The decNumber C library

Version 3.36 – 6th July 2006

Mike Cowlishaw

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

Table of Contents

Overview

The decNumber library implements the **General Decimal Arithmetic Specification**[1] in ANSI C. This specification defines a decimal arithmetic which meets the requirements of commercial, financial, and human-oriented applications.

The library fully implements the specification, and hence supports integer, fixed-point, and floating-point decimal numbers directly, including infinite, NaN (Not a Number), and subnormal values.

The code is optimized and tunable for common values (tens of digits) but can be used without alteration for up to a billion digits of precision and 9-digit exponents. It also provides functions for conversions between concrete representations of decimal numbers, including Packed Decimal (4-bit Binary Coded Decimal) and three compressed formats of decimal floating-point (4-, 8-, and 16-byte).

Library modules

The library comprises several modules (corresponding to classes in an object-oriented implementation). Each module has a *header* file (for example, `decNumber.h`) which defines its data structure, and a *source* file of the same name (*e.g.*, `decNumber.c`) which implements the operations on that data structure. These correspond to the instance variables and methods of an object-oriented design.

The core of the library is the *decNumber* module. This uses a decimal number representation designed for efficient computation in software and implements the arithmetic operations, together with some conversions and utilities. Once a number is held as a decNumber, no further conversions are necessary to carry out arithmetic.

Most functions in the decNumber module take as an argument a *decContext* structure, which provides the context for operations (precision, rounding mode, *etc.*) and also controls the handling of exceptional conditions (corresponding to the flags and trap enablers in a hardware floating-point implementation).

The decNumber representation is machine-dependent (for example, it contains integers which may be big-endian or little-endian), and is optimized for speed rather than storage efficiency.

1 See `http://www2.hursley.ibm.com/decimal` for details.

Four machine-independent (but optionally endian-dependent) compact storage formats are provided for interchange. These are:

decimal32 This is a 32-bit decimal floating-point representation, which provides 7 decimal digits of precision in a compressed format.[2]

decimal64 This is a 64-bit decimal floating-point representation, which provides 16 decimal digits of precision in a compressed format.

decimal128 This is a 128-bit decimal floating-point representation, which provides 34 decimal digits of precision in a compressed format.

decPacked The decPacked format is the classic packed decimal format implemented by IBM S/360 and later machines, where each digit is encoded as a 4-bit binary sequence (BCD) and a number is ended by a 4-bit sign indicator. The decPacked module accepts variable lengths, allowing for very large numbers (up to a billion digits), and also allows the specification of a *scale*.

The module for each format provides conversions to and from the core decNumber format. The decimal32, decimal64, and decimal128 modules also provide conversions to and from character string format (using the functions in the decNumber module).

Standards compliance

It is intended that the decNumber implementation complies with:

- the floating-point decimal arithmetic defined in ANSI X3.274-1996[3] (including errata through 2001)
- all requirements of IEEE 854-1987,[4] as modified by the current IEEE 754r revision work,[5] except that:

  1. The values returned after overflow and underflow do not change when an exception is trapped. This is because the IEEE 854 definition does not generalize to the power and exp operations. Similarly, the criteria for underflow do not depend on the setting of the underflow trap-enabler (the subnormal condition may be tested or trapped, instead).
  2. The IEEE remainder operator (decNumberRemainderNear) is restricted to those values where the intermediate integer can be represented in the current precision, because the conventional implementation of this operator would be very long-running for the range of numbers supported (up to $\pm 10^{1{,}000{,}000{,}000}$).

  Note that all other requirements of IEEE 854 (such as subnormal numbers and –0) are supported.

Please advise the author of any discrepancies with these standards.

---

2 See http://www2.hursley.ibm.com/decimal/decbits.html for details of the compressed formats.

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

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

5 See http://grouper.ieee.org/groups/754/

User’s Guide

To use the decNumber library efficiently it is best to first convert the numbers you are working with from their coded representation into decNumber format, then carry out calculations on them, and finally convert them back into the desired coded format.

Conversions to and from the decNumber format are fast; they are usually faster than even the simplest calculations (x=x+1, for example). Therefore, in general, the cost of conversions is small compared to that of calculation.

The coded formats currently provided for in the library are

- strings (ASCII bytes, terminated by `'\0'`, as usual for C)
- three formats of compressed floating-point decimals
- Packed Decimal numbers with optional scale.

The remainder of this section illustrates the use of these coded formats in conjunction with the core decContext and decNumber modules by means of examples.

Notes on running the examples

1. All the examples are written conforming to ANSI C, except that they use “line comment” notation (comments starting with `//`) from BCPL and C++ for more concise commentary. Most C compilers support this; if not, a short script can be used to convert the line comments to traditional block comments (`/* ... */`).

2. The header files and Example 6 use the standard integer types from `stdint.h` described in the ANSI C99 standard (ISO/IEC 9899:1999). If your C compiler does not supply `stdint.h`, the following will suffice:

```
/* stdint.h -- some standard integer types from C99 */
typedef unsigned char  uint8_t;
typedef          char   int8_t;
typedef unsigned short uint16_t;
typedef          short  int16_t;
typedef unsigned int   uint32_t;
typedef          int    int32_t;
typedef unsigned long long uint64_t;
typedef          long long int64_t;
```

   You may need to change these if (for example) the `int` type in your compiler does not describe a 32-bit integer. If there are no 64-bit integers available with your compiler, set the `DECUSE64` tuning parameter (see page 42) to 0; the last two `typedef`s above are then not needed.

3. One aspect of the examples is implementation-defined. It is assumed that the default handling of the SIGFPE signal is to end the program. If your implementation ignores this signal, the lines with `set.traps=0;` would not be needed in the simpler examples.

Example 1 – simple addition

This example is a simple test program which can easily be extended to demonstrate more complicated operations or to experiment with the functions available.

```
1.  // example1.c -- convert the first two argument words to decNumber,
2.  // add them together, and display the result
3.
4.  #define  DECNUMDIGITS 34           // work with up to 34 digits
5.  #include "decNumber.h"             // base number library
6.  #include <stdio.h>                 // for printf
7.
8.  int main(int argc, char *argv[]) {
9.    decNumber a, b;                  // working numbers
10.   decContext set;                  // working context
11.   char string[DECNUMDIGITS+14];    // conversion buffer
12.
13.   if (argc<3) {                    // not enough words
14.     printf("Please supply two numbers to add.\n");
15.     return 1;
16.     }
17.   decContextDefault(&set, DEC_INIT_BASE); // initialize
18.   set.traps=0;                     // no traps, thank you
19.   set.digits=DECNUMDIGITS;         // set precision
20.
21.   decNumberFromString(&a, argv[1], &set);
22.   decNumberFromString(&b, argv[2], &set);
23.   decNumberAdd(&a, &a, &b, &set);            // a=a+b
24.   decNumberToString(&a, string);
25.   printf("%s + %s => %s\n", argv[1], argv[2], string);
26.   return 0;
27.   } // main
```

This example is a complete, runnable program. In later examples we'll leave out some of the "boilerplate", checking, *etc.*, but this one should compile and be usable as it stands.

Lines 1 and 2 document the purpose of the program.

Line 4 sets the maximum precision of decNumbers to be used by the program, which is used by the embedded header file in line 5 (and also elsewhere in this program).

Line 6 includes the C library for input and output, so we can use the `printf` function. Lines 8 through 11 start the `main` function, and declare the variables we will use. Lines 13 through 16 check that enough argument words have been given to the program.

Lines 17–19 initialize the decContext structure, turn off error signals, and set the working precision to the maximum possible for the size of decNumbers we have declared.

Lines 21 and 22 convert the first two argument words into numbers; these are then added together in line 23, converted back to a string in line 24, and displayed in line 25.

Note that there is no error checking of the arguments in this example, so the result will be `NaN` (Not a Number) if one or both words is not a number. Error checking is introduced in Example 3 (see page 7).

Example 2 – compound interest

This example takes three parameters (initial amount, interest rate, and number of years) and calculates the final accumulated investment. For example:

```
100000 at 6.5% for 20 years => 352364.51
```

The heart of the program is:

```
1.  decNumber one, mtwo, hundred;                   // constants
2.  decNumber start, rate, years;                   // parameters
3.  decNumber total;                                // result
4.  decContext set;                                 // working context
5.  char string[DECNUMDIGITS+14];                   // conversion buffer
6.
7.  decContextDefault(&set, DEC_INIT_BASE);         // initialize
8.  set.traps=0;                                    // no traps
9.  set.digits=25;                                  // precision 25
10. decNumberFromString(&one,       "1", &set);     // set constants
11. decNumberFromString(&mtwo,     "-2", &set);
12. decNumberFromString(&hundred, "100", &set);
13.
14. decNumberFromString(&start, argv[1], &set);     // parameter words
15. decNumberFromString(&rate,  argv[2], &set);
16. decNumberFromString(&years, argv[3], &set);
17.
18. decNumberDivide(&rate, &rate, &hundred, &set);  // rate=rate/100
19. decNumberAdd(&rate, &rate, &one, &set);         // rate=rate+1
20. decNumberPower(&rate, &rate, &years, &set);     // rate=rate**years
21. decNumberMultiply(&total, &rate, &start, &set); // total=rate*start
22. decNumberRescale(&total, &total, &mtwo, &set);  // two digits please
23.
24. decNumberToString(&total, string);
25. printf("%s at %s%% for %s years => %s\n",
26.        argv[1], argv[2], argv[3], string);
27. return 0;
```

These lines would replace the content of the `main` function in Example 1 (adding the check for the number of parameters would be advisable).

As in Example 1, the variables to be used are first declared and initialized (lines 1 through 12), with the working precision being set to 25 in this case. The parameter words are converted into decNumbers in lines 14–16.

The next four function calls calculate the result; first the rate is changed from a percentage (*e.g.*, 6.5) to a per annum rate (1.065). This is then raised to the power of the number of years (which must be a whole number), giving the rate over the total period. This rate is then multiplied by the initial investment to give the result.

Next (line 22) the result is rescaled so it will have only two digits after the decimal point (an exponent of –2), and finally (lines 24–26) it is converted to a string and displayed.

Example 3 – passive error handling

Neither of the previous examples provide any protection against invalid numbers being passed to the programs, or against calculation errors such as overflow. If errors occur, therefore, the final result will probably be `NaN` or infinite (decNumber result structures are always valid after an operation, but their value may not be useful).

One way to check for errors would be to check the *status* field of the decContext structure after every decNumber function call. However, as that field accumulates errors until cleared deliberately it is often more convenient and more efficient to delay the check until after a sequence is complete.

This passive checking is easily added to Example 2. Replace lines 14 through 22 in that example with (the original lines repeated here are unchanged):

```
1.  decNumberFromString(&start, argv[1], &set);     // parameter words
2.  decNumberFromString(&rate,  argv[2], &set);
3.  decNumberFromString(&years, argv[3], &set);
4.  if (set.status) {
5.    printf("An input argument word was invalid [%s]\n",
6.           decContextStatusToString(&set));
7.    return 1;
8.    }
9.  decNumberDivide(&rate, &rate, &hundred, &set);  // rate=rate/100
10. decNumberAdd(&rate, &rate, &one, &set);         // rate=rate+1
11. decNumberPower(&rate, &rate, &years, &set);     // rate=rate**years
12. decNumberMultiply(&total, &rate, &start, &set); // total=rate*start
13. decNumberRescale(&total, &total, &mtwo, &set);  // two digits please
14. if (set.status & DEC_Errors) {
15.   set.status &= DEC_Errors;                     // keep only errors
16.   printf("Result could not be calculated [%s]\n",
17.          decContextStatusToString(&set));
18.   return 1;
19.   }
```

Here, in the `if` statement starting on line 4, the error message is displayed if the *status* field of the `set` structure is non-zero. The call to `decContextStatusToString` in line 6 returns a string which describes a set status bit (probably "`Conversion syntax`").

In line 14, the test is augmented by anding the `set.status` value with `DEC_Errors`. This ensures that only serious conditions trigger the message. In this case, it is possible that the `DEC_Inexact` and `DEC_Rounded` conditions will be set (if an overflow occurred) so these are cleared in line 15.

With these changes, messages are displayed and the `main` function ended if either a bad input parameter word was found (for example, try passing a non-numeric word) or if the calculation could not be completed (*e.g.*, try a value for the third argument which is not an integer).[6]

[6] Of course, in a user-friendly application, more detailed and specific error messages are appropriate. But here we are demonstrating error handling, not user interfaces.

Example 4 – active error handling

The last example handled errors passively, by testing the context *status* field directly. In this example, the C signal mechanism is used to handle traps which are raised when errors occur.

When one of the decNumber functions sets a bit in the context *status*, the bit is compared with the corresponding bit in the *traps* field. If that bit is set (is 1) then a C Floating-Point Exception signal (`SIGFPE`) is raised. At that point, a signal handler function (previously identified to the C runtime) is called.

The signal handler function can either simply log or report the trap and then return (and execution will continue as though the trap had not occurred) or – as in this example – it can call the C `longjmp` function to jump to a previously preserved point of execution.

Note that if a jump is used, control will not return to the code which called the decNumber function that raised the trap, and so care must be taken to ensure that any resources in use (such as allocated memory) are cleaned up appropriately.

To create this example, modify the Example 1 code this time, by first removing line 18 (`set.traps=0;`). This will leave the *traps* field with its default setting, which has all the `DEC_Errors` bits set, hence enabling traps for any of those conditions. Then insert after line 6 (before the `main` function):

```
1. #include <signal.h>                 // signal handling
2. #include <setjmp.h>                 // setjmp/longjmp
3.
4. jmp_buf preserve;                   // stack snapshot
5.
6. void signalHandler(int sig) {
7.   signal(SIGFPE, signalHandler);    // re-enable
8.   longjmp(preserve, sig);           // branch to preserved point
9.   }
```

Here, lines 1 and 2 include definitions for the C library functions we will use. Line 4 declares a global buffer (accessible to both the main function and the signal handler) which is used to preserve the point of execution to which we will jump after handling the signal.

Lines 6 through 9 are the signal handler. Line 7 re-enables the signal handler, as described below (in this example this is in fact unnecessary as we will be ending the program immediately). This is normally needed as handlers are disabled on entry, and need to be re-enabled if more than one trap is to be handled.

Line 8 jumps to the point preserved when the program starts up (in the next code insert). The value, `sig`, which the signal handler receives is passed to the preserved code. In this example, `sig` always has the value `SIGFPE`, but in a more complicated program the same signal handler could be used to handle other signals, too.

The next segment of code is inserted after line 11 of Example 1 (just after the existing declarations):

```
1. int value;                          // work variable
2.
3. signal(SIGFPE, signalHandler);     // set up signal handler
4. value=setjmp(preserve);            // preserve and test environment
5. if (value) {                       // (non-0 after longjmp)
6.   set.status &= DEC_Errors;        // keep only errors
7.   printf("Signal trapped [%s].\n", decContextStatusToString(&set));
8.   return 2;
9.   }
```

Here, a work variable is declared in line 1 and the signal handler function is registered (identified to the C run time) in line 3. The call to the `signal` function identifies the signal to be handled (`SIGFPE`) and the function (`signalHandler`) that will be called when the signal is raised, and enables the handler.

Next, in line 4, the `setjmp` function is called. On its first call, this saves the current point of execution into the `preserve` variable and then returns 0. The following lines (5–8) are then not executed and execution of the `main` function continues as before.

If a trap later occurs (for example, if one of the arguments is not a number) then the following takes place:

1. the `SIGFPE` signal is raised by the decNumber library
2. the `signalHandler` function is called by the C run time with argument `SIGFPE`
3. the function re-enables the signal, and then calls `longjmp`
4. this in turn causes the execution stack to be “unwound” to the point which was preserved in the initial call to `setjmp`
5. the `setjmp` function then returns, with the (non-0) value passed to it in the call to `longjmp`
6. the test in line 5 then succeeds, so line 6 clears any informational status bits in the *status* field in the context structure which was given to the decNumber routines and line 7 displays a message, using the same structure
7. finally, in line 8, the `main` function is ended by the `return` statement.

Of course, different behaviors are possible both in the signal handler, as already noted, and after the jump; the main program could prompt for new values for the input parameters and then continue as before, for example.

Example 5 – compressed formats

The previous examples all used decNumber structures directly, but that format is not necessarily compact and is machine-dependent. These attributes are generally good for performance, but are less suitable for the storage and exchange of numbers.

The decimal32, decimal64, and decimal128 forms are provided as efficient, machine-independent formats used for storing numbers of up to 7, 16 or 34 decimal digits respectively, in 4, 8, or 16 bytes. These formats are similar to, and are used in the same manner as, the C `float` and `double` data types.

Here's an example program. Like Example 1, this is runnable as it stands, although it's recommended that at least the argument count check be added.

```
1.  // example5.c -- decimal64 conversions
2.  #include "decimal64.h"             // decimal64 and decNumber library
3.  #include <stdio.h>                 // for (s)printf
4.
5.  int main(int argc, char *argv[]) {
6.    decimal64 a;                     // working decimal64 number
7.    decNumber d;                     // working number
8.    decContext set;                  // working context
9.    char string[DECIMAL64_String];   // number->string buffer
10.   char hexes[25];                  // decimal64->hex buffer
11.   int i;                           // counter
12.
13.   decContextDefault(&set, DEC_INIT_DECIMAL64); // initialize
14.
15.   decimal64FromString(&a, argv[1], &set);
16.   // lay out the decimal64 as eight hexadecimal pairs
17.   for (i=0; i<8; i++) {
18.     sprintf(&hexes[i*3], "%02x ", a.bytes[i]);
19.     }
20.   decimal64ToNumber(&a, &d);
21.   decNumberToString(&d, string);
22.   printf("%s => %s=> %s\n", argv[1], hexes, string);
23.   return 0;
24.   } // main
```

Here, the `#include` on line 2 not only defines the decimal64 type, but also includes the decNumber and decContext header files. Also, if `DECNUMDIGITS` (see page 21) has not already been defined, the `decimal64.h` file sets it to 16 so that any decNumbers declared will be exactly the right size to take any decimal64 without rounding.

The declarations in lines 6–11 create three working structures and other work variables; the decContext structure is initialized in line 13 (here, `set.traps` is 0).

Line 15 converts the input argument word to a decimal64 (with a function call very similar to decNumberFromString). Note that the value would be rounded if the number needed more than 16 digits of precision.

Lines 16–19 lay out the decimal64 as eight hexadecimal pairs in a string, so that its encoding can be displayed.

Lines 20–22 show how decimal64 numbers are used. First the decimal64 is converted to a decNumber, then arithmetic could be carried out, and finally the decNumber is converted back to some standard form (in this case a string, so it can be displayed in line 22). For example, if the input argument were "`79`", the following would be displayed:

```
79 => 22 38 00 00 00 00 00 79 => 79
```

The decimal32 and decimal128 forms are used in exactly the same way, for working with up to 7 or up to 34 digits of precision respectively. These forms have the same constants and functions as decimal64 (with the obvious name changes).

Like `decimal64.h`, the decimal32 and decimal128 header files define the `DECNUMDIGITS` constant (see page 21) to either 7 or 34 if it has not already been defined.

Example 6 – Packed Decimal numbers

This example reworks Example 2, starting and ending with Packed Decimal numbers. First, lines 4 and 5 of Example 1 (which Example 2 modifies) are replaced by the line:

```
#include "decPacked.h"
```

Then the following declarations are added to the `main` function:

```
1.  uint8_t startpack[]={0x01, 0x00, 0x00, 0x0C};   // investment=100000
2.  int32_t startscale=0;
3.  uint8_t ratepack[]={0x06, 0x5C};                // rate=6.5%
4.  int32_t ratescale=1;
5.  uint8_t yearspack[]={0x02, 0x0C};               // years=20
6.  int32_t yearsscale=0;
7.  uint8_t respack[16];                            // result, packed
8.  int32_t resscale;                               // ..
9.  char    hexes[49];                              // for packed->hex
10. int     i;                                      // counter
```

The first three pairs declare and initialize the three parameters, with a Packed Decimal byte array and associated scale for each. In practice these might be read from a file or database. The fourth pair is used to receive the result. The last two declarations (lines 9 and 10) are work variables used for displaying the result.

Next, in Example 2, line 5 is removed, and lines 14 through 26 are replaced by:

```
1.  decPackedToNumber(startpack, sizeof(startpack), &startscale, &start);
2.  decPackedToNumber(ratepack,  sizeof(ratepack),  &ratescale,  &rate);
3.  decPackedToNumber(yearspack, sizeof(yearspack), &yearsscale, &years);
4.
5.  decNumberDivide(&rate, &rate, &hundred, &set);  // rate=rate/100
6.  decNumberAdd(&rate, &rate, &one, &set);         // rate=rate+1
7.  decNumberPower(&rate, &rate, &years, &set);     // rate=rate**years
8.  decNumberMultiply(&total, &rate, &start, &set); // total=rate*start
9.  decNumberRescale(&total, &total, &mtwo, &set);  // two digits please
10.
11. decPackedFromNumber(respack, sizeof(respack), &resscale, &total);
12.
13. // lay out the total as sixteen hexadecimal pairs
14. for (i=0; i<16; i++) {
15.   sprintf(&hexes[i*3], "%02x ", respack[i]);
16.   }
17. printf("Result: %s (scale=%d)\n", hexes, resscale);
```

Here, lines 1 through 3 convert the Packed Decimal parameters into decNumber structures. Lines 5-9 calculate and rescale the total, as before, and line 11 converts the final decNumber into Packed Decimal and scale. Finally, lines 13-17 lay out and display the result, which should be:

```
Result: 00 00 00 00 00 00 00 00 00 00 00 03 52 36 45 1c  (scale=2)
```

Note that the number is right-aligned, with a sign nibble.

Module descriptions

The section contains a detailed description of each of the modules in the library. Each description is in three parts:

1. An overview of the module and a description of its primary data structure.
2. A description of other definitions in the header (**.h**) file. This summarizes the content of the header file rather than detailing every constant as it is assumed that users will have a copy of the header file available.
3. A description of the functions in the source (**.c**) file. This is a detailed description of each function and how to use it, the intent being that it should not be necessary to have the source file available in order to use the functions.

The modules all conform to some general rules:

- They are reentrant (they have no static variables and may safely be used in multi-threaded applications).
- All data structures are passed by reference, for best performance. Data structures whose references are passed as inputs are never altered unless they are also used as a result. Where appropriate, functions return a reference to a result argument.
- Up to some maximum (chosen by a tuning parameter in the `decNumberLocal.h` file), calculations do not require additional allocated memory, except for rounded input arguments. Whenever memory is allocated, it is always released before the function returns or raises any traps. The latter constraint implies that long jumps may safely be made from a signal handler handling any traps, for example.
- The names of all modules start with the string “`dec`”.
- The names of all public constants start with the string “`DEC`”.
- Public functions (and macros used as functions) in a module have names which start with the name of the module (for example, `decNumberAdd`). This naming scheme corresponds to the common naming scheme in object-oriented languages, where that function (method) might be called `decNumber.add`.
- The types `int` and `long` are not used; instead types defined in the C99 `stdint.h` header file are used to ensure integers are of the correct length.
- Strings always follow C conventions. That is, they are always terminated by a null character (`'\0'`).

decContext module

The decContext module defines the data structure used for providing the context for operations and for managing exceptional conditions.

The decContext structure comprises the following fields:

digits The *digits* field is used to set the precision to be used for an operation. The result of an operation will be rounded to this length if necessary, and hence the space needed for the result decNumber structure is limited by this field.

digits is of type `int32_t`, and must have a value in the range 1 through 999,999,999.

emax The *emax* field is used to set the magnitude of the largest *adjusted exponent* that is permitted. The adjusted exponent is calculated as though the number were expressed in scientific notation (that is, except for 0, expressed with one non-zero digit before the decimal point).

If the adjusted exponent for a result or conversion would be larger than *emax* then an overflow results.

emax is of type `int32_t`, and must have a value in the range 0 through 999,999,999.

emin The *emin* field is used to set the smallest *adjusted exponent* that is permitted for normal numbers. The adjusted exponent is calculated as though the number were expressed in scientific notation (that is, except for 0, expressed with one non-zero digit before the decimal point).

If the adjusted exponent for a result or conversion would be smaller than –*emin* then the result is subnormal. If the result is also inexact, an underflow results. The exponent of the smallest possible number (closest to zero) will be *emin*–*digits*+1.[7]

emin will usually equal –*emax*, but when a compressed format is used it will be –(*emax*–1).

emin is of type `int32_t`, and must have a value in the range –999,999,999 through 0.

round The *round* field is used to select the rounding algorithm to be used if rounding is necessary during an operation. It must be one of the values in the `rounding` enumeration:

`DEC_ROUND_CEILING`	Round towards +`Infinity`.
`DEC_ROUND_DOWN`	Round towards 0 (truncation).
`DEC_ROUND_FLOOR`	Round towards –`Infinity`.
`DEC_ROUND_HALF_DOWN`	Round to nearest; if equidistant, round down.

[7] See `http://www2.hursley.ibm.com/decimal/decarith.html` for details.

`DEC_ROUND_HALF_EVEN`	Round to nearest; if equidistant, round so that the final digit is even.
`DEC_ROUND_HALF_UP`	Round to nearest; if equidistant, round up.
`DEC_ROUND_UP`	Round away from 0.

status The *status* field comprises one bit for each of the exceptional conditions described in the specifications (for example, Division by zero is indicated by the bit defined as `DEC_Division_by_zero`). Once set, a bit remains set until cleared by the user, so more than one condition can be recorded.

status is of type `uint32_t` (unsigned integer). Bits in the field must only be set if they are defined in the decContext header file. In use, bits are set by the decNumber library modules when exceptional conditions occur, but are never reset. The library user should clear the bits when appropriate (for example, after handling the exceptional condition), but should never set them.

traps The *traps* field is used to indicate which of the exceptional conditions should cause a *trap*. That is, if an exceptional condition bit is set in the *traps* field, then a trap event occurs when the corresponding bit in the *status* field is set.

In this implementation, a trap is indicated by raising the signal `SIGFPE` (defined in `signal.h`), the Floating-Point Exception signal.

Applications may ignore traps, or may use them to recover from failed operations. Alternatively, applications can prevent all traps by clearing the *traps* field, and inspect the *status* field directly to determine if errors have occurred.

traps is of type `uint32_t`. Bits in the field must only be set if they are defined in the decContext header file.

Note that the result of an operation is always a valid number, but after an exceptional condition has been detected its value may be one of the *special values* (NaN or infinite). These values can then propagate through other operations without further conditions being raised.

clamp The *clamp* field adds explicit exponent clamping, as is applied when a result is encoded in one of the compressed formats. When 0, a result exponent is limited to *emax* (for example, the exponent of a zero result will be clamped to this value). When 1, a result exponent is limited to *emax*–(*digits*–1). As well as clamping zeros, this may cause the coefficient of a result to be padded with zeros on the right in order to bring the exponent within range.

For example, if *emax* is +96 and *digits* is 7, the result 1.23E+96 would have a [*sign*, *coefficient*, *exponent*] of `[0, 123, 94]` if *clamp* were 0, but would give `[0, 1230000, 90]` if *clamp* were 1.

clamp is of type `uint8_t` (an unsigned byte).

extended The *extended* field controls the level of arithmetic supported. When 1, special values are possible, some extra checking required for IEEE 854 conformance is enabled, and subnormal numbers can result from operations (that is, results whose adjusted exponent is as low as *emin*–(*digits*–1) are possible). When 0, the X3.274 subset is supported; in particular, –0 is not possible, operands are rounded, and the exponent range is balanced.

If *extended* will always be 1, then the `DECSUBSET` tuning parameter may be set to 0 in `decContext.h`. This will remove the *extended* field from the structure, and also remove all code that refers to it. This gives a 10%–20% speed improvement for many operations.

extended is of type `uint8_t` (an unsigned byte).

Please see the arithmetic specification for further details on the meaning of specific settings (for example, the rounding mode).

Definitions

The `decContext.h` header file defines the context used by most functions in the decNumber module; it is therefore automatically included by `decNumber.h`. In addition to defining the decContext data structure described above, it also includes:

- The enumeration of the rounding modes supported by this implementation (for the *round* field of the decContext).
- The exceptional condition flags, used in the *status* and *traps* fields.
- Constants describing the range of precision and adjusted exponent supported by the decNumber package.
- Groupings for the exceptional conditions flags, indicating how they correspond to the named conditions defined in IEEE 854, which are usually considered errors (`DEC_Errors`), *etc.*
- A character constant naming each of the exceptional conditions (intended for human-readable error reporting).
- Constants used for selecting initialization schemes.
- Definitions of the public functions in the decContext module.

Several of the exceptional condition flags merit special attention:

- The `DEC_Clamped` flag is set whenever the exponent of a result is clamped to a maximum or minimum value, derived from *emax* or *emin* and possibly modified by *clamp*.
- The `DEC_Inexact` flag is set whenever a result is inexact (non-zero digits were discarded) due to rounding of input operands or the result.
- The `DEC_Lost_digits` flag is set when an input operand is made inexact through rounding (which can only occur if extended is 0).
- The `DEC_Rounded` flag is set whenever a result or input operand is rounded (even if only zero digits were discarded).
- The `DEC_Subnormal` flag is set whenever a result is a subnormal value.

Unlike the other status flags, which indicate error conditions, execution continues normally when these events occur and the result is a number (unless an error condition also occurs). As usual, any or all of the conditions can be enabled for traps and in this case the operation is completed before the trap takes place.

Functions

The `decContext.c` source file contains the public functions defined in the header file, as follows.

decContextDefault(context, kind)

This function is used to initialize a decContext structure to default values. It is stongly recommended that this function always be used to initialize a decContext structure, even if most or all of the fields are to be set explicitly (in case new fields are added to a later version of the structure).

The arguments are:

context (`decContext *`) Pointer to the structure to be initialized.

kind (`int32_t`) The kind of initialization to be performed. Only the values defined in the decContext header file are permitted (any other value will initialize the structure to a valid condition, but with the `DEC_Invalid_operation` status bit set).

When *kind* is `DEC_INIT_BASE`, the defaults for the ANSI X3.274 arithmetic subset are set. That is, the *digits* field is set to 9, the *emax* field is set to 999999999, the *round* field is set to `ROUND_HALF_UP`, the *status* field is cleared (all bits zero), the *traps* field has all the `DEC_Errors` bits set (`DEC_Rounded`, `DEC_Inexact`, `DEC_Lost_digits`, and `DEC_Subnormal` are 0), *clamp* is set to 0, and *extended* (if present) is set to 0.

When *kind* is `DEC_INIT_DECIMAL32`, defaults for a *decimal32* number using IEEE 854 rules are set. That is, the *digits* field is set to 7, the *emax* field is set to 96, the *emin* field is set to –95, the *round* field is set to `DEC_ROUND_HALF_EVEN`, the *status* field is cleared (all bits zero), the *traps* field is cleared (no traps are enabled), *clamp* is set to 1, and *extended* (if present) is set to 1.

When *kind* is `DEC_INIT_DECIMAL64`, defaults for a *decimal64* number using IEEE 854 rules are set. That is, the *digits* field is set to 16, the *emax* field is set to 384, the *emin* field is set to –383, and the other fields are set as for `DEC_INIT_DECIMAL32`.

When *kind* is `DEC_INIT_DECIMAL128`, defaults for a *decimal128* number using IEEE 854 rules are set. That is, the *digits* field is set to 34, the *emax* field is set to 6144, the *emin* field is set to –6143, and the other fields are set as for `DEC_INIT_DECIMAL32`.

Returns *context*.

decContextSetStatus(context, status)

This function is used to set one or more status bits in the *status* field of a decContext. If any of the bits being set have the corresponding bit set in the *traps* field, a trap is raised (regardless of whether the bit is already set in the *status* field). Only one trap is raised even if more than one bit is being set.

The arguments are:

context (decContext *) Pointer to the structure whose status is to be set.

status (uint32_t) Any 1 (set) bit in this argument will cause the corresponding bit to be set in the context *status* field. Only bits defined in the decContext header file should be set; the effect of setting other bits is undefined.[8]

Returns *context*.

Normally, only library modules use this function. Applications may clear status bits but should not set them (except, perhaps, for testing).

Note that a signal handler which handles a trap raised by this function may execute a C long jump, and hence control may not return from the function. It should therefore only be invoked when any state and resources used (such as allocated memory) are clean.

decContextSetStatusFromString(context, string)

This function is used to set a status bit in the *status* field of a decContext, using the name of the bit as returned by the decContextStatusToString function. If the bit being set has the corresponding bit set in the *traps* field, a trap is raised (regardless of whether the bit is already set in the *status* field).

The arguments are:

context (decContext *) Pointer to the structure whose status is to be set.

string (char *) A string which must be exactly equal to one that might be returned by decContextStatusToString. If the string is "No status", the status is not changed and no trap is raised. If the string is "Multiple status", or is not recognized, then the call is in error.

Returns *context* unless the *string* is in error, in which case NULL is returned.

Normally, only library and test modules use this function. Applications may clear status bits but should not set them (except, perhaps, for testing).

Note that a signal handler which handles a trap raised by this function may execute a C long jump, and hence control may not return from the function. It should therefore only be invoked when any state and resources used (such as allocated memory) are clean.

decContextStatusToString(context)

This function returns a pointer (char *) to a human-readable description of a status bit. The string pointed to will be a constant.

The argument is:

context (decContext *) Pointer to the structure whose status is to be returned as a string. The bits set in the *status* field must comprise only bits defined in the header file.

If no bits are set in the *status* field, a pointer to the string "No status" is returned. If more than one bit is set, a pointer to the string "Multiple status" is returned.

---

8 If "private" bits were allowed, future extension of the library with other conditions would be impossible.

Note that the content of the string pointed to is a programming interface (it is understood by the `decContextSetStatusFromString` function) and is therefore not language- or locale-dependent.

decNumber module

The decNumber module defines the data structure used for representing numbers in a form suitable for computation, and provides the functions for operating on those values.

The decNumber structure is optimized for efficient processing of relatively short numbers (tens or hundreds of digits); in particular it allows the use of fixed sized structures and minimizes copy and move operations. The functions in the module, however, support arbitrary precision arithmetic (up to 999,999,999 decimal digits, with exponents up to 9 digits).

The essential parts of a decNumber are the *coefficient*, which is the significand of the number, the *exponent* (which indicates the power of ten by which the *coefficient* should be multiplied), and the *sign*, which is 1 if the number is negative, or 0 otherwise. The numerical *value* of the number is then given by: $(-1)^{sign} \times coefficient \times 10^{exponent}$.

Numbers may also be a *special value*. The special values are *NaN* (Not a Number), which may be *quiet* (propagates quietly through operations) or *signaling* (raises the Invalid operation condition when encountered), and ±*infinity*.

These parts are encoded in the four fields of the decNumber structure:

digits The *digits* field contains the length of the *coefficient*, in decimal digits.

digits is of type `int32_t`, and must have a value in the range 1 through 999,999,999.

exponent The *exponent* field holds the exponent of the number. Its range is limited by the requirement that the range of the *adjusted exponent* of the number be balanced and fit within a whole number of decimal digits (in this implementation, be –999,999,999 through +999,999,999). The adjusted exponent is the exponent that would result if the number were expressed with a single digit before the decimal point, and is therefore given by *exponent*+*digits*–1.

When the *extended* flag in the context is 1, gradual underflow (using *subnormal* values) is enabled. In this case, the lower limit for the adjusted exponent becomes –999,999,999–(*precision*–1), where *precision* is the digits setting from the context; the adjusted exponent may then have 10 digits.

exponent is of type `int32_t`.

bits The *bits* field comprises one bit which indicates the *sign* of the number (1 for negative, 0 otherwise), 3 bits which indicate the special values, and 4 further bits which are unused and reserved. These reserved bits must be zero.

If the number has a special value, just one of the indicator bits (`DECINF`, `DECNAN`, or `DECSNAN`) will be set (along with `DECNEG` iff the sign is 1). If `DECINF` is set *digits* must be 1 and the other fields must be 0. If the number is a NaN, the *exponent* must be zero and the coefficient holds any diagnostic information (with *digits* indicating its length, as for finite numbers). A zero coefficient indicates no diagnostic information.

bits is of type `uint8_t` (an unsigned byte). Masks for the named bits, and some useful macros, are defined in the header file.

lsu The *lsu* field is one or more *units* in length (of type `decNumberUnit`, an unsigned integer), and contains the digits of the *coefficient*. Each unit represents one or more of the digits in the *coefficient* and has a binary value in the range 0 through 10^n-1, where *n* is the number of digits in a unit and is the value set by `DECDPUN` (see page 41). The size of a unit is the smallest of 1, 2, or 4 bytes which will contain the maximum value held in the unit.

The units comprising the *coefficient* start with the least significant unit (lsu). Each unit except the most significant unit (msu) contains `DECDPUN` digits. The msu contains from 1 through `DECDPUN` digits, and must not be 0 unless *digits* is 1 (for the value zero). Leading zeros in the msu are never included in the *digits* count, except for the value zero.

The number of units predefined for the *lsu* field is determined by `DECNUMDIGITS`, which defaults to 1 (the number of units will be `DECNUMDIGITS` divided by `DECDPUN`, rounded up to a whole unit).

For many applications, there will be a known maximum length for numbers and `DECNUMDIGITS` can be set to that length, as in Example 1 (see page 5). In others, the length may vary over a wide range and it then becomes the programmer's responsibility to ensure that there are sufficient units available immediately following the decNumber *lsu* field. This can be achieved by enclosing the decNumber in other structures which append various lengths of unit arrays, or in the more general case by allocating storage with sufficient space for the other decNumber fields and the units of the number.

lsu is an array of type `decNumberUnit` (an unsigned integer whose length depends on the value of `DECDPUN`), with at least one element. If *digits* needs fewer units than the size of the array, remaining units are not used (they will neither be changed nor referenced). For special values, only the first unit need be 0.

It is expected that decNumbers will usually be constructed by conversions from other formats, such as strings or decimal64 structures, so the decNumber structure is in some sense an "internal" representation; in particular, it is machine-dependent.[9]

Examples:

If `DECDPUN` were 4, the value `-1234.50` would be encoded with:

digits = `6`
exponent = `-2`
bits = `0x80`
lsu = {`3450, 12`}

the value `0` would be:

digits = `1`
exponent = `0`
bits = `0x00`
lsu = {`0`}

[9] The layout of an integer might be big-endian or little-endian, for example.

and $-\infty$ (minus infinity) would be:

digits = `1`
exponent = `0`
bits = `0xC0`
lsu = {`0`}

Definitions

The `decNumber.h` header file defines the decNumber data structure described above. It also includes:

- The tuning parameter `DECDPUN`.

  This sets the number of digits held in one *unit* (see page 21), which in turn alters the performance and other characteristics of the library. Further details are given in the tuning section (see page 41).

  If this parameter is changed, the `decNumber.c` source file must be recompiled for the change to have effect.

- Constants naming the bits in the *bits* field, such as `DECNEG`, the sign bit.
- Definitions of the public functions and macros in the decNumber module.

Functions

The `decNumber.c` source file contains the public functions defined in the header file. These comprise conversions to and from strings, the arithmetic operations, and some utility functions.

The functions all follow some general rules:

- Operands to the functions which are decNumber structures (referenced by an argument) are never modified unless they are also specified to be the result structure (which is always permitted).

  Often, operations which do specify an operand and result as the same structure can be carried out in place, giving improved performance. For example, $x=x+1$, using the decNumberAdd function, can be several times faster than $x=y+1$.

- Each function forms its primary result by setting the content of one of the structures referenced by the arguments; a pointer to this structure is returned by the function.

- Exceptional conditions and errors are reported by setting a bit in the *status* field of a referenced decContext structure (see page 14). The corresponding bit in the *traps* field of the decContext structure determines whether a trap is then raised, as also described earlier.

- If an argument to a function is *corrupt* (it is a `NULL` reference, or it is an input argument and the content of the structure it references is inconsistent), the function is unprotected (may "crash") unless `DECCHECK` is enabled (see the next rule). However, in normal operation (that is, no argument is corrupt), the result will always be a valid decNumber structure. The value of the decNumber result may be infinite or a quiet NaN if an error was detected (*i.e.*, if one of the `DEC_Errors` bits (see page 16) is set in the decContext *status* field).

- For best performance, input operands are assumed to be valid (not corrupt) and are not checked unless `DECCHECK` (see page 42) is 1, which enables full operand checking (including `NULL` operands). Whether `DECCHECK` is 0 or 1, the value of a result is undefined if an argument is corrupt. `DECCHECK` checking is a diagnostic tool only; it will report the error and prevent code failure by ensuring that results are valid numbers (unless the result reference is `NULL`), but it does not attempt to correct arguments.

Conversion functions

The conversion functions build a decNumber from a string, or lay out a decNumber as a character string.

decNumberFromString(number, string, context)

This function is used to convert a character string to decNumber format. It implements the **to–number** conversion from the arithmetic specification.

The conversion is exact provided that the numeric string has no more significant digits than are specified in `context.digits`. If there are more digits in the string, the value will be rounded to fit, using the `context.round` rounding mode. The `context.digits` field therefore both determines the maximum precision for unrounded numbers and defines the minimum size of the decNumber structure required.

The arguments are:

number (decNumber *) Pointer to the structure to be set from the character string.

string (char *) Pointer to the input character string. This must be a valid numeric string, as defined in the appropriate specification. The string will not be altered.

context (decContext *) Pointer to the context structure whose *digits*, *emin*, and *emax* fields indicate the maximum acceptable precision and exponent range, and whose *status* field is used to report any errors. If its *extended* field is 1, then special values (±Inf, ±Infinity, ±NaN, or ±sNaN, independent of case) are accepted, and the sign and exponent of zeros are preserved. NaNs may also specify diagnostic information as a string of digits immediately following the name.

Returns *number*.

Possible errors are DEC_Conversion_syntax (the string does not have the syntax of a number, which depends on the setting of *extended* in the context), DEC_Overflow (the adjusted exponent of the number is larger than context.emax), or DEC_Underflow (the adjusted exponent is less than context.emin and the conversion is not exact). If any of these conditions are set, the *number* structure will have a defined value as described in the arithmetic specification (this may be a subnormal or infinite value).

decNumberToString(number, string)

This function is used to convert a decNumber number to a character string, using scientific notation if an exponent is needed (that is, there will be just one digit before any decimal point). It implements the **to–scientific–string** conversion.

The arguments are:

number (decNumber *) Pointer to the structure to be converted to a string.

string (char *) Pointer to the character string buffer which will receive the converted number. It must be at least 14 characters longer than the number of digits in the number (number->digits).

Returns *string*.

No error is possible from this function. Note that non-numeric strings (one of +Infinity, –Infinity, NaN, or sNaN) are possible, and NaNs may have a – sign and/or diagnostic information.

decNumberToEngString(number, string)

This function is used to convert a decNumber number to a character string, using engineering notation (where the exponent will be a multiple of three, and there may be up to three digits before any decimal point) if an exponent is needed. It implements the **to–engineering–string** conversion.

The arguments and result are the same as for the decNumberToString function, and similarly no error is possible from this function.

Arithmetic functions

The arithmetic functions all follow the same syntax and rules, and are summarized below. They all take the following arguments:

number (`decNumber *`) Pointer to the structure where the result will be placed.

lhs (`decNumber *`) Pointer to the structure which is the left hand side (lhs) operand for the operation. This argument is omitted for monadic operations.

rhs (`decNumber *`) Pointer to the structure which is the right hand side (rhs) operand for the operation.

context (`decContext *`) Pointer to the context structure whose settings are used for determining the result and for reporting any exceptional conditions.

Each function returns *number*.

Some functions, such as decNumberExp, as described as *mathematical functions*. These have some restrictions: `context.emax` must be < 10^6, `context.emin` must be > -10^6, and `context.digits` must be < 10^6. Non-zero operands to these functions must also fit within these bounds.

The precise definition of each operation can be found in the specification documents.

decNumberAbs(number, rhs, context)

The *number* is set to the absolute value of the *rhs*. This has the same effect as decNumberPlus unless *rhs* is negative, in which case it has the same effect as decNumberMinus.

decNumberAdd(number, lhs, rhs, context)

The *number* is set to the result of adding the *lhs* to the *rhs*.

decNumberCompare(number, lhs, rhs, context)

This function compares two numbers numerically. If the *lhs* is less than the *rhs* then the *number* will be set to the value –1. If they are equal (that is, when subtracted the result would be 0), then *number* is set to 0. If the *lhs* is greater than the *rhs* then the *number* will be set to the value 1. If the operands are not comparable (that is, one or both is a NaN) the result will be NaN.

decNumberCompareTotal(number, lhs, rhs, context)

This function compares two numbers using the IEEE 754r proposed ordering. If the *lhs* is less than the *rhs* in the total order then the *number* will be set to the value –1. If they are equal, then *number* is set to 0. If the *lhs* is greater than the *rhs* then the *number* will be set to the value 1.

The total order differs from the numerical comparison in that: –NaN < –sNaN < –Infinity < –finites < –0 < +0 < +finites < +Infinity < +sNaN < +NaN. Also, 1.000 < 1.0 (*etc.*) and NaNs are ordered by payload.

decNumberDivide(number, lhs, rhs, context)

The *number* is set to the result of dividing the *lhs* by the *rhs*.

decNumberDivideInteger(number, lhs, rhs, context)

The *number* is set to the integer part of the result of dividing the *lhs* by the *rhs*.

Note that it must be possible to express the result as an integer. That is, it must have no more digits than `context.digits`. If it does then `DEC_Division_impossible` is raised.

decNumberExp(number, rhs, context)

The *number* is set to *e* raised to the power of *rhs*, rounded if necessary using the digits setting in the *context* and using the *round–half–even* rounding algorithm.

Finite results will always be full precision and inexact, except when *rhs* is a zero or `–Infinity` (giving 1 or 0 respectively). Inexact results will almost always be correctly rounded, but may be up to 1 *ulp* (unit in last place) in error in rare cases.

This is a mathematical function; the 10^6 restrictions on precision and range apply as described above.

decNumberLn(number, rhs, context)

The *number* is set to the natural logarithm (logarithm in base *e*) of *rhs*, rounded if necessary using the digits setting in the *context* and using the *round–half–even* rounding algorithm. *rhs* must be positive or a zero.

Finite results will always be full precision and inexact, except when *rhs* is equal to 1, which gives an exact result of 0. Inexact results will almost always be correctly rounded, but may be up to 1 *ulp* (unit in last place) in error in rare cases.

This is a mathematical function; the 10^6 restrictions on precision and range apply as described above.

decNumberLog10(number, rhs, context)

The *number* is set to the logarithm in base ten of *rhs*, rounded if necessary using the digits setting in the *context* and using the *round–half–even* rounding algorithm. *rhs* must be positive or a zero.

Finite results will always be full precision and inexact, except when *rhs* is equal to an integral power of ten, in which case the result is the exact integer.

Inexact results will almost always be correctly rounded, but may be up to 1 *ulp* (unit in last place) in error in rare cases.

This is a mathematical function; the 10^6 restrictions on precision and range apply as described above.

decNumberMax(number, lhs, rhs, context)

This function compares two numbers numerically and sets *number* to the larger. If the numbers compare equal then *number* is chosen with regard to sign and exponent. Unusually, if one operand is a quiet NaN and the other a number, then the number is returned.

decNumberMin(number, lhs, rhs, context)

This function compares two numbers numerically and sets *number* to the smaller. If the numbers compare equal then *number* is chosen with regard to sign and exponent. Unusually, if one operand is a quiet NaN and the other a number, then the number is returned.

decNumberMinus(number, rhs, context)

The *number* is set to the result of subtracting the *rhs* from 0. That is, it is negated, following the usual arithmetic rules; this may be used for implementing a prefix minus operation.

decNumberMultiply(number, lhs, rhs, context)

The *number* is set to the result of multiplying the *lhs* by the *rhs*.

decNumberNormalize(number, rhs, context)

This function has the same effect as decNumberPlus except that the final result is set to its simplest form. That is, a non-zero number which has any trailing zeros in the coefficient has those zeros removed by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly, and a zero has its exponent set to 0.

decNumberPlus(number, rhs, context)

The *number* is set to the result of adding the *rhs* to 0. This takes place according to the settings given in the *context*, following the usual arithmetic rules. This may therefore be used for rounding or for implementing a prefix plus operation.

decNumberPower(number, lhs, rhs, context)

The *number* is set to the result of raising the *lhs* to the power of the *rhs*, rounded if necessary using the settings in the *context*.

Results will be exact when the *rhs* has an integral value and the result does not need to be rounded, and also will be exact in certain special cases, such as when the *lhs* is a zero (see the arithmetic specification for details).

Inexact results will always be full precision, and will almost always be correctly rounded, but may be up to 1 *ulp* (unit in last place) in error in rare cases.

This is a mathematical function; the 10^6 restrictions on precision and range apply as described above, except that the normal range of values and context is allowed if the *rhs* has an integral value in the range –1999999997 through +999999999.[10]

decNumberQuantize(number, lhs, rhs, context)

This function is used to modify a number so that its exponent has a specific value, equal to that of the *rhs*. The decNumberRescale (see page 28) function may also be used for this purpose, but requires the exponent to be given as a decimal number.

---

[10] This relaxation of the restrictions provides upwards compatibility with an earlier version of the decNumberPower function which could only handle an *rhs* with an integral value.

When *rhs* is a finite number, its *exponent* is used as the requested exponent (it provides a “pattern” for the result). Its coefficient and sign are ignored.

The *number* is set to a value which is numerically equal (except for any rounding) to the *lhs*, modified as necessary so that it has the requested exponent. To achieve this, the *coefficient* of the *number* is adjusted (by rounding or shifting) so that its *exponent* has the requested value. For example, if the *lhs* had the value `123.4567`, and the *rhs* had the value `0.12`, the result would be `123.46` (that is, `12346` with an *exponent* of –2, matching the exponent of the *rhs*).

Note that the exponent of the *rhs* may be positive, which will lead to the *number* being adjusted so that it is a multiple of the specified power of ten.

If adjusting the exponent would mean that more than `context.digits` would be needed in the *coefficient*, then the `DEC_Invalid_operation` condition is raised. This guarantees that in the absence of error the exponent of *number* is always equal to that of the *rhs*.

If either operand is a *special value* then the usual rules apply, except that if either operand is infinite and the other is finite then the `DEC_Invalid_operation` condition is raised, or if both are infinite then the result is the first operand.

decNumberRemainder(number, lhs, rhs, context)

The *number* is set to the remainder when *lhs* is divided by the *rhs*.

That is, if the same *lhs*, *rhs*, and *context* arguments were given to the decNumberDivideInteger and decNumberRemainder functions, resulting in *i* and *r* respectively, then the identity

$$lhs = (i \times rhs) + r$$

holds.

Note that, as for decNumberDivideInteger, it must be possible to express the integer part of the result as an integer. That is, it must have no more digits than `context.digits`. If it does then `DEC_Division_impossible` is raised.

decNumberRemainderNear(number, lhs, rhs, context)

The *number* is set to the remainder when *lhs* is divided by the *rhs*, using the rules defined in IEEE 854. This follows the same definition as decNumberRemainder, except that the nearest integer (or the nearest even integer if the remainder is equidistant from two) is used for *i* instead of the result from decNumberDivideInteger.

For example, if *lhs* had the value 10 and *rhs* had the value 6 then the result would be -2 (instead of 4) because the nearest multiple of 6 is 12 (rather than 6).

decNumberRescale(number, lhs, rhs, context)

This function is used to rescale a number so that its exponent has a specific value, given by the *rhs*. The decNumberQuantize (see page 27) function may also be used for this purpose, and is often easier to use.

The *rhs* must be a whole number (before any rounding); that is, any digits in the fractional part of the number must be zero. It must have no more than nine digits, or `context.digits` digits, (whichever is smaller) in the integer part of the number.

The *number* is set to a value which is numerically equal (except for any rounding) to the *lhs*, rescaled so that it has the requested exponent. To achieve this, the *coefficient* of the *number* is adjusted (by rounding or shifting) so that its *exponent* has the value of the *rhs*. For example, if the *lhs* had the value `123.4567`, and decNumberRescale was used to set its exponent to –2, the result would be `123.46` (that is, `12346` with an *exponent* of –2).

Note that the *rhs* may be positive, which will lead to the *number* being adjusted so that it is a multiple of the specified power of ten.

If adjusting the scale would mean that more than `context.digits` would be needed in the *coefficient*, then the `DEC_Invalid_operation` condition is raised. This guarantees that in the absence of error the exponent of *number* is always equal to the *rhs*.

decNumberSameQuantum(number, lhs, rhs)

This function is used to test whether the exponents of two numbers are equal. The coefficients and signs of the operands (*lhs* and *rhs*) are ignored.

If the exponents of the operands are equal, or if they are both Infinities or they are both NaNs, *number* is set to 1. In all other cases, *number* is set to 0. No error is possible.

decNumberSquareRoot(number, rhs, context)

The *number* is set to the square root of the *rhs*, rounded if necessary using the digits setting in the *context* and using the *round–half–even* rounding algorithm. The preferred exponent of the result is `floor(exponent/2)`.

decNumberSubtract(number, lhs, rhs, context)

The *number* is set to the result of subtracting the *rhs* from the *lhs*.

decNumberToIntegralValue(number, rhs, context)

The *number* is set to the *rhs*, with any fractional part removed if necessary using the rounding mode in the *context*.

No error is possible, no flags are set (unless the operand is a signaling NaN), and the result may have a positive exponent.

Utility functions

The utility functions provide for copying, trimming, and zeroing numbers, and for determining the version of the decNumber package.

decNumberCopy(number, source)

This function is used to copy the content of one decNumber structure to another. It is used when the structures may be of different sizes and hence a straightforward structure copy by C assignment is inappropriate. It also may have performance benefits when the number is short relative to the size of the structure, as only the units containing the digits in use in the source structure are copied.

The arguments are:

number (`decNumber *`) Pointer to the structure to receive the copy. It must have space for `source->digits` digits.

source (`decNumber *`) Pointer to the structure which will be copied to *number*. All fields are copied, with the units containing the `source->digits` digits being copied starting from *lsu*. The *source* structure is unchanged.

Returns *number*. No error is possible from this function.

decNumberIsInfinite(number)

This function is used to test whether a number is infinite.

The argument is:

number (`decNumber *`) Pointer to the structure whose value is to be tested.

Returns 1 (true) if the number is infinite, or 0 (false) otherwise. This function may be implemented as a macro; no error is possible.

decNumberIsNaN(number)

This function is used to test whether a number is a NaN (quiet or signaling).

The argument is:

number (`decNumber *`) Pointer to the structure whose value is to be tested.

Returns 1 (true) if the number is a NaN, or 0 (false) otherwise. This function may be implemented as a macro; no error is possible.

decNumberIsNegative(number)

This function is used to test whether a number is negative (either minus zero or less than zero).

The argument is:

number (`decNumber *`) Pointer to the structure whose value is to be tested.

Returns 1 (true) if the number is negative, or 0 (false) otherwise. This function may be implemented as a macro; no error is possible.

decNumberIsQNaN(number)

This function is used to test whether a number is a Quiet NaN.

The argument is:

number (decNumber *) Pointer to the structure whose value is to be tested.

Returns 1 (true) if the number is a Quiet NaN, or 0 (false) otherwise. This function may be implemented as a macro; no error is possible.

decNumberIsSNaN(number)

This function is used to test whether a number is a Signaling NaN.

The argument is:

number (decNumber *) Pointer to the structure whose value is to be tested.

Returns 1 (true) if the number is a Signaling NaN, or 0 (false) otherwise. This function may be implemented as a macro; no error is possible.

decNumberIsZero(number)

This function is used to test whether a number is a zero (either positive or negative).

The argument is:

number (decNumber *) Pointer to the structure whose value is to be tested.

Returns 1 (true) if the number is zero, or 0 (false) otherwise. This function may be implemented as a macro; no error is possible.

decNumberTrim(number)

This function is used to remove insignificant trailing zeros from a number. That is, if the number has any fractional trailing zeros they are removed by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly.

The argument is:

number (decNumber *) Pointer to the structure whose value is to be trimmed.

Returns *number*. No error is possible from this function.

decNumberVersion()

This function returns a pointer (char *) to a human-readable description of the version of the decNumber package being run. The string pointed to will have at most 16 characters and will be a constant, and will comprise two words (the name and a decimal number identifying the version) separated by a blank. For example:

```
decNumber 3.02
```

No error is possible from this function.

decNumberZero(number)

This function is used to set the value of a decNumber structure to zero.

The argument is:

number (`decNumber *`) Pointer to the structure to be set to 0. It must have space for one digit.

Returns *number*. No error is possible from this function.

decimal32, decimal64, and decimal128 modules

The decimal32, decimal64, and decimal128 modules define the data structures and functions for compressed formats which are 32, 64, or 128 bits (4, 8, or 16 bytes) long, respectively. These provide up to 7, 16, or 34 digits of decimal precision in a compact and machine-independent form. Details of the formats are available at:

`http://www2.hursley.ibm.com/decimal/decbits.html`

Apart from the different lengths and ranges of the numbers, the three modules are identical, so this section just describes the decimal64 format. The definitions and functions for the other two formats are identical, except for the obvious name and value changes.

Note that these formats are now included in the draft of the proposed IEEE-SA 754 standard ("754r"). However, they are still subject to change; use at your own risk.

In this implementation each format is represented as an array of unsigned bytes. There is therefore just one field in the decimal64 structure:

bytes The *bytes* field represents the eight bytes of a decimal64 number, using Densely Packed Decimal encoding for the coefficient.[11]

The storage of a number in the bytes array may be chosen to either follow the byte ordering ("endianness") of the computing platform or to use fixed ordering (big-endian, with `bytes[0]` containing the sign bit of the format). This choice is made at compile time by setting the `DECENDIAN` tuning parameter (see page 42).

The decimal64 module includes private functions for coding and decoding Densely Packed Decimal data; these functions are shared by the other compressed format modules.

Definitions

The `decimal64.h` header file defines the decimal64 data structure described above. It includes the `decNumber.h` header file, to simplify use, and (if not already defined) it sets the `DECNUMDIGITS` constant to 16, so that any declared decNumber will be the right size to contain any decimal64 number.

If more than one of the three decimal format header files are used in a program, they must be included in decreasing order of size so that the largest value of `DECNUMDIGITS` will be used.

The `decimal64.h` header file also contains:

- Constants defining aspects of decimal64 numbers, including the maximum precision, the minimum and maximum (adjusted) exponent supported, the bias applied to the

[11] See `http://www2.hursley.ibm.com/decimal/DPDecimal.html` for a summary of Densely Packed Decimal encoding.

exponent, the length of the number in bytes, and the maximum number of characters in the string form of the number (including terminator).

- Macros for accessing the leading fields of the number (comprising the sign, combination field, and exponent continuation).
- Definitions of the public functions in the decimal64 module.

The decimal64 module also contains the shared routines for compressing and expanding Densely Packed Decimal data, and uses the `decDPD.h` header file. The latter contains look-up tables which are used for encoding and decoding Densely Packed Decimal data (only two tables of the four tables are used in a given compilation). These tables are automatically generated and should not need altering.

Functions

The `decimal64.c` source file contains the public functions defined in the header file. These comprise conversions to and from strings, and to and from decNumber form.

When a decContext structure is used to report errors, the same rules are followed as for other modules. That is, a trap may be raised, *etc.*

decimal64FromString(decimal64, string, context)

This function is used to convert a character string to decimal64 format. It implements the **to–number** conversion in the arithmetic specification (that is, it accepts subnormal numbers, NaNs, and infinities, and it preserves the sign and exponent of 0). If necessary, the value will be rounded to fit.

The arguments are:

decimal64 (`decimal64 *`) Pointer to the structure to be set from the character string.

string (`char *`) Pointer to the input character string. This must be a valid numeric string, as defined in the specification. The string will not be altered.

context (`decContext *`) Pointer to the context structure whose *status* field is used to control the conversion and report any error, as for the decNumberFromString function (see page 23) except that the precision and exponent range are fixed for each format.

Returns *decimal64*.

Possible errors are `DEC_Conversion_syntax` (the string does not have the syntax of a number), `DEC_Overflow` (the adjusted exponent of the number is positive and is greater than `context.emax`), or `DEC_Underflow` (the adjusted exponent of the number is negative and is less than `context.emin` and the conversion is not exact). If one of these conditions is set, the *decimal64* structure will have the value `NaN`, `Infinity`, or a finite (possibly subnormal) number respectively, with the same sign as the converted number after overflow or underflow.

decimal64ToString(decimal64, string)

This function is used to convert a decimal64 number to a character string, using scientific notation if an exponent is needed (that is, there will be just one digit before any decimal point). It implements the **to–scientific–string** conversion in the arithmetic specification.

The arguments are:

decimal64 (`decimal64 *`) Pointer to the structure to be converted to a string.

string (`char *`) Pointer to the character string buffer which will receive the converted number. It must be at least `DECIMAL64_String` (24) characters long.

Returns *string*.

No error is possible from this function.

decimal64ToEngString(decimal64, string)

This function is used to convert a decimal64 number to a character string, using engineering notation (where the exponent will be a multiple of three, and there may be up to three digits before any decimal point) if an exponent is needed. It implements the **to–engineering–string** conversion in the arithmetic specification.

The arguments and result are the same as for the decimal64ToString function, and similarly no error is possible from this function.

decimal64FromNumber(decimal64, number, context)

This function is used to convert a decNumber to decimal64 format.

The arguments are:

decimal64 (`decimal64 *`) Pointer to the structure to be set from the decNumber. This may receive a numeric value (including subnormal values and –0) or a special value.

number (`decNumber *`) Pointer to the input structure. The decNumber structure will not be altered.

context (`decContext *`) Pointer to a context structure whose *status* field is used to report any error and whose other fields are used to control rounding, *etc.*, as required.

Returns *decimal64*.

The possible errors are as for the decimal64FromString function (see page 34), except that `DEC_Conversion_syntax` is not possible.

decimal64ToNumber(decimal64, number)

This function is used to convert a decimal64 number to decNumber form in preparation for arithmetic or other operations.

The arguments are:

decimal64 (`decimal64 *`) Pointer to the structure to be converted to a decNumber. The decimal64 structure will not be altered.

number (`decNumber *`) Pointer to the result structure. It must have space for 16 digits of precision.

Returns *number*.

No error is possible from this function.

decPacked module

The decPacked module provides conversions to and from Packed Decimal numbers. Unlike the other modules, no specific decPacked data structure is defined because packed decimal numbers are usually held as simple byte arrays, with a scale either being held separately or implied.

Packed Decimal numbers are held as a sequence of Binary Coded Decimal digits, most significant first (at the lowest offset into the byte array) and one per 4 bits (that is, each digit taking a value of 0–9, and two digits per byte), with optional leading zero digits. The final sequence of 4 bits (called a "*nibble*") will have a value greater than nine which is used to represent the sign of the number. The sign nibble may be any of the six possible values:

`1010` (`0x0a`) plus

`1011` (`0x0b`) minus

`1100` (`0x0c`) plus (preferred)

`1101` (`0x0d`) minus (preferred)

`1110` (`0x0e`) plus

`1111` (`0x0f`) plus[12]

Packed Decimal numbers therefore represent decimal integers. They often have associated with them a second integer, called a *scale*. The scale of a number is the number of digits that follow the decimal point, and hence, for example, if a Packed Decimal number has the value `–123456` with a scale of 2, then the value of the combination is `–1234.56`.

Definitions

The `decPacked.h` header file does not define a specific data structure for Packed Decimal numbers.

It includes the `decNumber.h` header file, to simplify use, and (if not already defined) it sets the `DECNUMDIGITS` constant to 32, to allow for most common uses of Packed Decimal numbers. If you wish to work with higher (or lower) precisions, define `DECNUMDIGITS` to be the desired precision before including the `decPacked.h` header file.

The `decPacked.h` header file also contains:

- Constants describing the six possible values of sign nibble, as described above.
- Definitions of the public functions in the decPacked module.

---

12 Conventionally, this sign code can also be used to indicate that a number was originally unsigned.

Functions

The `decPacked.c` source file contains the public functions defined in the header file. These provide conversions to and from decNumber form.

decPackedFromNumber(bytes, length, scale, number)

This function is used to convert a decNumber to Packed Decimal format.

The arguments are:

bytes (`uint8_t *`) Pointer to an array of unsigned bytes which will receive the number.

length (`int32_t`) Contains the length of the byte array, in bytes.

scale (`int32_t *`) Pointer to an `int32_t` which will receive the scale of the number.

number (`decNumber *`) Pointer to the input structure. The decNumber structure will not be altered.

Returns *bytes* unless the decNumber has too many digits to fit in *length* bytes (allowing for the sign) or is a special value (an infinity or NaN), in which cases `NULL` is returned and the *bytes* and *scale* values are unchanged.

The number is converted to bytes in Packed Decimal format, right aligned in the *bytes* array, whose length is given by the second parameter. The final 4-bit nibble in the array will be one of the preferred sign nibbles, `1100` (`0x0c`) for + or `1101` (`0x0d`) for –. The maximum number of digits that will fit in the array is therefore *length*×2–1. Unused bytes and nibbles to the left of the number are set to 0.

The *scale* is set to the scale of the number (this is the exponent, negated). To force the number to a particular scale, first use the `decNumberRescale` function (see page 28) on the number, negating the required scale in order to adjust its *exponent* and *coefficient* as necessary.

decPackedToNumber(bytes, length, scale, number)

This function is used to convert a Packed Decimal format number to decNumber form in preparation for arithmetic or other operations.

The arguments are:

bytes (`uint8_t *`) Pointer to an array of unsigned bytes which contain the number to be converted.

length (`int32_t`) Contains the length of the byte array, in bytes.

scale (`int32_t *`) Pointer to an `int32_t` which contains the scale of the number to be converted. This must be set; use 0 if the number has no associated scale (that is, it is an integer). The effective exponent of the resulting number (that is, the number of significant digits in the number, less the *scale*, less 1) must fit in 9 decimal digits.

number (`decNumber *`) Pointer to the decNumber structure which will receive the number. It must have space for *length*×2–1 digits.

Returns *number*, unless the effective exponent was out of range or the format of the *bytes* array was invalid (the final nibble was not a sign, or an earlier nibble was not in the range 0–9). In these error cases, `NULL` is returned and *number* will have the value 0.

Note that –0 and zeros with non-zero exponents are possible resulting numbers.

Additional options

This section describes some additional features of the decNumber package, intended to be used when extending the package or tuning its performance. If you are just using the package for applications, using full IEEE arithmetic, you should not need to modify the parameters controlling these features.

Tuning and testing parameters

The decNumber package incorporates a number of compile-time parameters. If any of these parameters is changed, all the decNumber source files being used must be recompiled to ensure correct operation.

Two parameters are used to tune the trade-offs between storage use and speed. The first of these determines the granularity of calculations (the number of digits per unit of storage) and is normally set to three or to a power of two. The second is normally set so that short numbers (tens of digits) require no storage management – working buffers for operations will be stack based, not dynamically allocated.

These are:

`DECDPUN` This parameter is set in the `decNumber.h` file, and must be an integer in the range 1 through 9. It sets the number of digits held in one *unit* (see page 21), which in turn alters the performance and other characteristics of the library. In particular:

- If `DECDPUN` is 1, conversions are fast, but arithmetic operations are at their slowest. In general, as the value of `DECDPUN` increases, arithmetic speed improves and conversion speed gets worse.
- Conversions between the decNumber internal format and the decimal64 and other compressed formats are fastest – sometimes by as much as a factor of 4 or 5 – when `DECDPUN` is 3 (because Densely Packed Decimal encodes digits in groups of three).
- If `DECDPUN` is not 1, 3, or a power of two, calculations converting digits to units and vice versa are slow; this may slow some operations by up to 20%.
- If `DECDPUN` is greater than 4, either non-ANSI-89 C integers or library calls have to be used for 64-bit intermediate calculations.[13]

The suggested value for `DECDPUN` is 3, which gives good performance for working with the compressed decimal formats. If the compressed formats are not being used, or 64-bit integers are unavailable (see `DECUSE64`, below), then measuring the effect of changing `DECDPUN` to 4 is suggested. If the library is to be used for high precision calculations (many tens of digits) then it is recommended that measurements be made to evaluate whether to set `DECDPUN` to 8 (or possibly to 9, though this will often be slower).

`DECBUFFER` This parameter is set in the `decNumberLocal.h` file, and must be a non-negative integer. It sets the precision, in digits, which the operator functions will handle without allocating dynamic storage.[14]

---

13 The decNumber library currently assumes that non-ANSI-89 64-bit integers are available if DECDPUN is greater than 4. See also the DECUSE64 tuning parameter.

14 Dynamic storage may still be allocated in certain cases, but in general this is rare.

One or more DECBUFFER-sized buffers will be allocated on the stack, depending on the function; comparison, additions, subtractions, and exponentiation all allocate one, multiplication allocates two, and division allocates three; more complex operations may allocate more. It is recommended that DECBUFFER be a multiple of DECDPUN and also a multiple of 4, and large enough to hold common numbers in your application.

A third compile-time parameter controls the layout of the compressed decimal formats (see page 33). The storage of a number in these formats may be chosen to either follow the byte ordering ("endianness") of the computing platform or to use fixed ordering. For best performance when using these formats, this parameter should be set to 1. The parameter is set in the `decNumberLocal.h` file, and is:

DECENDIAN This must be either 1 or 0. If 1, which is recommended, the formats will be stored following the endianness of the underlying computing platform. For example, for AMD and Intel x86 architecture machines, which are *little–endian*, the byte containing the sign bit of the format is at the highest memory address; for IBM z-Series machines, which are *big–endian*, the byte containing the sign bit of the format is at the lowest memory address. This setting means that the decimal formats will be stored using the same ordering as binary integer and floating-point formats on the same machine, and also allows much faster conversions (up to a factor of three) to and from the decNumber internal form.

Setting DECENDIAN to 0 forces the formats to be stored using fixed, big-endian, ordering. This is provided for compatibility with earlier versions of the decNumber package.

A fourth compile-time parameter allows the use of 64-bit integers to improve the performance of certain operations (notably multiplication and the mathematical functions), even when DECDPUN is less than 5. (64-bit integers are required when DECDPUN is 5 or more.) The parameter is set in the `decNumberLocal.h` file, and is:

DECUSE64 This must be either 1 or 0. If 1, which is recommended, 64-bit integers will be used for most multiplications and mathematical functions when DECDPUN<=4, and for most operations when DECDPUN>4. If set to 0, 64-bit integer support is not used when DECDPUN<=4, and the maximum value for DECDPUN is then 4.

Three further compile-time parameters control the inclusion of extra code which provides for full checking of input arguments, run-time internal tracing control, and storage allocation auditing. These options are usually disabled, for best performance, but are useful for testing and when introducing new conversion routines, *etc.* These parameters are all set in the `decNumberLocal.h` file, and are:

DECCHECK This must be either 1 or 0. If 1, code which checks input structure references will be included in the module. This checks that the structure references are not NULL, and that they refer to valid (internally consistent in the current context) structures. If an invalid reference is detected, the `DEC_Invalid_operation` status bit is set (which may cause a trap), and any result will be a valid number of undefined value. This option is useful for verifying programs which construct decNumber structures explicitly.

Some operations take more than twice as long with this checking enabled, so it is normally assumed that all decNumbers are valid and `DECCHECK` is set to 0.

`DECALLOC` This must be either 1 or 0. If 1, all dynamic storage usage is audited and extra space is allocated to enable buffer overflow corruption checks. The cost of these checks is fairly small, but the setting should normally be left as 0 unless changes are being made to the `decNumber.c` source file.

`DECTRACE` This must be either 1 or 0. If 1, certain critical values are traced (using `printf`) as operations take place. This is intended for development use only, so again should normally be left as 0.

A final compile-time parameter enables the inclusion of extra code which implements and enforces the subset arithmetic defined by ANSI X3.274. This option should be disabled, for best performance, unless the subset arithmetic is required. The parameter is set in the `decContext.h` file, and is:

`DECSUBSET` This must be either 1 or 0. If 1, subset arithmetic is enabled. This setting includes the *extended* flag in the decContext structure and all code which depends on that flag. Setting `DECSUBSET` to 0 improves the performance of many operations by 10%–20%.

Appendix – Changes

This appendix documents changes since the first (internal) release of this document (Draft 1.50, 21 Feb 2001).

Changes in Draft 1.60 (9 July 2001)

- The significand of a number has been renamed from *integer* to *coefficient*, to remove possible ambiguities.
- The **decNumberRescale** function has been redefined to match the base specification. In particular its *rhs* now specifies the new exponent directly, rather than as a negated exponent.
- In general, all functions now return a reference to their primary result structure.
- The **decPackedToNumber** function now handles only "classic" Packed Decimal format (there must be a sign nibble, which must be the final nibble of the packed bytes). This improved conversion speed by a factor of two.
- Minor clarifications and editorial changes have been made.

Changes in Draft 1.65 (25 September 2001)

- The rounding modes `DEC_ROUND_CEILING` and `DEC_ROUND_FLOOR` have been added.
- Minor clarifications and editorial changes have been made.

Changes in Version 2.00 (4 December 2001)

This is the first public release of this document.

- The **decDoubleToSingle** function will now round the value of the decDouble number if it has more than 15 digits.
- The **decNumberToInteger**, **decNumberRemainderNear**, and **decNumberVersion** functions have been added.
- Relatively minor changes have been made throughout to reflect support for the extended specification.

Changes in Version 2.11 (25 March 2002)

- The header files have been reorganized in order to move private type names (such as `Int` and `Flag`) out of the external interface header files. In the external interface, integer types now use the `stdint.h` names from C99.
- All but one of the compile-time parameters have been moved to the "internal" `decNumberLocal.h` header file, and so are described in a new section (see page 40).
- The **decNumberAbs**, **decNumberMax**, and **decNumberMin** functions have been added.
- Minor clarifications and editorial changes have been made.

Changes in Version 2.12 (23 April 2002)

- The **decNumberTrim** function has been added.
- The **decNumberRescale** function has been updated to match changed specifications; it now sets the exponent as requested even for zero values.
- Minor clarifications and editorial changes have been made.

Changes in Version 2.15 (5 July 2002)

The package has been updated to reflect the changes included in the combined arithmetic specification. These preserve more digits of the coefficient together with extended zero values if *extended* in the context is 1. Notably:

- The **decNumberDivide** and **decNumberPower** functions do not remove trailing zeros after the operation. (The **decNumberTrim** function can be used to effect this, if required.)
- A non-zero exponent on a zero value is now possible and is preserved in a manner consistent with other numbers (that is, zero is no longer a special case).
- The **decPackedToNumber** function has been enhanced to allow zeros with non-zero exponents to be converted without loss of information.

Changes in Version 2.17 (1 September 2002)

- The **decNumberFromString**, **decSingleFromString**, and **decDoubleFromString** functions will now round the coefficient of a number to fit, if necessary. They also now accept subnormal values and preserve the exponent of a 0. If an overflow or underflow occurs, the `DEC_Overflow` or `DEC_Underflow` conditions are raised, respectively.
- The package has been corrected to ensure that subnormal values are no more precise than permitted by IEEE 854.
- The underflow condition is now raised according to the IEEE 854 untrapped underflow criteria (instead of according to the IEEE 854 trapped criteria). That is, underflow is now only raised when a result is both subnormal and inexact.
- The `DEC_Subnormal` condition has been added so that subnormal results can be detected even if no Underflow condition is raised.
- Minor clarifications and editorial changes have been made.

Changes in Version 2.28 (1 November 2002)

- The **decNumberNormalize** function has been added, as an operator. This makes the coefficient of a number as short as possible while maintaining its numerical value.
- The **decNumberSquareRoot** function has been added. This returns the exact square root of a number, rounded to the specified precision and normalized.
- When the *extended* setting is 1, long operands are used without input rounding, to give a correctly rounded result (without double rounding). The DEC_Lost_digits flag can therefore only be set when *extended* is 0.
- Minor editorial changes have been made.

Changes in Version 3.04 (22 February 2003)

The major change in decNumber version 3 is the replacement of the decSingle and decDouble formats by the three new formats *decimal32*, *decimal64*, and *decimal128*. These formats are now included in an unapproved draft of the proposed IEEE-SA 754 standard. However, they are still subject to change; use at your own risk.

Related and other enhancements include:

- The exponent minimum field, *emin*, has been added to the decContext structure. This allows the unbalanced exponents used in the new formats.
- The exponent clamping flag, *clamp*, has been added to the decContext structure. This provides explicit exponent clamping as used in the new formats.
- A new condition flag, `DEC_Clamped` has been introduced. This reports any situation where the exponent of a finite result has been limited to fit in the available exponent range.
- The header file `bcd2dpd.h` has been renamed `decDPD.h` to better describe its function.
- The `DECSUBSET` tuning parameter has been added. This controls the inclusion of the code and flags required for subset arithmetic; when set to 0, the performance of many operations is improved by 10%–20%.
- Double rounding which was possible with certain subnormal results has been eliminated.
- Minor editorial changes have been made.

Changes in Version 3.09 (23 July 2003)

This version implements some minor changes which track changes agreed by the IEEE 754 revision committee.

- The **decNumberQuantize** function has been added. Its function is identical to **decNumberRescale** except that the second argument specifies the target exponent “by example” rather than by value.
- The **decNumberQuantize** and **decNumberRescale** functions now report `DEC_Invalid_operation` rather than `DEC_Overflow` if the result cannot fit.

- The **decNumberToInteger** function has been replaced by the **decNumberToIntegralValue** function. This implements the new rules for *round-to-integral-value* agreed by IEEE 754r. Notably:
  - the exponent is only set to zero if the operand had a negative exponent
  - the Inexact flag is not set.
- The **decNumberSquareRoot** function no longer normalizes. Its preferred exponent is floor(operand.exponent/2).

Changes in Version 3.12 (1 September 2003)

This version adds a new function and slightly reorganizes the decimalnn modules.

- The **decNumberSameQuantum** function has been added. This tests whether two numbers have the same exponents.
- The `decimal128.h`, `decimal64.h`, and `decimal32.h` header files now check that (if more than one is included) they are included in order of reducing size. This makes it harder to use a decNumber structure which is too small.
- . The shared DPD pack/unpack routines have been moved from `decimal32.c` to `decimal64.c`, because the latter is more likely to be used alone.

Changes in Version 3.16 (2 October 2003)

- NaN values may now use the coefficient to convey diagnostic information, and NaN sign information is propagated along with that information.
- The **decNumberQuantize** function now allows both arguments to be infinite, and treats NaNs in the same way as other functions.

Changes in Version 3.19 (21 November 2003)

- The **decNumberIsInfinite**, **decNumberIsNaN**, **decNumberIsNegative**, and **decNumberIsZero** functions have been added to simplify tests on numbers. These functions are currently implemented as macros.

Changes in Version 3.24 (25 August 2004)

- The **decNumberMax** and **decNumberMin** functions have been altered to conform to the *maxnum* and *minnum* functions proposed by IEEE 754r. That is, a total ordering is provided for numerical comparisons, and if one operand is a quiet NaN but the other is a number then the number is returned.
- The **decimal64FromString** function (and the same function for the other two formats) now uses the rounding mode provided in the context structure.

Changes in Version 3.25 (15 June 2005)

- Arguments to functions which are "input only" are now decorated with the *const* keyword to make the functions easier and safer to call from a C++ wrapper class.
- The performance of arithmetic when DECDPUN<=3 has been improved substantially; DECDPUN==3 performance is now similar to DECDPUN==4.
- An error in the decNumberRescale and decNumberQuantize functions has been corrected. This returned 1.000 instead of NaN for quantize(0.9998, 0.001) under a context with precision=3.

Changes in Version 3.32 (12 December 2005)

- The **decNumberExp** function has been added. This returns *e* raised to the power of the operand.
- The **decNumberLn** and **decNumberLog10** functions have been added. These return the natural logarithm (logarithm in base *e*) or the logarithm in the base ten of the operand, respectively.
- The **decNumberPower** function has been enhanced by removing restrictions; notably it now allows raising numbers to non-integer powers.
- The `DECENDIAN` tuning parameter (see page 42) has been added. This allows the compressed decimal formats (see page 33) to be stored using platform-dependent ordering for better performance and compatibility with binary formats. This parameter can be set to 0 to get the same (big-endian) ordering on all platforms, as in earlier versions of the decNumber package.
- The `DECUSE64` tuning parameter (see page 42) has been added. This allows 64-bit integers to be used to improve the performance of operations when `DECDPUN`<=4. This parameter can be set to 0 to ensure only 32-bit integers are used when `DECDPUN`<=4.
- The compressed decimal formats are widely used with the decNumber package, so the initial setting of `DECDPUN` has been changed to 3 (from 4), and `DECENDIAN` and `DECUSE64` are both set to 1 (to use platform ordering and 64-bit arithmetic). These settings significantly improve the speed of conversions to and from the compressed formats and the speed of multiplications and other operations.
- Minor clarifications and editorial changes have been made.

Changes in Version 3.36 (6 July 2006)

- The **decNumberCompareTotal** (total ordering comparison), **decNumberIsQNaN**, and **decNumberIsSNaN** functions have been added.

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

Z

