Standard Decimal Arithmetic Extended Specification

9th August 2000

Mike Cowlishaw

IBM Fellow
IBM UK Laboratories
mfc@uk.ibm.com

Draft – Version 0.30

Table of Contents

Introduction

This document extends the general purpose decimal arithmetic defined in the **Standard Decimal Arithmetic Specification**[1] (the *base specification*). A correct implementation of the combined base and extended specifications will conform to the decimal arithmetic defined in the ANSI standard X3.274-1996[2] and will also conform to the ANSI/IEEE standard 854-1987.[3] This document is meaningful only in the context of the base specification; it is not in itself a complete specification.

The primary audience for this document is implementers, so examples and explanatory material are included. Explanatory material is identified as Notes, Examples, or footnotes, and is not part of the formal specification. Additional explanatory material can be found in the article *A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic*.[4]

For further background details, including the base specification and a suggested concrete representation which conforms to IEEE 854-1987, please see the material at the associated web site: `http://www2.hursley.ibm.com/decimal`

Appendix A (see page 18) summarizes the changes to this specification since the first public draft.

Comments on this draft are welcome. Please send any comments, suggestions, and corrections to the author, Mike Cowlishaw (`mfc@uk.ibm.com`).

Acknowledgements

This document, in conjunction with the base specification, is in effect an embodiment of IEEE 854. It therefore owes a great debt to the authors of that standard. Special thanks for his contribution to this work are due to Fred Ris.

1 See `http://www2.hursley.ibm.com/decimal/decspec.html`

2 *American National Standard for Information Technology – Programming Language REXX, X3.274-1996*, American National Standards Institute, New York, 1996.

3 IEEE 854-1987 – *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1987.

4 by W. J. Cody *et al*, published in the IEEE Micro magazine, August 1984, pp86–100.

Scope

Objectives

This document extends the general purpose decimal arithmetic defined in the **Standard Decimal Arithmetic Specification**[5] (the *base specification*). A correct implementation of the combined base and extended specifications will conform to the decimal arithmetic defined in the ANSI standard X3.274-1996[6] and will also conform to the ANSI/IEEE standard 854-1987.[7]

Inclusions

This specification defines the following:

- Additional constraints and values for decimal numbers
- Additional arithmetical operations on decimal numbers
- Additional context information which alters the results of operations, and default contexts
- Additional constraints and rules for exceptional conditions.

Exclusions

This specification does not define the following:

- Items already defined as requirements in the base specification
- Concrete representations (storage format) of decimal numbers
- The means by which operations are effected
- Concrete representations (storage format) of context information

[5] See `http://www2.hursley.ibm.com/decimal/decspec.html`

[6] *American National Standard for Information Technology – Programming Language REXX, X3.274-1996*, American National Standards Institute, New York, 1996.

[7] IEEE 854-1987 – *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1987.

The Arithmetic Model

This specification extends the model of decimal arithmetic described in the base specification. New parameters and constraints are added to the abstract representation of numbers and context, and default contexts are defined.

As in the base specification, this extended specification does not define the *concrete representation* (specific layout in storage, or in a processor's register, for example) of numbers or context.

Abstract representation of numbers

In addition to the three parameters already defined (*sign*, *integer*, and *exponent*), *numbers* must be able to represent one of three named *special values*:

1. *infinity* – a value representing an infinitely large number (∞, see IEEE 854 §6.1)
2. *quiet NaN* – a value representing undefined results ("Not a Number") which does not cause an invalid operation condition. It is recommended that additional diagnostic information be associated with quiet NaNs (see IEEE 854 §6.2)
3. *signaling NaN* – a value representing undefined results ("Not a Number") which will cause an invalid operation condition if used in any operation defined in this specification (see IEEE 854 §6.2).

When a number has one of these special values, its *integer* and *exponent* are undefined.[8] The *sign*, however, is significant (that is, there can be both positive and negative infinity and NaNs).

For this specification, an additional constraint applies to the *exponent*:

- E_{limit} must be greater than $5 \times$ `ilength`, where `ilength` is the length of the *integer* in decimal digits (see IEEE 854 §3.1).

Notation

In addition to the triad notation of [*sign*, *integer*, *exponent*] introduced in the base specification, duples are used to indicate the special values.

[8] Typically, in a concrete representation, certain out-of-range values of the exponent are used to indicate the special values, and the integer is used to carry additional diagnostic information for quiet NaNs.

These have the form [*sign*, *special–value*], where the *sign* is indicated as before, and the *special–value* is one of `inf`, `qNaN`, or `sNaN`, representing *infinity*, *quiet NaN*, or *signaling NaN*, respectively.

So, for example, the duple `[1,inf]` represents the number $-\infty$, and the duple `[0,qNaN]` represents a non-negative quiet NaN.

Abstract representation of context

The abstract representation of *context* is extended so that it comprises the following parameters:

flags and trap-enablers

The exceptional conditions (see page 15) are grouped into six *signals*, which can be controlled individually. The context contains a *flag* (which is either 0 or 1) and a *trap–enabler* (which also is either 0 or 1) for each signal.

For each of the six signals, the corresponding flag is set to 1 when the signal occurs. It is only reset to 0 by explicit user action.

For each of the six signals, the corresponding trap-enabler indicates the action to be taken when the signal occurs (see IEEE 854 §7). If 0, a defined result is supplied, and execution continues (for example, an overflow is perhaps converted to a positive or negative infinity). If 1, then execution of the operation is ended and control passes to a “trap handler”. The trap handler will have access to the *trap–result* (see below) which is the defined result from the condition that caused the exception.

The six signals are:

invalid-operation

raised when a result would be undefined or impossible

division-by-zero

raised when a non-zero dividend is divided by zero

overflow

raised when the exponent of a result is too large to be represented

underflow

raised when the exponent of a result is too small to be represented

inexact

raised when a result is not exact, or overflows or underflows without being trapped

lost-digits

raised when the *lost–digits* condition is detected.

The *lost–digits* trap-enabler is the same parameter as the *lost–digits* context parameter in the base specification.

This specification does not define the means by which flags and traps are reset or altered, respectively, or the means by which traps are effected.[9]

trap-result

A number which is the result to be made available to a trap handler. Its value is undefined except in a trap handler, after an exceptional condition is trapped.

special-values

A value which must be either 0 or 1. If 1, the three special values will be accepted by arithmetic operations, the sign of the value –0 is preserved in the results of arithmetic operations, and extra checking is performed on the length of operands. If 0, the special values are not permitted as operands for arithmetic, and the *sign* of a zero value result is always 0.[10]

precision

This sets the precision of arithmetic operations, as defined in the base specification. Additional constraints and recommendations apply:

- An implementation must designate a precision to be known as *single precision* (see IEEE 854 §3.2.1). This must be greater than 5 (see IEEE 854 §3.1) and within the range of implemented precisions.[11]
- An implementation may also designate a precision to be known as *double precision*, which must be within the range of implemented precisions (see IEEE 854 §3.2.2). If a double precision is designated, then the following constraints apply:
    - If the value of *single precision* is given by P_s, and the value of *double precision* is given by P_d, then P_d must be greater than or equal to $2 \times P_s + 1$ (see IEEE 854 §3.2.2).
    - The maximum *exponent* (E_{limit}) at the designated single precision must be at least 1 less than the E_{limit} at double precision, divided by 8 (see IEEE 854 §3.2.2).[12]

  If these constraints cannot be implemented (for example, an implementation may support very large exponents and not be able to have different exponent limits for differing precisions), then a double precision must not be designated.

[9] IEEE 854 suggests that there be a mechanism allowing traps to return a substitute result to the operation that raised the exception, but this may not be possible in some environments (including some object-oriented environments).

[10] When 0, this parameter (together with appropriately set trap enablers and the use of the **to–number** operation), can be used to ensure that numbers with special values or the value −0 can never occur, as in the base specification. Similarly, this parameter should be set to 1 for IEEE 854 compliance.

[11] This is the "narrowest basic precision" described in IEEE 854 §3.2.1. Strictly speaking, *single precision* should be the narrowest precision supported; however it is assumed that when precision is fully variable the intent of IEEE 854 is that the designation applies to the narrowest *default* precision – the programmer is permitted to specify a narrower precision explicitly.

[12] This constraint is very slightly tighter than that defined by IEEE 854, which specifies that E_{limit} for double be greater than or equal to $8 \times E_{limit}$ for double, plus 7. Given the requirement for human-oriented limits, in the base specification, it is suggested that the E_{limit} for single be one tenth of, or one digit shorter than, the E_{limit} for double.

rounding

This sets the rounding algorithm to be used by arithmetic operations, as defined in the base specification. Additional constraints apply:

- The *round–half–even* algorithm must be supported.
- The following additional rounding algorithms are defined (see IEEE 854 §4.2), and must be supported:

  round-ceiling

  (Round toward +∞.) If all of the discarded digits are zero or if the *sign* is 1 the result is unchanged. Otherwise, the result should be incremented by 1 (rounded up). If this would cause overflow then the result will be `[0,inf]`.

  round-down

  (Round toward 0.) The discarded digits are ignored; the integer is always left unchanged.

  round-floor

  (Round toward –∞.) If all of the discarded digits are zero or if the *sign* is 0 the result is unchanged. Otherwise, the sign is 1 and the integer should be incremented by 1. If this would cause overflow then the result will be `[1,inf]`.

Notes:

1. For completeness, implementations may wish to offer two further rounding modes: *round–half–down* (round to nearest, where a 0.5 case is rounded down) and *round–up* (round away from zero).
2. The setting of *precision* may be used to reduce a result from double to single precision, using the **plus** operation. This meets the requirements of IEEE 854 § 4.3.

Default contexts

This specification defines two *default contexts*, which define suitable context settings for base arithmetic (as defined in the base specification) or the extended arithmetic required by IEEE 854. It is recommended that the default contexts be easily selectable by the user.

Base default context

In the *base default context*, the parameters are set as follows:

- *flags* – all set to 0
- *trap–enablers* – *inexact* and *lost–digits* are set to 0; the others are all set to 1
- *trap–result* – is undefined; it is recommended that it be set to `[0,0,0]`
- *special–values* – is set to 0
- *precision* – is set to 9

- *rounding* – is set to *round–half–up*

Note that with these parameters, any operation that completes will have a numeric value (that is, not a special value), and zero will have a *sign* of 0, as in the base specification.

Extended default context

In the *extended default context*, the parameters are set as follows:

- *flags* – all set to 0
- *trap–enablers* – all set to 0 (IEEE 854 §7)
- *trap–result* – is undefined; it is recommended that it be set to `[0,qNaN]`
- *special–values* – is set to 1 (IEEE 854 §1)
- *precision* – is set to the designated *single precision*
- *rounding* – is set to *round–half–even* (IEEE 854 §4.1)

It is recommended that if a *double precision* is designated then a third *extended double default context* be provided, with the same settings as the extended default context except that the *precision* is set to the double precision.

Conversions

This section adds new rules and a new conversion operation to the base specification (see IEEE 854 §5.6).

It is recommended that implementations also provide conversions to and from binary floating point or integer numbers, if appropriate (that is, if such encodings are supported in the environment of the implementation). It is suggested that such conversions be exact, if possible (that is, when converting from binary to decimal), or alternatively give the same results as converting using an appropriate string representation as an intermediate form.

It is also recommended that if a number is too large to be converted to a given binary integer format then an exceptional or error condition be raised, rather than losing high-order significant bits (decapitating).

Notes

1. The **to–number** operation is unchanged by this specification.
2. The setting of *precision* may be used to convert a number from any precision to any other precision, using the **plus** operation. This meets the requirements of IEEE 854 §5.3.
3. Integers are a proper subset of numbers, hence no conversion operation from an integer to a number is necessary. Conversion from a number to an integer is effected by using the **round–to–integer** operation (see page 13). This meets the requirements of IEEE 854 §5.4 and §5.5.

Numeric string syntax

The syntax for *numeric strings* is extended to allow for the special values of numbers, by replacing the final production (`numeric–string`) by:

```
special–value  ::=  'NaN' | 'NaNq' | 'Infinity' | 'Inf'
numeric–value  ::=  decimal–part [exponent–part] | special–value
numeric–string ::=  [sign] numeric–value
```

where the characters in the strings accepted for `special–value` may be in any case.

This does not affect the **to–number** operation, which only accepts numeric strings as defined in the base specification.

Examples:

Some numeric strings are:

```
        "Inf"         /* The same as Infinity          */
        "–infinity"   /* The same as –Inf              */
        "NaN"         /* Not–a–Number                  */
```

to-scientific-string and to-engineering-string

These operations are extended to allow string representations of numbers which have special values; both provide the same results in these cases, following the rules:

- If the *special–value* is *signaling NaN* then the resulting string is “`NaN`”.
- If the *special–value* is *quiet NaN* then the resulting string is “`NaNq`”.
- If the *special–value* is *infinity* then the resulting string is “`Infinity`”.
- As with other numbers, if the *sign* of the number is 1 then in all the above cases the string is preceded by a “`–`” character. Otherwise (the *sign* is is 0) no sign character is prefixed.

Examples:

For each abstract representation [*sign*, *special–value*] on the left, the resulting string is shown on the right.

```
    [0,inf]       "Infinity"
    [1,inf]       "–Infinity"
    [0,sNaN]      "NaN"
    [1,sNaN]      "–NaN"
    [0,qNaN]      "NaNq"
```

Notes

1. The values *quiet NaN* and *signaling NaN* are distinguished in string form in order to preserve the one-to-one mapping between abstract representations and the **to–scientific–string** representation.
2. IEEE 854 allows additional information to be suffixed to the string representation of special values. Any such suffixes are not permitted by this specification (again, to preserve the one-to-one mapping). It is suggested that if additional information is held in a concrete representation then a separate mechanism or operation is provided for accessing that information.

to-extended-number – conversion from numeric string

This operation extends the **to–number** operation of the base specification to accept numeric string representations of special values. It follows the definition of the **to–number** operation, and in addition:

- The string “`NaN`”, optionally preceded by a sign character and independent of case, will be accepted by **to–extended–number** and converted to *signaling NaN*.

- The string “NaNq”, optionally preceded by a sign character and independent of case, will be accepted by **to–extended–number** and converted to *quiet NaN*.
- The strings “Infinity” and “Inf”, optionally preceded by a sign character and independent of case, will be accepted by **to–extended–number** and converted to *infinity*.
- In all three cases above, the *sign* of the number is set to 1 if the string is preceded by a “–”. Otherwise the *sign* is set to 0.
- If the *integer* is 0 and the numeric string starts with a “–” sign then the *sign* of the number will be 1. That is, the sign of a negative zero is preserved by **to–extended–number**.

Examples:

For each string on the left, the resulting abstract representation [*sign*, *integer*, *exponent*] or [*sign*, *special–value*] is shown on the right.

```
"0"            [0,0,0]
"0.00"         [0,0,–2]
"123"          [0,123,0]
"–123"         [1,123,0]
"1.23E3"       [0,123,1]
"1.23E+3"      [0,123,1]
"12.3E+7"      [0,123,6]
"12.0"         [0,120,–1]
"12.3"         [0,123,–1]
"0.00123"      [0,123,–5]
"–1.23E–12"    [1,123,–14]
"1234.5E–4"    [0,12345,–5]
"–0"           [1,0,0]
"–0.00"        [1,0,–2]
"inf"          [0,inf]
"+inFiniTy"    [0,inf]
"–Infinity"    [1,inf]
"–NAN"         [1,sNaN]
"NaNQ"         [0,qNaN]
```

Note: As usual, an implementation does not have to make operations logically distinct, provided that the function of each defined operation is available. For example, in a software implementation, the **to–number** and **to–extended–number** operations could be implemented as a single method, taking a parameter which switches the operation.

Arithmetic operations

This section adds new rules and operations to the base specification, notably to permit the production and handing of special values and –0.

The same notation for examples is used as in the Arithmetic operations section of the base specification.

Arithmetic operation rules

The following additional rules apply to all arithmetic operations:

- If *special–values* is 0, then special values are not permitted as an operand to an arithmetic operation; an Invalid operation exceptional condition (see page 15) results in this case. If *special–values* is 1, then special values are permitted as operands to an arithmetic operation (see IEEE 854 §6).
- Arithmetic using the special value *infinity* follows the usual rules, where `[1,inf]` is less than every finite number and `[0,inf]` is greater than every finite number. Under these rules, a infinite result is always exact. Certain uses of infinity raise exceptional conditions (see page 15), which are listed under each condition.
- *signaling NaNs* always raise the Invalid operation condition when used as an operand to an arithmetic operation.
- The result of any arithmetic operation which has an operand which is a NaN (a *quiet NaN*, or *signaling NaNs* when the *invalid–operation* trap enabler is 0) is `[0,qNaN]`. In this case, the signs of the operands are ignored (the following rules do not apply).
- The *sign* of the result of a multiplication or division will be 1 only if the operands have different signs.
- The *sign* of the result of an addition or subtraction will be 1 only if the result is less than zero, except for the special case below where the result is –0.
- If *special–values* is 0, a zero result is always `[0,0,0]`, as in the base specification. If *special–values* is 1, then a result of `[1,0,0]` is possible. This can occur under the following conditions only:
  - the operation is an addition or subtraction and the result has an *integer* of 0 and the *rounding* is *round–floor*, unless both operands to the addition or subtraction had an *integer* of 0 and a *sign* of 0
  - the operation is a multiplication or division and the result has an *integer* of 0 and the signs of the operands are different.

    - the operation is **square–root** (see below) and the operand has a value of –0.

- If the length of the *integer* of an operand is greater than *precision* then if the *lost–digits* condition is not raised and *special–values* is 1 then an Invalid operation condition is raised.[13]

Examples:

For these examples, *special–values* has the value 1, and the *divide–by–zero* trap enabler has the value 0.

```
add('Infinity', '1')        ==>  'Infinity'
add('NaNq', '1')            ==>  'NaNq'
subtract('1', 'Infinity')   ==>  '-Infinity'
multiply('-1', 'Infinity')  ==>  '-Infinity'
multiply('-1', '0')         ==>  '-0'
divide('-1', 'Infinity')    ==>  '-0'
divide('1', '0')            ==>  'Infinity'
divide('1', '-0')           ==>  '-Infinity'
divide('-1', '0')           ==>  '-Infinity'
```

Notes:

1. Quiet NaNs are permitted to propagate diagnostic information pertaining to the origin of the NaN (see IEEE 854 §6.2). Any such diagnostic information, and the means by which it is propagated, is outside the scope of this specification.
2. Overflow and underflow may result in infinite or zero results if the corresponding trap is not enabled, as defined under the relevant Exceptional condition (see page 15).
3. The rules above imply that the **compare** operation can now return *quiet NaN* as a result, which indicates an “unordered” comparison (see IEEE 854 §5.7).
4. As stated in the base specification, an implementation may use the **compare** operation “under the covers” to implement a closed set of comparison operations (greater than, equal, *etc.*) if desired. In this case, the additional constraints in IEEE 854 §5.7 will apply; they are not repeated here.

square-root

square–root takes one operand, which must be greater than or equal to 0. If the value of the operand is –0 then the result is `[1,0,0]`.

Otherwise, the result is the exact square root of the operand, rounded according to the settings of *precision* and *rounding*. Finally, any insignificant trailing zeros are removed (that is, if the *integer* is a multiple of a power of ten then it is divided by that power of ten and the *exponent* increased accordingly).

[13] This rule is required to comply with IEEE 854 §5.1, second sentence.

Examples:

For these examples, *special–values* has the value 1, and *precision* is 9.

```
square–root('0')     ==> '0'
square–root('-0')    ==> '-0'
square–root('1.00')  ==> '1'
square–root('7')     ==> '2.64575131'
```

remainder-near

remainder–near takes two operands. If the operands are given by *x* and *y*, then the result is defined to be $x - y \times n$, where *n* is the integer nearest the exact value of $x \div y$ (if two integers are equally near then the even one is chosen). If the result is equal to 0 then its sign will be the sign of *x*. (See IEEE §5.1.)

Examples:

```
remainder–near('2.1', '3')    ==>  '-0.9'
remainder–near('10', '3')     ==>  '1'
remainder–near('-10', '3')    ==>  '-1'
remainder–near('10.2', '1')   ==>  '0.2'
remainder–near('10', '0.3')   ==>  '0.1'
remainder–near('3.6', '1.3')  ==>  '-0.3'
```

Notes:

1. The **remainder–near** operation differs from the **remainder** operation in that it does not give the same results for numbers whose values are equal to integers as would the usual remainder operator on integers. For example, the operation `remainder('10', '6')` gives the result `'4'`, and `remainder('10.0', '6')` gives `'4.0'` (as would `remainder('10', '6.0')` or `remainder('10.0', '6.0')`). However, `remainder–near('10', '6')` gives the result `'-2'` because its integer division step chooses the closest integer, not the one nearer zero.

2. The result of this operation is always exact.[14]

3. This operation is sometimes known as “IEEE remainder”.

round-to-integer

round–to–integer takes one operand. If the operand is infinite, zero, or has an *exponent* which is zero or positive, then the result is the same as the operand.[15] Otherwise (the exponent is negative), the result is the operand, rounded to the nearest integer using the *rounding* algorithm. This will raise the Inexact condition unless the decimal part was 0. After the rounding the *exponent* will be 0.

[14] There is an open question, here. The exact result would appear to require the possibility of subnormal numbers (see IEEE 854 §2), but subnormal numbers are in conflict with the requirement to have a balanced exponent range for numbers (see IEEE 854 §3.1).

[15] Unless the operand caused an exceptional condition, as usual.

Examples:

```
round–to–integer('2.1')    ==>  '2'
round–to–integer('100')    ==>  '100'
round–to–integer('100.0')  ==>  '100'
round–to–integer('100.5')  ==>  '100'
round–to–integer('10E+5')  ==>  '10E+5'
```

Note: IEEE 854 refers to §4 for this operation, but then implies that *round–half–even* rounding should always be used (whereas §4 specifically allows directed rounding). It is assumed that it was not intended to exclude directed rounding.

Exceptional conditions

This section lists, in the abstract, the exceptional conditions that can arise during the operations defined in this specification. This list is a superset of the list in the base specification (that is, the exceptions in the base specification are repeated here, with the addition of the **Invalid operation** and **Inexact** conditions).

For each condition, the corresponding *signal* in the *context* (see page 6) is given, along with the defined result if the signal is not trapped. Unless stated otherwise, this is also the defined result if the signal is trapped. The value of the trap-enabler for each signal in the context determines whether an operation is completed after the condition is detected or whether the condition is trapped and hence not immediately completed (see IEEE 854 §8).

The following exceptional conditions can occur:

Invalid operation

This occurs and signals *invalid–operation* if:

- *special–values* is 0, and any operand to an arithmetic operation is infinite or a NaN
- an operand to an operation is `[0,sNaN]` (signalling NaN)
- an attempt is made to add `[0,inf]` to `[1,inf]` during an addition or subtraction operation
- an attempt is made to multiply 0 by `[0,inf]` or `[1,inf]`
- an attempt is made to divide either `[0,inf]` or `[1,inf]` by either `[0,inf]` or `[1,inf]`
- the dividend for the **divide–integer** operation or a remainder operation is either `[0,inf]` or `[1,inf]`
- the operand of the **square–root** operation has a *sign* of 1 and a non-zero *integer*
- *special–values* is 1, and any operand to an arithmetic operation has an *integer* whose length is greater than *precision* and the *lost–digits* condition does not occur.

The result of the operation after any of these invalid operations is `[0,qNaN]`.

Division by zero

This occurs and signals *division–by–zero* if division by zero was attempted, and the dividend was not zero.

The result of the operation is `[`*sign*`,inf]`, where *sign* is the sign of the dividend.

Division undefined

This occurs and signals *invalid–operation* if division by zero was attempted, and the dividend is also zero. The result is `[0,qNaN]`.

Division impossible

This occurs and signals *invalid–operation* if the integer result of a **divide–integer** or **remainder** operation had too many digits (would be longer than *precision*), or if the divisor for either of these operations is 0. The result is `[0,qNaN]`.

Overflow

This occurs and signals *overflow* if the *exponent* of a result (from an operation that is not an attempt to divide by zero) would be greater than the largest value that can be handled by the implementation (the value E_{max} as defined in the base specification).

The result depends on the rounding mode:

- For *round–half–up* and *round–half–even*, the result of the operation is `[`*sign*`,inf]`, where *sign* is the sign of the intermediate result.
- For *round–down*, the result is the largest finite number that can be represented, using double precision[16] if designated, or single precision otherwise, with the sign of the intermediate result.
- For *round–ceiling*, the result is the same as for *round–down* if the sign of the intermediate result is 1, or is `[0,inf]` otherwise.
- For *round–floor*, the result is the same as for *round–down* if the sign of the intermediate result is 0, or is `[1,inf]` otherwise.

The result for a trapped overflow is different, and depends on whether the overflow was the result of a conversion or an arithmetic operation *(to be added, see IEEE 854 §7.3)*.

Underflow

This occurs and signals *underflow* if the *exponent* of a result (from an operation that is not an attempt to divide by zero) would be smaller (more negative) than the smallest value that can be handled by the implementation (the value E_{min} as defined in the base specification).

The results for underflows, whether trapped or not, are handled in the same way as for the corresponding overflow, with zero being used instead of infinity.[17]

---

[16] A designated precision is specified here, as in an arbitrary-precision implementation the largest finite number may be ill-defined.

[17] Note that underflows can never result in a subnormal number (see IEEE 854 §2) because the abstract representation cannot represent these; subnormal numbers would be numbers with an *exponent* less than E_{min}.

Lost digits

This occurs and signals *lost–digits* if an operand to an arithmetic operation has more leading significant digits than the *precision* in the context. The result (the value used for the operand, in this case) is the operand rounded to *precision* digits.

Conversion syntax

This occurs and signals *invalid–operation* if an string is being converted to a number and it does not conform to the appropriate numeric string syntax (see page 8). The result is `[0,qNaN]`.

Conversion overflow

This occurs and signals *overflow* if an string is being converted to a number and the value of the *integer* or *exponent* resulting from the conversion is too large for an implementation to handle (perhaps because the concrete representation has size limits).

If the *exponent* is too large, the result is as defined for the **Overflow** condition. Otherwise (the *integer* is too large), the result is the number rounded to double precision (if designated) or to single precision otherwise.

Conversion underflow

This occurs and signals *underflow* if an string is being converted to a number and the value of the *exponent* resulting from the conversion is too small for an implementation to handle (perhaps because the concrete representation has size limits).

The result is as defined for the **Underflow** condition.

Inexact

This occurs and signals *inexact* whenever the result of an operation is not exact (that is, it needed to be rounded and any discarded digits were non-zero), or if an overflow or underflow condition occurs and is not trapped. The result in all cases is unchanged.

The *inexact* signal may be tested (or trapped) to determine if a given operation (or sequence of operations) was inexact.[18]

Insufficient storage

For many implementations, storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered an operating environment error, which can be either be handled as appropriate for the environment, or treated as an **Invalid operation** condition.

The Lost digits and Inexact conditions can coincide with each other or with other conditions. In these cases then any trap enabled for another condition takes precedence over (is handled before) both, and any Lost digits trap takes precedence over Inexact.

It is recommended that implementations distinguish the different conditions listed above, and also provide additional information about exceptional conditions where possible (for example, the operation being attempted and the values of the operand or operands involved – see also IEEE 854 §8.1).

[18] Note that IEEE 854 is inconsistent in its treatment of Inexact in that it states in §7 that the Inexact exception can coincide with Underflow, but does not allow the possibility of Underflow signalling Inexact in §7.5. It is assumed that the latter is an accidental omission.

Appendix A – Changes

This appendix documents changes since the first public draft of this specification (0.30, 9 Aug 2000). It is not part of the specification.

(None yet.)

Index