in the exponent-part. If there is no exponent-part, q is set to 0.

-- If there is a fractional-constant, q is decreased by the number of digits to the right of the decimal point and the decimal point is removed to form a digit-sequence.

-- c is set to the value of the digit-sequence (after any decimal point has been removed).

-- Rounding required because of insufficient precision or range in the type of the result will round c to the full precision available in the type, and will adjust q accordingly within the limits of the type, provided the rounding does not yield an infinity (in which case an appropriately signed internal representation of infinity is returned). If the full precision of the type would require q to be smaller than the minimum for the type, then q is pinned at the minimum and c is denormalized accordingly, perhaps to zero.

Examples:

Following are subject sequences of the decimal form and the resulting triples (s, c, q) produced by strtod64. Note that for _Decimal64, the precision (maximum coefficient length) is 16 and the quantum exponent range is -398 <= q <= 369.

```
 "0"            ( 1,0,0)
 "0.00"         ( 1,0,-2)
 "123"          ( 1,123,0)
 "-123"         (-1,123,0)
 "1.23E3"       ( 1,123,1)
 "1.23E+3"      ( 1,123,1)
 "12.3E+7"      ( 1,123,6)
 "12.0"         ( 1,120,-1)
 "12.3"         ( 1,123,-1)
 "0.00123"      ( 1,123,-5)
 "-1.23E-12"    (-1,123,-14)
 "1234.5E-4"    ( 1,12345,-5)
 "-0"           (-1,0,0)
 "-0.00"        (-1,0,-2)
 "0E+7"         ( 1,0,7)
 "-0E-7"        (-1,0,-7)
 "12345678901234567890" ( 1, 1234567890123457, 4) or ( 1,
1234567890123456, 4) depending on rounding mode
 "1234E-400"    ( 1, 12, -398) or ( 1, 13, -398) depending on rounding
mode
 "1234E-402"    ( 1, 0, -398) or (1, 1, -398) depending on rounding
mode
```

In 9.7, make the same replacement (above) for 7.24.4.1.3 [#5].

The determination of representation described above is intended to be equivalent to the following (copied from http://www2.hursley.ibm.com/decimal/daconvs.html#reftonum), but adapted to the context of the C standard and the TR.

* The decimal-part and exponent-part (if any) are then extracted from the string and the exponent-part (following the indicator) is converted to form the integer /exponent/ which will be negative if the exponent-part began with a '-' sign. If there is no exponent-part, the /exponent/ is set to 0.
  If the decimal-part included a decimal point the /exponent/ is then reduced by the count of digits following the decimal point (which may be zero) and the decimal point is removed. The remaining string of digits has any leading zeros removed (except for the rightmost digit) and is then converted to form the /coefficient/ which will be zero or positive.
  A numeric string to finite number conversion is always exact unless there is an underflow or overflow (see below) or the number

of digits in the decimal-part of the string is greater than the
/precision/ in the context. In this latter case the coefficient
will be rounded (shortened) to exactly /precision/ digits, using
the /rounding/ algorithm, and the /exponent/ is increased by the
number of digits removed. The /rounded/ and other flags may be
set, as if an arithmetic operation had taken place (see below).
If the value of the /adjusted exponent/
<http://www2.hursley.ibm.com/decimal/damodel.html#reffinite> is
less than E_min , then the number is subnormal
<http://www2.hursley.ibm.com/decimal/damodel.html#refsubnorm>. In
this case, the calculated coefficient and exponent form the
result, unless the value of the /exponent/ is less than E_tiny ,
in which case the /exponent/ will be set to E_tiny
<http://www2.hursley.ibm.com/decimal/damodel.html#refsubnorm>, and
the coefficient will be rounded (possibly to zero) to match the
adjustment of the exponent, with the /sign/ remaining as set
above. If this rounding gives an inexact result then the Underflow
exceptional condition
<http://www2.hursley.ibm.com/decimal/daexcep.html#refunderf> is
raised.
If (after any rounding of the coefficient) the value of the
/adjusted exponent/ is larger than E_max
<http://www2.hursley.ibm.com/decimal/damodel.html#reffinite>, then
an exceptional condition (overflow) results. In this case, the
result is as defined under the Overflow exceptional condition
<http://www2.hursley.ibm.com/decimal/daexcep.html#refoverf>, and
may be infinite. It will have the /sign/ as set above.

Specification to meet the 754R requirement for printf is trickier. The C committee has considered leaving representation-preserving formatting to the user, who could employ the '*' mechanism for specifying a precision that would distinguish the representation. For example, %.10De would preserve a _Decimal64 representation whose coefficient had length 11. But how to do this for an arbitrary representation? Given a function that returns the quantum exponent (for _Decimal64), say int quantexpd64(_Decimal64), the coefficient length can be computed as ilogbd64(x) - quantexpd64(x) + 1. (We could provide a function that returned the coefficient as a string, or the coefficient length itself.) However, special cases -- zeros, infinities, and NaNs -- would have to be handled separately, and more effort would be required for simpler style f formatting of small integers and short fractions. Seems like too much to burden the user with.

The complexity described above shows a need for a new or adapted specifier, or other means, for a convenient way for users to print a decimal floating number and preserve its representation. The proposal below is to adapt the decimal-modified %g specifiers, though a new specifier would also do. Another approach would be to provide a function that returned a representation-preserving string, which could be printed with %s.

Since many mappings of internal decimal representations to external decimal character sequences would combine with the preceding scanf specification to meet the 754R requirement, specification of printf formatting is subject to considerations of style and what users prefer. The formatting specified above is similar to that described in http://www2.hursley.ibm.com/decimal/daconvs.html#reftostr, but adapted to the context and style of the C standard and the TR.

Suggested TR changes for printf:

Append to TR 9.5, under "Suggested changes to C99", the following:

In 7.19.6.1 and in 7.24.2.1, under g, G conversion specifiers, append:

If an H, D, or DD modifier is present and the precision is missing, then for a decimal floating argument represented
by a triple of integers (s, c, q), where n is the number of digits in the coefficient c,

-- if 0 >= q >= -(n+5), use style f formatting with formatting precision equal to -q,

-- otherwise, use style e formatting with formatting precision equal to n - 1, except if c = 0 then the digit-sequence in the exponent-part shall have the value q (rather than 0).

Examples:

Following are representations of _Decimal64 arguments as triples (s, c, q) and the corresponding character sequences printf produces with %Dg:

```
( 1, 123, 0)      123
(-1, 123, 0)      -123
( 1, 123, -2)     1.23
( 1, 123, 1)      1.23e+03
(-1, 123, 1)      -1.23e+03
( 1, 123, -8)     0.00000123
( 1, 123, -9)     1.23e-07
( 1, 1234567890123456, 0)      1234567890123456
( 1, 1234567890123456, 1)      1.234567890123456e+16
( 1, 1234567890123456, -1)     123456789012345.6
( 1, 1234567890123456, -21)    0.000001234567890123456
( 1, 1234567890123456, -22)    1.234567890123456e-07
( 1, 0, 0)        0
(-1, 0, 0)        -0
( 1, 0, -6)       0.000000
( 1, 0, -7)       0e-07
( 1, 0, 2)        0e+02
( 1, 5, -6)       0.000005
( 1, 50, -7)      0.0000050
( 1, 5, -7)       5e-07
```
