

The decNumber C library

Version 2.00 – 4th December 2001

Mike Cowlshaw

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

Table of Contents

Overview 1

User's Guide 3

- Example 1 – simple addition 4
- Example 2 – compound interest 5
- Example 3 – passive error handling 6
- Example 4 – active error handling 7
- Example 5 – decSingle and decDouble 9
- Example 6 – Packed Decimal numbers 11

Module descriptions 12

- decContext module 13
 - Definitions 14
 - Functions 15
- decNumber module 18
 - Definitions 20
 - Functions 22
 - Conversion functions 22
 - Arithmetic functions 24
 - Utility functions 26
- decSingle module 28
 - Definitions 28
 - Functions 29
- decDouble module 31
 - Definitions 31
 - Functions 31
- decPacked module 35
 - Definitions 35
 - Functions 36

Appendix – Changes 38

Index 39

Overview

The decNumber library is an implementation of **Standard Decimal Arithmetic**¹ (both the *base* and *extended specifications*), written in ANSI C. Standard Decimal Arithmetic is a specification for decimal arithmetic which meets the requirements of commercial, financial, and human-oriented applications while conforming to the relevant ANSI² and IEEE³ standards.

The library fully implements the specifications,⁴ and hence supports integer, fixed-point, and floating-point decimal numbers directly, together with infinities and NaN (Not a Number) 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), single length decimal floating-point (8-byte), and double length decimal floating-point (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.

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).

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

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

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

⁴ Except for one missing function, square root, which will be added later if time permits.

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. Three machine-independent and more compact storage formats are therefore provided for interchange. These are:

- decSingle* This is a “single precision” decimal floating-point representation, where each number is stored in 8 bytes. It provides 15 decimal digits of precision and 3 of exponent, in a format similar to that defined in IEEE 754-1985.⁵
- decDouble* This is a “double precision” decimal floating-point representation, where each number is stored in 16 bytes. It provides 33 decimal digits of precision and 4 of exponent, also similar to IEEE 754-1985.
- 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 decSingle and decDouble modules also provide conversions to and from character string format (using the functions in the decNumber module). Conversions between the decSingle and decDouble formats are also included.

Standards compliance

It is intended that the decNumber implementation complies with:

- the floating-point decimal arithmetic defined in ANSI X3.274-1996 (including errata through 2001)
- all requirements of IEEE 854-1987 except that:
 1. The values returned after overflow and underflow do not change when an exception is trapped. This is because of the difficulty of making this thread-safe, and also because the IEEE 854 definition does not generalize to the power operator.
 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 $\pm 10^{1,000,000,000}$).
 3. The string representations of NaN are "NaN" and "sNaN" (as proposed in the current IEEE review), rather than just "NaN" with an optional sign.
 4. One required function is missing (square root) – it is intended that this will be added later, as time permits.

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.

⁵ ANSI/IEEE 754-1985 – *IEEE Standard for Binary Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1985.

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; only the simplest calculations ($x=x+1$, for example) are faster. 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)
- single and double precision 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. 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 38                // work with up to 38 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 6).

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 – decSingle and decDouble

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 `decSingle` (see page 28) and `decDouble` (see page 31) forms are provided as efficient, machine-independent formats used for storing numbers of up to 15 or 33 decimal digits respectively, in 8 or 16 bytes. These formats are very 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 -- decSingle conversion and square
2. #include "decSingle.h"           // decSingle and number library
3. #include <stdio.h>               // for (s)printf
4.
5. int main(int argc, char *argv[]) {
6.     decSingle a;                 // working decSingle number
7.     decNumber d;                 // working number
8.     decContext set;              // working context
9.     char string[DECSINGLE_String]; // number->string buffer
10.    char hexes[25];               // single->hex buffer
11.    int i;                         // counter
12.
13.    decContextDefault(&set, DEC_INIT_SINGLE); // initialize
14.
15.    decSingleFromString(&a, argv[1], &set);
16.    // lay out the single as eight hexadecimal pairs
17.    for (i=0; i<8; i++) {
18.        sprintf(&hexes[i*3], "%02x ", a.bytes[i]);
19.    }
20.    decSingleToNumber(&a, &d);
21.    decNumberMultiply(&d, &d, &d, &set); // square it
22.    decNumberToString(&d, string);
23.    printf("%s => %s=> %s (squared)\n", argv[1], hexes, string);
24.    return 0;
25. } // main
```

Here, the `#include` on line 2 not only defines the `decSingle` type, but also includes the `decNumber` and `decContext` header files. Also, if `DECNUMDIGITS` (see page 19) has not already been defined, the `decSingle.h` file sets it to 15 so that any `decNumbers` declared will be exactly the right size to take any `decSingle` without overflow.

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

Line 15 converts the input argument word to a `decSingle` (with a function call very similar to `decNumberFromString`). Note that a `DEC_Conversion_overflow` would occur if the number needed more than 15 digits of precision.

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

Lines 20–22 show how `decSingles` are used. First the `decSingle` is converted to a `decNumber`, then arithmetic is 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 23). For example, if the input argument were “79”, the following would be displayed:

```
79 => 3f f0 00 00 00 00 00 79 => 6241 (squared)
```

The `decDouble` form is used in exactly the same way, for working with up to 33 digits of precision. In this case, only `decDouble.h` need be included, as this in turn includes `decSingle.h`.

The `decDouble` type has the same constants and functions as `decSingle` (with the obvious name changes) and, in addition, provides conversions between the `decSingle` and `decDouble` types.

Like `decSingle.h`, the `decDouble` header file defines the `DECNUMDIGITS` (see page 19) constant if it has not already been defined, in this case to 33 (the maximum precision a `decDouble` can encode).

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:

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

Then the following declarations are added to the main function:

```
1. uByte startpack[]={0x01, 0x00, 0x00, 0x0C};      // investment=100000
2. Int    startscale=0;
3. uByte ratepack[]={0x06, 0x5C};                  // rate=6.5%
4. Int    ratescale=1;
5. uByte yearspack[]={0x02, 0x0C};                  // years=20
6. Int    yearsscale=0;
7. uByte respack[16];                               // result, packed
8. Int    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 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 input arguments 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 decNumber header 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 type “int” is not used; instead the symbol “Int” is defined to be a 32-bit integer in `decContext.h` (see page 14).
- 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.

There are six fields in the decContext structure:

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 `Int`, 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. If the adjusted exponent for a result or conversion would be smaller than *-emax* then an underflow results.

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

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:

`ROUND_CEILING` Round towards +infinity.

`ROUND_DOWN` Round towards 0 (truncation).

`ROUND_FLOOR` Round towards -infinity.

`ROUND_HALF_DOWN` Round to nearest; if equidistant, round down.

`ROUND_HALF_EVEN` Round to nearest; if equidistant, round so that the final digit is even.

`ROUND_HALF_UP` Round to nearest; if equidistant, round up.

`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 `uInt` (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 `uInt`. 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.

extended The *extended* field controls special processing for mathematical applications. When 0, zeros are treated as in the base specification and the exponent range is balanced. When 1, a value of `-0` is 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 `-emax-(digits-1)` are possible).

extended is of type `Flag` (unsigned char).

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

Definitions

The `decContext.h` header file defines the base types used by the `decNumber` module (such as `Int`, a 32-bit integer, and `uInt`, an unsigned `Int`). It is therefore automatically included by `decNumber.h`. In addition to defining the base types and the `decContext` data structure described above, it 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.
- 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.

Three of the exceptional condition flags merit special attention; the `DEC_Rounded` flag⁷ is set whenever a result or input operand is rounded (even if only zero digits were discarded), the `DEC_Inexact` flag⁸ is set whenever a result is inexact (non-zero digits were discarded) due to rounding of input operands or the result, and the `DEC_Lost_digits` flag is set when an input operand is made inexact through rounding.

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 three 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 strongly 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 (Int) The kind of initialization to be performed. Currently only the three 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 ANSI X3.274 arithmetic 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`, and `DEC_Lost_digits` are 0), and *extended* is set to 0.

When *kind* is `DEC_INIT_SINGLE`, defaults for a single precision number using IEEE 854 rules are set. That is, the *digits* field is set to 15, the *emax* field is set to 999, the *round* field is set to `ROUND_HALF_EVEN`, the *status* field is cleared (all bits zero), the *traps* field is cleared (no traps are enabled), and *extended* is set to 1.

When *kind* is `DEC_INIT_DOUBLE`, defaults for a double precision number using IEEE 854 rules are set. That is, the *digits* field is set to 33, the *emax* field is set to 9999, and the other fields are set as for `DEC_INIT_SINGLE`.

Returns *context*.

⁷ The `DEC_Rounded` condition is defined in the extended decimal arithmetic specification.

⁸ The `DEC_Inexact` condition is defined in IEEE 854 and in the extended decimal arithmetic specification.

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 (*uInt*) 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.⁹

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.

⁹ If “private” bits were allowed, future extension of the library with other conditions would be impossible.

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.

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).

Compile-time parameters in the header file provide 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.*

Two further compile-time parameters tune the trade-offs between storage use and speed. The first of these 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. The second determines the granularity of calculations (the number of digits per unit of storage) and is normally set to a power of two.

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: $(-1)^{\text{sign}} \times \text{coefficient} \times 10^{\text{exponent}}$.

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 $\pm\text{infinity}$.

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 `Int` (signed integer), 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 $\text{exponent} + \text{digits} - 1$.
- 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 $-999,999,999 - (\text{precision} - 1)$, where *precision* is the digits setting from the context; the adjusted exponent may then have 10 digits.
- exponent* is of type `Int`.

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 value is $-\infty$), and in this case *digits* must be 1 and the other fields must be 0.

bits is of type `uByte` (unsigned byte). Masks for the named bits are defined in the header file.

lsu The *lsu* field is one or more *units* in length, 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 20). 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 4). 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 `Unit` (an unsigned byte or integer, depending 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 *decSingle* structures, so the *decNumber* structure is in some sense an "internal" representation; in particular, it is machine-dependent.¹⁰

¹⁰ The layout of an `Int` or `Unit` might be big-endian or little-endian, for example.

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}
```

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 parameters:

`DECBUFFER` This must be a non-negative integer. It sets the precision, in digits, which the operator functions will handle without allocating dynamic storage.¹¹

Up to three `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. The storage used for each buffer is given by $(\text{DECBUFFER} + \text{DECDPUN} - 1) / \text{DECDPUN} * \text{sizeof}(\text{Unit})$.

It is recommended that `DECBUFFER` be a multiple of `DECDPUN`, and large enough to hold common numbers in your application.

`DECDPUN` This must be an integer in the range 1 through 9. It sets the number of digits held in one *unit* (see page 19), 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.
- If `DECDPUN` is not 1 or a power of two, calculations converting digits to units and vice versa are slow; this slows all operations.

¹¹ Dynamic storage may still be allocated when operands need rounding, but in general this is rare.

- If `DECDPUN` is greater than 4, either non-ANSI C integers or library calls have to be used for 64-bit intermediate calculations.¹²

The suggested value for `DECDPUN` is 4, which gives good performance for commonplace numbers, and guarantees ANSI C code. If the library is to be used for high precision calculations (many tens of digits) then measurements should be made to evaluate whether to set `DECDPUN` to 8 or possibly 9.

If either of these parameters is changed, the `decNumber.c` source file must be recompiled for the changes to have effect.

- The conditional code parameters:

DECHECK 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 `DECHECK` 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 to arithmetic functions have been made in 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.

If any of these parameters are changed, the `decNumber.c` source file must be recompiled for the changes to have effect.

- Constants describing the maximum precision and adjusted exponent supported by the implementation.
- Constants naming the bits in the *bits* field, such as `DECNEG`, the sign bit.
- Definition of the `powers` array, used by several modules in the library. This array (located in `decnumber.c`) is of type `uInt[]` (unsigned integer) and comprises eleven elements, containing in sequence the value of each power of 10 from 0 through 10, where the power of 10 is determined by the offset into the array. The element `powers[3]`, for example, contains 1000.
- Definitions of the public functions in the `decNumber` module.

¹² The `decNumber` library currently assumes that non-ANSI 64-bit integers are available if `DECDPUN` is greater than 4.

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 13). 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 `DECHECK` 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 14) is set in the `decContext status` field).
- For best performance, input operands are assumed to be valid (not corrupt) and are not checked unless `DECHECK` (see page 21) is 1, which enables full operand checking (including `NULL` operands). Whether `DECHECK` is 0 or 1, the value of a result is undefined if an argument is corrupt. `DECHECK` 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 in the base specification or (if *extended* in the context is 1) the **to-extended-number** conversion in the extended specification.

The conversion is exact; if the numeric string has more significant digits than specified in the *context* an exceptional condition occurs. The `context.digits` field therefore

determines the maximum acceptable precision 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* and *emax* fields indicate the maximum acceptable precision and exponent, and whose *status* field is used to report any errors. If its *extended* field is 1, then special values ($\pm\text{Inf}$, $\pm\text{Infinity}$, NaN, or sNaN, independent of case) are accepted, and the sign of -0 is preserved.

Returns *number*.

Possible errors are `DEC_Conversion_syntax` (the string does not have the syntax of a number), `DEC_Conversion_overflow` (the number has more than `context.digits` significant digits or its adjusted exponent is larger than `context.emax`), or `DEC_Conversion_underflow` (the number has an acceptable number of significant digits but the adjusted exponent is less than `-context.emax`). If any of these conditions are set, the *number* structure will have a defined value as described in the extended specification.

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 of both specifications.

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 $+\text{Infinity}$, $-\text{Infinity}$, NaN, or sNaN) are possible.

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 in the base specification.

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:

<i>number</i>	(decNumber *) Pointer to the structure where the result will be placed.
<i>lhs</i>	(decNumber *) Pointer to the structure which is the left hand side (lhs) operand for the operation. This argument is omitted for the two monadic operators, decNumberPlus and decNumberMinus.
<i>rhs</i>	(decNumber *) Pointer to the structure which is the right hand side (rhs) operand for the operation.
<i>context</i>	(decContext *) Pointer to the context structure whose settings are used for determining the result and for reporting any exceptional conditions.

Each function returns *number*.

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

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.

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.

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*.

decNumberPlus(number, rhs, context)

The *number* is set to the result of adding the *rhs* to 0. That is, it is normalized 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*.

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.

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 *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 `DEC_Overflow` is raised. This guarantees that the *exponent* of the result is always as specified by the *rhs*, except when the result is 0.

decNumberSubtract(number, lhs, rhs, context)

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

decNumberToInteger(number, rhs, context)

The *number* is set to the *rhs*, rounded to an integer if necessary using the rounding mode in the *context*.

Unlike the `decNumberRescale` (see page 25) function, if more than `context.digits` would be needed in the *coefficient* to express the number with an exponent of 0 then the number is left unscaled (that is, the exponent may be positive).

Utility functions

The utility functions provide for copying 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 the fields of the structure 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.

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 2.00
```

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.

decSingle module

The decSingle module defines the data structure used for single precision decimal numbers, in a compact and machine-independent form. Single precision decimal numbers may have up to 15 decimal digits of precision and 3 decimal digits of exponent.¹³

The precise layout of a decSingle number is defined elsewhere;¹⁴ in this implementation it is represented as an array of unsigned bytes. There is therefore just one field in the decSingle structure:

bytes The *bytes* field represents the eight bytes of a single precision number using Densely Packed Decimal encoding for the coefficient.¹⁵

The decSingle module includes private functions for coding and decoding Densely Packed Decimal data; these functions are shared by the decDouble module.

Definitions

The `decSingle.h` header file defines the decSingle 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 15, so that any declared decNumber will be the right size to contain any decSingle number.

The `decSingle.h` header file also contains:

- Constants defining aspects of decSingle numbers, including the maximum precision and (adjusted) exponent supported, the bias applied to the 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, exponent, and reserved bits).
- Definitions of the public functions in the decSingle module.

A further header file, `bcd2dspd.h`, is also used by the decSingle module. This contains two look-up tables, used for encoding and decoding Densely Packed Decimal data. These tables are automatically generated and should not need altering.

¹³ The exponent can be 4 digits, for subnormal numbers.

¹⁴ See <http://www2.hursley.ibm.com/decimal/deccode.html>

¹⁵ See <http://www2.hursley.ibm.com/decimal/DPDecimal.html> for a summary of Densely Packed Decimal encoding.

Functions

The `decSingle.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. Functions for converting between the `decSingle` and `decDouble` forms are located in the `decDouble` module (see page 31).

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.*

decSingleFromString(single, string, context)

This function is used to convert a character string to `decSingle` format. It implements the **to-extended-number** conversion in the extended specification (that is, it accepts the special values $\pm\text{Inf}$, $\pm\text{Infinity}$, `NaN`, or `sNaN`, independent of case, and preserves `-0`).

The arguments are:

- single* (`decSingle *`) 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 base specification. The string will not be altered.
- context* (`decContext *`) Pointer to the context structure whose *status* field is used to report any error. Note that the settings of the context have no effect on the conversion (no rounding takes place, for example).

Returns *single*.

Possible errors are `DEC_Conversion_syntax` (the string does not have the syntax of a number), `DEC_Conversion_overflow` (the number has more than 15 significant digits or 3 significant digits of positive exponent), or `DEC_Conversion_underflow` (the number has 15 or fewer significant digits but the exponent is too negative). If either of these conditions is set, the *single* structure will have the value `NaN`, `Infinity`, or `0`, respectively, with the same sign as the converted number in the last two cases.

decSingleToString(single, string)

This function is used to convert a `decSingle` 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 extended specification.

The arguments are:

- single* (`decSingle *`) 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 23 characters long.

Returns *string*.

No error is possible from this function.

decSingleToEngString(single, string)

This function is used to convert a decSingle 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 extended specification.

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

decSingleFromNumber(single, number, context)

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

The arguments are:

- single* (decSingle *) 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 *round* field is used to control rounding as required.

Returns *single*.

An error will occur if the decNumber is outside the range supported by a decSingle. The possible errors are DEC_Overflow (if its adjusted exponent is greater than +999), or DEC_Underflow (if the adjusted exponent is less than -999). After overflow or underflow the result will have the same sign as the *number* and a value as though the **plus** operator had been used on the *number* using a context which enforces the constraints of single precision.¹⁶

decSingleToNumber(single, number)

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

The arguments are:

- single* (decSingle *) Pointer to the structure to be converted to a decNumber. The decSingle structure will not be altered.
- number* (decNumber *) Pointer to the result structure. It must have space for 15 digits of precision.

Returns *number*.

No error is possible from this function.

¹⁶ Note that subnormal numbers are a possible result when an Underflow condition is raised.

decDouble module

The `decDouble` module defines the data structure used for double precision decimal numbers, in a compact and machine-independent form. Double precision decimal numbers may have up to 33 decimal digits of precision and 4 decimal digits of exponent.¹⁷

The precise layout of a `decDouble` number is defined elsewhere; in this implementation it is represented as an array of unsigned bytes. There is therefore just one field in the `decDouble` structure:

bytes The *bytes* field represents the sixteen bytes of a double precision number using Densely Packed Decimal encoding for the coefficient.

The `decDouble` module uses private functions, located in the `decSingle` module, for coding and decoding Densely Packed Decimal data.

Definitions

The `decDouble.h` header file defines the `decDouble` 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 33, so that any declared `decNumber` will be the right size to contain any `decDouble` number.

The `decDouble.h` header file also contains:

- Constants defining aspects of `decDouble` numbers, including the maximum precision and (adjusted) exponent supported, the bias applied to the 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, exponent, and reserved bits).
- Definitions of the public functions in the `decDouble` module.

Functions

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

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.*

¹⁷ The exponent can be 5 digits, for subnormal numbers.

decDoubleFromString(double, string, context)

This function is used to convert a character string to decDouble format. It implements the **to-extended-number** conversion in the extended specification (that is, it accepts the special values $\pm\text{Inf}$, $\pm\text{Infinity}$, NaN, or sNaN, independent of case, and preserves -0).

The arguments are:

double (decDouble *) 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 base specification. The string will not be altered.

context (decContext *) Pointer to the context structure whose *status* field is used to report any error. Note that the settings of the context have no effect on the conversion. (No rounding takes place, for example).

Returns *double*.

Possible errors are DEC_Conversion_syntax (the string does not have the syntax of a number), DEC_Conversion_overflow (the number has more than 33 significant digits or 4 significant digits of positive exponent), or DEC_Conversion_underflow (the number has 33 or fewer significant digits but the exponent is too negative). If either of these conditions is set, the *double* structure will have the value NaN, Infinity, or 0, respectively, with the same sign as the converted number in the last two cases.

decDoubleToString(double, string)

This function is used to convert a decDouble 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 extended specification.

The arguments are:

double (decDouble *) 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 42 characters long.

Returns *string*.

No error is possible from this function.

decDoubleToEngString(double, string)

This function is used to convert a decDouble 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 extended specification.

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

decDoubleFromNumber(double, number, context)

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

The arguments are:

- double* (decDouble *) 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 *round* field is used to control rounding as required.

Returns *double*.

An error will occur if the decNumber is outside the range supported by a decDouble. The possible errors are DEC_Overflow (if its adjusted exponent is greater than +9999), or DEC_Underflow (if the adjusted exponent is less than -9999). After overflow or underflow the result will have the same sign as the *number* and a value as though the **plus** operator had been used on the *number* using a context which enforces the constraints of double precision.¹⁸

decDoubleToNumber(double, number)

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

The arguments are:

- double* (decDouble *) Pointer to the structure to be converted to a decNumber. The decDouble structure will not be altered.
- number* (decNumber *) Pointer to the result structure. It must have space for 33 digits of precision.

Returns *number*.

decDoubleFromSingle(double, single, context)

This function is used to convert a decSingle number to decDouble format.

The arguments are:

- double* (decDouble *) Pointer to the structure to be set from the decSingle.
- single* (decSingle *) Pointer to the input structure. The decSingle structure will not be altered.
- context* (decContext *) Pointer to a context structure. This is provided for symmetry with the decDoubleToSingle function; it is currently unused.

Returns *double*.

No error is possible from this function (a decSingle will always fit in a decDouble).

¹⁸ Note that subnormal numbers are a possible result when an Underflow condition is raised.

decDoubleToSingle(double, single, context)

This function is used to convert a decDouble number to decSingle format.

The arguments are:

- double* (decDouble *) Pointer to the input structure. The decDouble structure will not be altered.
- single* (decSingle *) Pointer to the structure to be set from the decDouble.
- context* (decContext *) Pointer to a context structure whose *status* field is used to report any error and whose *round* field is used to control rounding as required.

Returns *single*.

If the coefficient of the *double* has more than 15 digits then the value of the *double* rounded to 15 digits (using the rounding mode from the *context*) is used for the conversion.

An error will occur if the decDouble is outside the range supported by a decSingle. The possible errors are `DEC_Overflow` (if its adjusted exponent is greater than +999), or `DEC_Underflow` (if the adjusted exponent is less than -999). After overflow or underflow the result will have the same sign as the *double* and a value as though the **plus** operator had been used on the *double* using a context which enforces the constraints of single precision.¹⁹

¹⁹ Note that subnormal numbers are a possible result when an Underflow condition is raised.

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) plus20
```

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.

²⁰ 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* (`uByte *`) Pointer to an array of unsigned bytes which will receive the number.
- length* (`Int`) Contains the length of the byte array, in bytes.
- scale* (`Int *`) Pointer to an `Int` 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 $length \times 2 - 1$. 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 25) 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* (`uByte *`) Pointer to an array of unsigned bytes which contain the number to be converted.
- length* (`Int`) Contains the length of the byte array, in bytes.
- scale* (`Int *`) Pointer to an `Int` 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 $length \times 2 - 1$ 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 is a possible resulting number, but the exponent of a zero will always be 0 (that is, the scale is ignored if the *coefficient* is 0).

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 `ROUND_CEILING` and `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.

Index

// comments in C programs 3
.c (source) files 1
.h (header) files 1

A

addition 24, 26
adjusted exponent 13, 18
ANSI standard
 for REXX 1
 IEEE 754-1985 2
 IEEE 854-1987 1
 X3.274-1996 1
arguments
 corrupt 22
 modification of 22
 passed by reference 12
arithmetic
 decimal 1
 decNumber 24
auditing, of storage allocation 21

B

base specification 1
BCD
 See Binary Coded Decimal
Binary Coded Decimal 1, 2, 35
bits
 in decNumber 19
bytes

in decDouble 31
in decSingle 28

C

checking, of arguments 21, 22
code parameter
 DECALLOC 21
 DECHECK 21
 DECTrace 21
coefficient
 in decNumber 18
comparison 24
compound interest 5
constants
 naming convention 12
conversion
 decNumber 22
 double to number 33
 double to single 34
 double to string 32
 number to double 33
 number to packed 36
 number to single 30
 number to string 23
 packed to number 36
 single to double 33
 single to number 30
 single to string 29, 30
 string to double 32
 string to number 22
 string to single 29
copying numbers 26
corrupt arguments 22

D

- DEC_Conversion_overflow condition 9
- DEC_Division_impossible 24, 25
- DEC_Errors bits 6, 7, 14, 22
- DEC_Inexact condition 6, 15
- DEC_Lost_digits condition 15
- DEC_Overflow condition 26
- DEC_Rounded condition 6, 15
- DECALLOC code parameter 21
- DECBUFFER tuning parameter 20
- DECCHECK code parameter 21, 22
- decContext 1
 - digits 13
 - emax 13
 - extended 14
 - module 13
 - round 13
 - status 13
 - traps 14
- decContextDefault function 15
- decContextSetStatus function 16
- decContextSetStatusFromString function 16
- decContextStatusToString function 17
- decDouble 2
 - bytes 31
 - module 31
 - using 10
- decDoubleFromNumber function 33
- decDoubleFromSingle function 33
- decDoubleFromString function 32
- decDoubleToEngString function 32
- decDoubleToNumber function 33
- decDoubleToSingle function 34
- decDoubleToString function 32
- DECDPUN tuning parameter 19, 20
- decimal arithmetic 1
 - using 3
- DECNEG sign bit 21
- decNumber 1
 - bits 19
 - coefficient 18
 - digits 18
 - examples 20
 - exponent 18
 - lsu 19
 - module 18
 - msu 19
 - sign 18, 19
 - significand 18
 - size 18
 - special values 19
 - version 26
- decNumber.h 4
- decNumberAdd function 24
- decNumberCompare function 24
- decNumberCopy function 26
- decNumberDivide function 24
- decNumberDivideInteger function 24
- decNumberFromString function 22
- decNumberMinus function 24
- decNumberMultiply function 24
- decNumberPlus function 25
- decNumberPower function 25
- decNumberRemainder function 25
- decNumberRemainderNear function 25
- decNumberRescale function 25
- decNumberSubtract function 26
- decNumberToEngString function 23
- decNumberToInteger function 26
- decNumberToString function 23
- decNumberVersion function 26
- decNumberZero function 27
- DECNUMDIGITS constant 9, 10, 19
 - set by decPacked.h 35
 - set by decSingle.h 28, 31
- decPacked 2
 - module 35
 - using 11
- decPackedFromNumber function 36
- decPackedToNumber function 36
- decSingle 2
 - bytes 28
 - module 28
 - using 10
- decSingleFromNumber function 30
- decSingleFromString function 29
- decSingleToEngString function 30
- decSingleToNumber function 30
- decSingleToString function 29
- DECTrace code parameter 21
- digits
 - in decContext 13
 - in decNumber 18
- division 24, 25
- double floating-point 1
- dynamic storage 12, 20, 21
 - auditing 21

E

- emax
 - in decContext 13
- engineering notation 23, 30, 32
- error handling 14
 - active 7
 - passive 6
 - with signal 7
- example 3
 - active error handling 7
 - compound interest 5
 - decDouble numbers 9
 - decNumber 20
 - decPacked module 11
 - decSingle numbers 9
 - Example 1 4
 - Example 2 5
 - Example 3 6
 - Example 4 7
 - Example 5 9
 - Example 6 11
 - passive error handling 6
 - simple addition 4
 - special values 20
- exceptional conditions 14
- exponent
 - adjusted 13, 18
 - in decNumber 18
 - setting 25
- exponent maximum 13
- exponentiation 25
- extended
 - in decContext 14
- extended specification 1

F

- file
 - header 1
 - source 1
- Flag data type 14
- functions
 - arithmetic 24
 - conversions 22
 - naming convention 12
 - utilities 26

H

- header file 1
 - decContext 14
 - decDouble 31
 - decNumber 20
 - decPacked 35
 - decSingle 28

I

- IEEE standard 754-1985 2
- IEEE standard 854-1987 1
- Inexact condition 15
- infinite results 22
- infinity 18
- initializing numbers 22, 27
- Int data type 12, 13, 14
- integer rounding 26

L

- longjmp function 7
- Lost digits condition 15
- lsu, in decNumber 19

M

- maximum exponent 13
- minus operation 24
- modification of arguments 22
- module 12
 - decContext 13
 - decDouble 31
 - decNumber 18
 - decPacked 35
 - decSingle 28
 - naming convention 12
 - reentrancy 12
- monadic operators 24
- msu, in decNumber 19

multiplication 24

N

naming convention

constants 12

functions 12

modules 12

NaN 18

quiet 18

results 22

signaling 18

negation 24

nibble 35

P

Packed Decimal 1, 2, 35

parameters

code 21

tuning 20

plus operation 25

power operator 25

powers of 10 array 21

prefix

minus 24

plus 25

printf function 4

Q

quiet NaN 18

R

reentrant modules 12

references, to arguments 12

remainder 25

rescaling 25

results

rounding of 15

undefined 22

round

See also rounding

in decContext 13

round-to-integer operation 26

ROUND_CEILING 13

ROUND_DOWN 13

ROUND_FLOOR 13

ROUND_HALF_DOWN 13

ROUND_HALF_EVEN 13

ROUND_HALF_UP 13

ROUND_UP 13

Rounded condition 15

rounding

detection of 15

enumeration 13

to integer 26

using decNumberPlus 25

S

scale 2, 35

setting 25

scientific notation 23, 29, 32

setjmp function 8

SIGFPE

implementation issues 3

signal 7, 8, 14

sign

DECNEG bit 21

in decNumber 18, 19

signal

function 8

handler 7

signaling NaN 18

significand

See also coefficient

in decNumber 18

single floating-point 1

size, of decNumber 18

source file 1

decContext 15

decDouble 31

decNumber 22

decPacked 36

decSingle 29

special values 14, 18, 20

in decNumber 19

specification

- base 1
- extended 1
- speed of operations 20
- Standard Decimal Arithmetic 1
- status
 - in decContext 13
- stdio.h 4
- storage allocation 21
 - auditing 21
- subnormal values 18

T

- traps 14
 - in decContext 14
- tuning parameter 12
 - DECBUFFER 20
 - DECDPUN 20

U

- uInt data type 13, 14
- undefined results 22
- unit
 - in decNumber 19
 - size of 19, 20
- User's Guide 3
- utilities
 - decNumber 26

V

- value of a number 18
- version, of decNumber 26

Z

- zero decNumber 19, 20
- zeroing numbers 27