The decNumber C library

Version 3.50 – 4th June 2007

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. Both arbitrary-precision and fixed-size representations are supported.

The arbitrary-precision 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 BCD (4-bit Binary Coded Decimal) and three fixed-size formats of decimal floating-point. called *decFloats* (see page 48).

The three fixed-size formats are also supported by three modules called *decFloats* (see page 48), which have an extensive set of functions that work directly from the formats and provide arithmetical, logical, and shifting operations, together with conversions to binary integers, Packed BCD, and 8-bit BCD. Most of the functions defined in the proposed IEEE 754 revision ("754r")[2] are included, together with other functions outside the scope of that standard but essential for a decimal-only language implementation.

Library structure

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 an arbitrary-precision decimal number representation designed for efficient computation in software and implements the arithmetic and logical operations, together with a number of 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 con-

[1] See `http://www2.hursley.ibm.com/decimal/#arithmetic` for details.

[2] See `http://grouper.ieee.org/groups/754/`

trols the handling of exceptional conditions (corresponding to the flags and trap enablers in a hardware floating-point implementation).

The decNumber representation is variable-length and machine-dependent (for example, it contains integers which may be big-endian or little-endian).

In addition to the arbitrary-precision decNumber format, three fixed-size compact storage formats are provided for conversions and interchange.[3] These formats are endian-dependent but otherwise are machine-independent:

decimal32	a 32-bit decimal floating-point representation which provides 7 decimal digits of precision in a compressed format
decimal64	a 64-bit decimal floating-point representation which provides 16 decimal digits of precision in a compressed format
decimal128	a 128-bit decimal floating-point representation which provides 34 decimal digits of precision in a compressed format.

A fourth, machine-independent, Binary Coded Decimal (BCD) format is also provided:

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.

The decimal32, decimal64, and decimal128 formats are also supported directly by three modules which can be used stand-alone (that is, they have no dependency on the decNumber module). These are:

decSingle	a module that provides the functions for the decimal32 format; this format is the 754r *storage format* and so provides utilities and conversions only
decDouble	a module that provides the functions for the decimal64 format; this format is a 754r *basic format* and so a full set of arithmetic and other functions is included
decQuad	a module that provides the functions for the decimal128 format; this format is a 754r *basic format*; it contains the same set of functions as decDouble.[4]

These modules use the same context mechanism (decContext) as decNumber and so can be used together with the decNumber module when required in order to use the mathematical functions in that module or to use its arbitrary-precision capability. Examples are included in the User's Guide (see page 4).

[3] See `http://www2.hursley.ibm.com/decimal/decbits.html` for details of the formats.

[4] Except for two which convert to or from a wider format.

Relevant standards

It is intended that, where applicable, functions in the decNumber package follow the requirements of:

- the floating-point decimal arithmetic defined in ANSI X3.274-1996[5] (including errata through 2001)
- the decimal arithmetic requirements of IEEE 854-1987,[6] as modified by the current IEEE 754r revision work,[7] except that:
  1. The values returned on 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 $\pm 10^{1,000,000,000}$).

Please advise the author of any discrepancies with these standards.

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

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

[7] See `http://grouper.ieee.org/groups/754/` ; this revision is in ballot, but is still subject to change.

User’s Guide

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

Conversions to and from the decNumber format are fast; they are usually faster than all but 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.

However, when arbitrary-precision calculation is not required (that is, up to 34 digits of precision is all that is required) it is even more efficient to use one of the *decFloats* modules (see page 48) for arithmetic and other operations. The decFloats modules work directly from the decimal-encoded compressed formats and avoid the need for conversions to and from the decNumber format. Tables comparing the performance of the decFloats modules with decNumber can be found in Appendix A (see page 70).

The remainder of this section illustrates the use of the coded formats and the decFloats modules 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 (`/* ... */`). Note that the decNumber header files use only block comments so do not require conversion.

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 66) 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 8).

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 provides 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).[8]

[8] 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 26) 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 26) 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=%ld)\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.

Example 7 – Using the decQuad module

This example reworks Example 1, but using the decQuad module for all conversions and the arithmetic.

```
 1. // example7.c -- using decQuad to add two numbers together
 2.
 3. #include "decQuad.h"                 // decQuad library
 4. #include <stdio.h>                   // for printf
 5.
 6. int main(int argc, char *argv[]) {
 7.   decQuad a, b;                      // working decQuads
 8.   decContext set;                    // working context
 9.   char string[DECQUAD String];       // number->string buffer
10.
11.   if (argc<3) {                      // not enough words
12.     printf("Please supply two numbers to add.\.n");
13.     return 1;
14.     }
15.   decContextDefault(&set, DEC INIT DECQUAD); // initialize
16.
17.   decQuadFromString(&a, argv[1], &set);
18.   decQuadFromString(&b, argv[2], &set);
19.   decQuadAdd(&a, &a, &b, &set);      // a=a+b
20.   decQuadToString(&a, string);
21.
22.   printf("%s + %s => %s\n", argv[1], argv[2], string);
23.   return 0;
24.   } // main
```

This example is a complete, runnable program. Like Example 1, it takes two argument words, converts them to a decimal format (in this case decQuad, the 34-digit format), adds them, and converts the result back to a string for display.

Line 3 includes the decQuad header file. This in turn includes the other necessary header, decContext. The context variable `set` is used to set the rounding mode for the conversions from string and for the add, and its status field is used to report any errors (not checked in this example). No other field in the context is used.

To compile and run this, only the files `example7.c`, `decContext.c`, and `decQuad.c` are needed.

To use the 16-digit format instead of the 34-digit format, change `decQuad` to `decDouble` and `QUAD` to `DOUBLE` in the example. Note that in this case the file `decQuad.c` is still needed, as decDouble requires decQuad.

Example 8 – Using decQuad with decNumber

This example shows how the decNumber and decQuad modules can be mixed, in this case to raise one number to the power of another. (In this case, the use of the decQuad module could be avoided – this is just to demonstrate how to use the two modules together.)

```
// example8.c -- using decQuad with the decNumber module

#include "decQuad.h"                // decQuad library
#include "decimal128.h"             // interface to decNumber
#include <stdio.h>                  // for printf

int main(int argc, char *argv[]) {
  decQuad a;                        // working decQuad
  decNumber numa, numb;             // working decNumbers
  decContext set;                   // working context
  char string[DECQUAD String];      // number->string buffer

  if (argc<3) {                     // not enough words
    printf("Please supply two numbers for power(2*a, b).\n");
    return 1;
    }
  decContextDefault(&set, DEC INIT DECQUAD); // initialize

  decQuadFromString(&a, argv[1], &set);      // get a
  decQuadAdd(&a, &a, &a, &set);              // double a
  decQuadToNumber(&a, &numa);                // convert to decNumber
  decNumberFromString(&numb, argv[2], &set);
  decNumberPower(&numa, &numa, &numb, &set); // numa=numa**numb
  decQuadFromNumber(&a, &numa, &set);        // back via a Quad
  decQuadToString(&a, string);               // ..

  printf("power(2*%s, %s) => %s\n", argv[1], argv[2], string);
  return 0;
  } // main
```

Here, the decimal128 module is used as a "proxy" between the decNumber and decQuad formats. The decimal128 and decQuad structures are identical (except in name) so pointers to the structures can safely be cast from one to the other. The `decQuadToNumber` and `decQuadFromNumber` functions are in fact macros which cast the data pointer and then use the `decimal128ToNumber` or `decimal128FromNumber` function to effect the conversion. Using a proxy in this way avoids any dependencies between decQuad and decNumber.

Note that the same decContext structure (`set`) is used for both decQuad and decNumber function calls. decQuad uses only the *round* and *status* fields, but decNumber also needs the other fields. All the fields are initialized by the call to `decContextDefault`.

The inclusion of `decimal128.h` also sets up the `DECNUMDIGITS` required and includes `decNumber.h`. The decimal128 module requires decimal64 (for shared code and tables), so the full list of files to compile for this example is: `example8.c`, `decContext.c`, `decQuad.c`, `decNumber.c`, `decimal128.c`, and `decimal64.c`.

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.
- Only arbitrary-precision calculations might allocate memory. Up to some maximum precision (chosen by a tuning parameter in the `decNumberLocal.h` file), even these calculations do not require allocated memory, except for rounded input arguments and some mathematical functions. 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`”, and 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 `decNumber` module uses all of these fields for full control of arbitrary-precision arithmetic; the *decFloats* modules (decQuad, *etc.*) are fixed-size and fixed-format and use only the *round* and *status* fields.

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.[9] *emin* is usually set to –*emax* or to –(*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.

[9] 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.
`DEC_ROUND_05UP`	The same as `DEC_ROUND_UP`, except that rounding up only occurs if the digit to be rounded up is 0 or 5.
`DEC_ROUND_DEFAULT`	The same as `DEC_ROUND_HALF_EVEN`.

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 and `decContextSetStatus` is called (which happens automatically at the end of any operation which sets a status bit).

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 controls 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 a maximum of *emax* and a minimum of *emin* (for example, the exponent of a zero result will be clamped to be in this range). When 1, a result exponent has the same minimum but is limited to a maximum of *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.

Also when 1, *clamp* limits the length of NaN payloads to *digits*–1 (rather than *digits*) when constructing a NaN by conversion from a string.

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 (see page 67) 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 `decClass` enumeration (and corresponding strings) which is used to classify numbers with the decNumberClass function (see page 37) or the equivalent functions in decQuad, *etc.*
- The exceptional condition flags, used in the *status* and *traps* fields. The flags used can be modified by the `DECEXTFLAG` tuning parameter (see page 66).
- 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 an extreme 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.

Note that of the above only the `DEC_Inexact` flag is set by the *decFloats* modules. The other informational flags are only set by the decNumber module.

Functions

The `decContext.c` source file contains the public functions defined in the header file, as follows. In all these functions, only status bits (*etc.*) that are defined in the `decContext.h` header file should be used.[10]

decContextClearStatus(context, status)

This function is used to clear (set to zero) one or more status bits in the *status* field of a decContext.

The arguments are:

context	(`decContext *`) Pointer to the structure whose status is to be updated.
status	(`uint32_t`) Any 1 (set) bit in this argument will cause the corresponding bit to be cleared in the context status field.

Returns *context*.

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).

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

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` or `DEC_INIT_DECSINGLE`, defaults for a *decimal32* number using IEEE 754r 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` or `DEC_INIT_DECDOUBLE`, defaults for a *decimal64* number using IEEE 754r 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` or `DEC_INIT_DECQUAD`, defaults for a *decimal128* number using IEEE 754r 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*.

decContextGetRounding(context)

This function is used to return the *round* (rounding mode) field of a decContext.

The argument is:

context	(`decContext *`) Pointer to the structure whose rounding mode is to be returned.

Returns the `enum rounding` rounding mode.

decContextGetStatus(context)

This function is used to return the *status* field of a decContext.

The argument is:

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

Returns the `uint32_t` status field.

decContextRestoreStatus(context, status, mask)

This function is used to restore one or more status bits in the *status* field of a decContext from a saved status field.

The arguments are:

context	(`decContext *`) Pointer to the structure whose status is to be updated.
status	(`uint32_t`) A saved status field (as saved by `decContextSaveStatus` or retrieved by `decContextGetStatus`).

mask	(`uint32_t`) Any 1 (set) bit in this argument will cause the corresponding bit to be restored (set to 0 or 1, taken from the corresponding bit in *status*) in the context status field.

Returns *context*.

Note that setting a bit using this function does not cause a trap (use the decContextSetStatus function can be used to raise a trap, if desired).

decContextSaveStatus(context, mask)

This function is used to save one or more status bits from the *status* field of a decContext.

The arguments are:

context	(`decContext *`) Pointer to the structure whose status is to be saved.
mask	(`uint32_t`) Any 1 (set) bit in this argument will cause the corresponding bit to be saved from the context status field.

Returns the `uint32_t` which is the logical And of the context status field and the *mask*.

decContextSetRounding(context, rounding)

This function is used to set the rounding mode in the *round* field of a decContext.

The arguments are:

context	(`decContext *`) Pointer to the structure whose rounding mode is to be set.
rounding	(`enum rounding`) The rounding mode to be copied to the context *round* field.

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.

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.

decContextSetStatusFromStringQuiet(context, string)

This function is identical to decContextSetStatusFromString except that the context *traps* field is ignored (*i.e.*, no trap is raised).

decContextSetStatusQuiet(context, status)

This function is identical to decContextSetStatus except that the context *traps* field is ignored (*i.e.*, no trap is raised).

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.

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.

decContextTestSavedStatus(status, mask)

This function is used to test one or more status bits in a saved status field.

The arguments are:

status	(`uint32_t`) A saved status field (as saved by `decContextSaveStatus` or retrieved by `decContextGetStatus`).
mask	(`uint32_t`) Any 1 (set) bit in this argument will cause the corresponding bit in *status* to be included in the test.

Returns the `uint32_t` which is the logical And of *status* and *mask*.

decContextTestStatus(context, mask)

This function is used to test one or more status bits in a context.

The arguments are:

context	(`decContext *`) Pointer to the structure whose status is to be tested.
mask	(`uint32_t`) Any 1 (set) bit in this argument will cause the corresponding bit in context *status* field to be included in the test.

Returns the `uint32_t` which is the logical And of the context *status* field and *mask*.

decContextZeroStatus(context)

This function is used to clear (set to zero) all the status bits in the *status* field of a decContext.

The argument is:

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

Returns *context*.

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, set by the compile-time definition `DECDPUN` (see page 68). 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 6). 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.[11]

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}`

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

and `-∞` (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 26), which in turn alters the performance and other characteristics of the library. Further details are given in the tuning section (see page 68).

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

- The `decClass` enumeration (and corresponding strings) which is used to classify decNumbers with the decNumberClass function (see page 37).
- 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 and logical 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 17). 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 19) 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 69) is 1, which enables full operand checking. 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. Additional Utility functions (see page 37) are included in the package for conversions to and from BCD strings and binary integers.

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` and the adjusted exponent is in the range specified by `context.emin` and `context.emax`. If there are more than `context.digits` digits in the string, or the exponent is out of range, the value will be rounded as necessary 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 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 and logical functions

The arithmetic and logical 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*. The decNumberFMA function (see page 31) also takes a third numeric operand *fhs* (`decNumber *`), a pointer to the structure which is the "far hand side" operand for the operation.

Some functions, such as decNumberExp, are 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 document.

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*.

decNumberAnd(number, lhs, rhs, context)

The *number* is set to the result of the digit-wise logical **and** of *lhs* and *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.

decNumberCompareSignal(number, lhs, rhs, context)

This function compares two numbers numerically. It is identical to decNumberCompare except that all NaNs (including quiet NaNs) signal.

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.

decNumberCompareTotalMag(number, lhs, rhs, context)

This function compares the magnitude of two numbers using the IEEE 754r proposed ordering. It is identical to decNumberCompareTotal except that the signs of the operands are ignored and taken to be 0 (non-negative).

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.

decNumberFMA(number, lhs, rhs, fhs, context)

The *number* is set to the result of multiplying the *lhs* by the *rhs* and then adding *fhs* to that intermediate result. It is equivalent to a multiplication followed by an addition except that the intermediate result is not rounded and will not cause overflow or underflow. That is, only the final result is rounded and checked.

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

decNumberInvert(number, rhs, context)

The *number* is set to the result of the digit-wise inversion of *rhs* (A 0 digit becomes 1 and vice versa.)

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.

decNumberLogB(number, rhs, context)

The *number* is set to the adjusted exponent of *rhs*, according to the rules for the "logB" operation of the IEEE 754r proposal. This returns the exponent of *rhs* as though its decimal point had been moved to follow the first digit while keeping the same value. The result is not limited by *emin* or *emax*.

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.

decNumberMaxMag(number, lhs, rhs, context)

This function compares the magnitude of two numbers numerically and sets *number* to the larger. It is identical to decNumberMax except that the signs of the operands are ignored and taken to be 0 (non-negative).

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.

decNumberMinMag(number, lhs, rhs, context)

This function compares the magnitude of two numbers numerically and sets *number* to the smaller. It is identical to decNumberMin except that the signs of the operands are ignored and taken to be 0 (non-negative).

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*.

decNumberNextMinus(number, rhs, context)

The *number* is set to the closest value to *rhs* in the direction of –Infinity. This is computed as though by subtracting an infinitesimal amount from *rhs* using `DEC_ROUND_FLOOR`, except that no flags are set unless *rhs* is an sNaN.

This function is a generalization of the IEEE 754 *nextDown* operation.

decNumberNextPlus(number, rhs, context)

The *number* is set to the closest value to *rhs* in the direction of +Infinity. This is computed as though by adding an infinitesimal amount to *rhs* using `DEC_ROUND_CEILING`, except that no flags are set unless *rhs* is an sNaN.

This function is a generalization of the IEEE 754 *nextUp* operation.

decNumberNextToward(number, lhs, rhs, context)

The *number* is set to the closest value to *lhs* in the direction of *rhs*. This is computed as though by adding or subtracting an infinitesimal amount to *lhs* using `DEC_ROUND_CEILING` or `DEC_ROUND_FLOOR`, depending on whether *rhs* is larger or smaller than *lhs*. If *rhs* is numerically equal to *lhs* then the result is a copy of *lhs* with the *sign* taken from *rhs*. Flags are set as usual for an addition or subtraction except that if the operands are equal or the result is normal (finite, non-zero, and not subnormal) no flags are set.

This function is a generalization of the IEEE 754 *nextAfter* operation.

decNumberOr(number, lhs, rhs, context)

The *number* is set to the result of the digit-wise logical **inclusive or** of *lhs* and *rhs*.

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.[12]

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 35) function may also be used for this purpose, but requires the exponent to be given as a decimal number.

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)

Integer-divides *lhs* by *rhs* and places the remainder from the division in *number*.

That is, if the same *lhs*, *rhs*, and *context* arguments were given to the

[12] 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.

decNumberDivideInteger and decNumberRemainder functions, resulting in `i` and `r` respectively, then the identity

lhs = (`i` × *rhs*) + `r`

holds.

Note that, as for decNumberDivideInteger, it must be possible to express the integer part of the result (`i`) as an integer. That is, it must have no more digits than `context.digits`. If it does have more 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 34) 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*.

decNumberRotate(number, lhs, rhs, context)

This function is used to rotate the digits in the coefficient of a number as though its coefficient had the length given by `context.digits` and its most-significant digit were connected to its least-significant digit.

The *number* is set to a copy of *lhs* with the digits of its coefficient rotated to the left (if *rhs* is positive) or to the right (if *rhs* is negative) without adjusting the exponent or the sign. If *lhs* has fewer digits than `context.digits` the coefficient is padded with zeros on the left before the rotate. Any insignificant leading zeros in the result are removed, as usual.

rhs is the count of digits to rotate; it must be an integer (that is, it must have an *exponent* of 0) and must be in the range `-context.digits` through +`context.digits`.

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.

decNumberScaleB(number, lhs, rhs, context)

This function is used to adjust (scale) the exponent of a number, using the rules of the "scaleB" operation in the IEEE 754r proposal. The *number* is set to the result of multiplying *lhs* by ten raised to the power of *rhs*. *rhs* must be an integer (that is, it must have an *exponent* of 0) and it must also be in the range $-n$ through $+n$, where n is $2 \times$ (`context.emax+context.digits`).

decNumberShift(number, lhs, rhs, context)

This function is used to shift the digits in the coefficient of a number. The *number* is set to a copy of *lhs* with the digits of its coefficient shifted to the left (if *rhs* is positive) or to the right (if *rhs* is negative) without adjusting the exponent or the sign. The coefficient is padded with zeros on the left or right, as necessary. Any leading zeros in the result are ignored, as usual.

rhs is the count of digits to shift; it must be an integer (that is, it must have an *exponent* of 0) and must be in the range `-context.digits` through +`context.digits`.

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*.

decNumberToIntegralExact(number, rhs, context)

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

The `Inexact` flag is set if the result is numerically different from *rhs*. Other than that, no flags are set (unless the operand is a signaling NaN). The result may have a positive exponent.

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 flags, not even `Inexact`, are set (unless the operand is a signaling NaN). The result may have a positive exponent.

decNumberXor(number, lhs, rhs, context)

The *number* is set to the result of the digit-wise logical **exclusive or** of *lhs* and *rhs*.

Utility functions

The utility functions include copying, trimming, test, and initializing functions, along with specialized conversions and a function for determining the version of the decNumber package.

decNumberClass(number, context)

This function is used to determine the class of a decNumber. The arguments are:

number	(`decNumber *`) Pointer to the decNumber to be classified.
context	(`decContext *`) Pointer to the context (the value of *emin* is used to determine if a finite number is normal or subnormal).

Returns an `enum decClass` (defined in `decNumber.h`), which can be converted to a character string using decNumberClassToString. No error is possible from this function.

decNumberClassToString(number, context)

This function is used to convert a decClass enumeration to a string. The argument is:

class	(`enum decClass`) The enumeration to be converted.

Returns a string (`const char *`) which points to one of the constant strings `"sNaN"`, `"NaN"`, `"-Infinity"`, `"-Normal"`, `"-Subnormal"`, `"-Zero"`, `"+Zero"`, `"+Subnormal"`, `"+Normal"`, `"+Infinity"`, or `"Invalid"`. No error is possible from this function.

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.

decNumberCopyAbs(number, source)

This function is used to copy the absolute value of the content of one decNumber structure to another. It is identical to decNumberCopy except that the *sign* of the result is always 0. This is equivalent to the quiet *abs* function described in IEEE 754r.

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

decNumberCopyNegate(number, source)

This function is used to copy the value of the content of one decNumber structure to another while inverting its sign. It is identical to decNumberCopy except that the *sign* of the result is the inverse of that in *source*. This is equivalent to the quiet *negate* function described in IEEE 754r.

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

decNumberCopySign(number, source, pattern)

This function is used to copy the value of the content of one decNumber structure to another and changing its sign to that of a third. It is identical to decNumberCopy except that the *sign* of the result is taken from the third argument instead of from *source*. This is equivalent to the quiet *copysign* function described in IEEE 754r.

The first two arguments are as for decNumberCopy. The third is:

pattern	(`decNumber *`) Pointer to the structure which provides the sign.

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

decNumberFromInt32(number, i)

This function is used to convert a signed (two's complement) 32-bit binary integer to a decNumber. The arguments are:

number	(`decNumber *`) Pointer to the structure that will received the converted integer. This must have space for the digits needed to represent the value of *i*, which may need up to ten digits.
i	(`int32_t`) The integer to be converted.

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

decNumberFromUInt32(number, u)

This function is used to convert an unsigned 32-bit binary integer to a decNumber. The arguments are:

number	(`decNumber *`) Pointer to the structure that will received the converted integer. This must have space for the digits needed to represent the value of *u*, which may need up to ten digits.
u	(`uint32_t`) The integer to be converted.

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

decNumberGetBCD(number, bcd)

This function is used to convert the coefficient of a decNumber to Binary Coded Decimal, one digit (value 0–9) per byte. The arguments are:

number	(`decNumber *`) Pointer to the structure containing the coefficient to be converted.
bcd	(`uint8_t *`) Pointer to the byte array which will receive the converted coefficient; the most significant digit of the coefficient will be placed in `bcd[0]`. The first `number->digits` elements of *bcd* will have their values set; no other elements are affected.

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

decNumberIsCanonical(number)

This function is used to test whether the encoding of a decNumber is canonical.

The argument is:

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

Returns 1 (true) always, because decNumbers always have canonical encodings (the function is provided for compatibility with the the IEEE 754r operation *isCanonical*). This function may be implemented as a macro; no error is possible.

decNumberIsFinite(number)

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

The argument is:

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

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

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 (that is, it is a finite number or a NaN). 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, less than zero, or a NaN with a *sign* of 1).

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.

decNumberIsNormal(number)

This function is used to test whether a number is normal (that is, finite, non-zero, and not subnormal).

The arguments are:

number	(`decNumber *`) Pointer to the structure whose value is to be tested.
context	(`decContext *`) Pointer to the context (the value of *emin* is used to determine if a finite number is normal or subnormal).

Returns 1 (true) if the number is normal, 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.

decNumberIsSpecial(number)

This function is used to test whether a number has a special value (Infinity or NaN); it is the inversion of decNumberIsFinite (see page 39).

The argument is:

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

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

decNumberIsSubnormal(number)

This function is used to test whether a number is subnormal (that is, finite, non-zero, and magnitude less than 10^{emin}).

The arguments are:

number	(`decNumber *`) Pointer to the structure whose value is to be tested.
context	(`decContext *`) Pointer to the context (the value of *emin* is used to determine if a finite number is normal or subnormal).

Returns 1 (true) if the number is subnormal, 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.

decNumberRadix()

This function returns the radix (number base) used by the decNumber package. This always returns 10. This function may be implemented as a macro; no error is possible.

decNumberReduce(number, rhs, context)

This function has the same effect as decNumberPlus except that the final result is set to its simplest (shortest) form without changing its value. 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.

The decNumberTrim function (see page 42) can be used to remove only fractional trailing zeros.

This function was previously called *decNumberNormalize* (and is still available under that name for compatibility).

decNumberSetBCD(number, bcd, n)

This function is used to set the coefficient of a decNumber from a Binary Coded Decimal array which has one digit (value 0–9) per byte. The arguments are:

number	(`decNumber *`) Pointer to the structure whose coefficient is to be set.
bcd	(`uint8_t *`) Pointer to the byte array which provides the coefficient; the most significant digit of the coefficient is at `bcd[0]` and the least significant is at `bcd[n-1]`.
n	(`uint32_t *`) Count of the BCD digits to be converted.

number must have space for at least *n* digits. If *number* is a NaN, or is Infinite, or is to become a zero, *n* must be 1 and `bcd[0]` must be zero.

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

decNumberToInt32(number, context)

This function is used to convert a decNumber to a signed (two's complement) 32-bit binary integer. The arguments are:

number	(`decNumber *`) Pointer to the structure that will have its value converted.
context	(`decContext *`) Pointer to the context (used only for reporting an error).

The `DEC_Invalid_operation` condition is raised if *number* does not have an exponent of 0, or if it is a NaN or Infinity, or if it is out-of-range (cannot be represented). In this case the result is 0. Note that a –0 is not out of range (it is numerically equal to zero and will be converted without raising the condition).

Returns the signed integer (`int32_t`).

decNumberToUInt32(number, context)

This function is used to convert a decNumber to an unsigned 32-bit binary integer. The arguments are:

number	(`decNumber *`) Pointer to the structure that will have its value converted.
context	(`decContext *`) Pointer to the context (used only for reporting an error).

The `DEC_Invalid_operation` condition is raised if *number* does not have an exponent of 0, or if it is a NaN or Infinity, or if it is out-of-range (cannot be represented). In this case the result is 0. Note that a –0 is not out of range (but all values less than zero are).

Returns the unsigned integer (`uint32_t`).

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 decNumberReduce function (see page 41) can be used to remove all trailing zeros.

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 3.40
```

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 conversion 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`

These modules provide the interface between the compressed numbers and the decNumber internal format (and also provide string conversions). The *decFloats* modules (see page 48) provide arithmetic and other functions which work on data in the same formats directly. Example 7 and Example 8 in the User's Guide (see page 14) show how to work with data in the formats with or without using the decNumber module.

Apart from the different lengths and ranges of the numbers, these three modules are identical, so this section just describes the decimal64 module. The definitions and functions for the other two formats are the same, 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.[13]

The storage of a number in the bytes array is assumed to follow the byte ordering ("endianness") of the computing platform (if big-endian, then `bytes[0]` contains the sign bit of the format). The code in these modules requires that the `DECLITEND` tuning parameter (see page 66) be set to match the endianness of the platform.

Note that the equivalent structures in the decFloats modules are identical except for their names. It is therefore safe to cast pointers between them if they are the same size (for example between decimal64 and decDouble). This means that these modules can be used as *proxies* between the decNumber module and the decFloats modules.

The decimal64 module includes private functions for coding and decoding Densely Packed Decimal data; these functions are shared by the other compressed format modules. Hence, when using any of these three then `decimal64.c` must be compiled too.

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

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 exponent, the length of the number in bytes, and the maximum number of characters in the string form of the number (including terminator)
- 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 three of of the tables in the header file are used). 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 decNumbers, and some utilities.

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 which controls the conversion, as for the decNumberFromString function (see page 28) except that the precision and exponent range are fixed for each format (the values of *emax*, *emin*, and *digits* are ignored).

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 *emax* for the format), or `DEC_Underflow` (the adjusted exponent of the number is negative and is less than *emin* for the format and the conversion is not exact). If one of these conditions is set, the *decimal64* structure will have the value `NaN`, ±`Infinity` or the largest possible finite number, 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 45), 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.

decimal64Canonical(decimal64, source)

This function is used to ensure that a decimal64 number is encoded with the canonical form. That is, all declets use the preferred 1000 encodings and an infinity has a coefficient of zero.

The arguments are:

decimal64	`(decimal64 *)` Pointer to the structure to receive a copy of *source*, with canonical encoding.
source	`(decimal64 *)` Pointer to the structure to be converted to a canonical encoding.

Returns *decimal64*; no error is possible from this function.

decimal64IsCanonical(decimal64)

This function is used to test whether a decimal64 number is encoded with the canonical form. That is, that all declets use the preferred 1000 encodings and an infinity has a coefficient of zero.

The argument is:

decimal64	`(decimal64 *)` Pointer to the structure to be tested.

Returns an unsigned integer `(uint32_t *)` which is 1 if *decimal64* has canonical encoding, or 0 otherwise. No error is possible from this function.

decFloats modules

The *decFloats* modules are decSingle, decDouble, and decQuad. These are based on the decimal types in the proposed IEEE 754 Floating-Point Standard revision.

In contrast to the arbitrary-precision decNumber module, these modules work directly from the decimal-encoded formats designed by the IEEE 754 committee, which are also now implemented in IBM System z (Z9) and IBM System p (Power6) processors. Conversions to and from the decNumber internal format are no needed (typically the the numbers are represented internally in “unpacked” BCD or in a base of some other power of ten), and no memory allocation is necessary, so these modules are much faster than using decNumber for implementing the types.

Like the decNumber module, the decFloats modules

- need only 32-bit integer support; 64-bit integers are not required and binary floating-point is not used
- support both big-endian and little-endian encodings and platforms
- support both ASCII/UTF8 and EBCDIC strings
- use only ANSI C.

The modules should therefore be usable on any platform with an ANSI C compiler that supports 32-bit integers.

The decFloats modules define the data structures and a large set of functions for working directly with the same compressed formats as decimal32, decimal64, and decimal128. The names are different to allow them to be used stand-alone or with the decNumber module, as illustrated in Examples 7 and 8 in the User’s Guide (see page 14).

These three modules all share many of the same functions (working on the different sizes of the formats). The decQuad module has all the same functions as decDouble except for two functions which would convert to or from a wider format. The decSingle module is a limited (“storage”) format which has a only a few conversion and miscellaneous functions; it is intended that any computation be carried out in a wider format.

The remainder of this section therefore describes only the decDouble format – in the list of functions, assume that there is a corresponding decQuad format function unless stated and assume there is *not* a corresponding decSingle format function unless stated.

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

bytes The *bytes* field represents the eight bytes of a decDouble number, using Densely Packed Decimal encoding for the coefficient.[14]

The storage of a number in the bytes array is assumed to follow the byte ordering (“endianness”) of the computing platform (if big-endian, then `bytes[0]` contains the sign

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

bit of the format). The code in these modules requires that the `DECLITEND` tuning parameter (see page 66) be set to match the endianness of the platform.

The decSingle and decDouble modules both require that the next wider format be included in a program compilation (so that conversion to and from that wider format can be effected), hence the decQuad module is always needed.[15] It, therefore, contains the constant lookup tables from the the `decDPD.h` header file which are shared by all three modules. These tables are automatically generated and should not need altering.

Most of the code for these modules is included from the shared source files `decCommon.c` and `decBasic.c`. The former contains the functions available in all three modules[16] and the latter the functions available only in decDouble and decQuad.

Definitions

The `decDouble.h` header file defines the decDouble data structure described above. It includes the `decContext.h` and and `decQuad.h` header files, which are both required for use.[17] If more than one of the three decFloats formats are used in a program, it is only necessary to include the smaller or smallest.

The `decDouble.h` header file also contains:

- Constants defining aspects of decDouble numbers, including the maximum precision, the minimum and maximum (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)
- Definitions of the public functions in the decDouble module
- Macros defining conversions to and from the decNumber format. These are macros in order to avoid a compile-time dependency on the decNumber module; they use decimal64 as a proxy, and their usage is shown in Example 8 in the User's Guide (see page 15).

[15] This requirement is different from the decimal32, decimal64, and decimal128 modules because they can convert to wider or narrower formats using the decNumber format as an intermediate step.

[16] Except that the widening and narrowing functions are not used by decQuad.

[17] The `decSingle.h` header file also includes `decDouble.h`, but the `decQuad.h` header file only includes `decContext.h`.

Functions

The `decDouble.c` source file contains the public functions defined in the header file. These comprise conversions to and from strings and other formats, arithmetic and logical operations, and utilities.

The functions are described briefly, below. More details of the operation of each function can be found in the description of the corresponding function in the decNumber module and details of the underlying model and arithmetic can be found in the *General Decimal Arithmetic Specification*.[18]

In the descriptions below, many parameters are defined as one of the following:

x, y, z	(`const decDouble *`) decimal input arguments to a function
r	(`decDouble *`) a decimal result argument to a function (which may be the same as an input argument); unless stated otherwise this is also the return value from the function, and the result will be canonical
set	(`decContext *`) the context for a function. Only two fields of the context structure are used: *round* (the rounding mode) and *status* (the bits in which are used to indicate any error, *etc.*).

Note that the *trap* field in the context is not used; the decDouble functions do not check for traps after every operation to avoid the overhead that would incur. The `decContextSetStatus` function (see page 22) can be used to explicitly test status to trap.

Note also that the only informational flag set by decNumber is `DEC_Inexact`; the others are never set by the decFloats module in order to improve performance and also to avoid the need for passing a context to many functions.[19]

In the following list, every function has corresponding decQuad format function (for example, `decQuadAbs(r, x, set)`) unless stated, and does *not* have a corresponding decSingle format function unless stated.

decDoubleAbs(r, x, set)

Returns the absolute value of *x*. This has the same effect as decDoublePlus unless *x* is negative, in which case it has the same effect as decDoubleMinus. The effect is also the same as decDoubleCopyAbs except that NaNs are handled normally (the sign of a NaN is not affected, and an sNaN will set `DEC_Invalid_operation`) and the result will be canonical.

decDoubleAdd(r, x, y, set)

Adds *x* and *y* and places the result in *r*.

[18] See `http://www2.hursley.ibm.com/decimal/#arithmetic` for details.

[19] The `DEC_Subnormal` flag is particularly expensive to maintain.

decDoubleAnd(r, x, y, set)

Carries out the digit-wise logical And of *x* and *y* and places the result in *r*.

The operands must be zero or positive (sign=0), an integer (finite with exponent=0) and comprise only zeros and/or ones; if not, `DEC_Invalid_operation` is set.

decDoubleCanonical(r, x)

This copies *x* to *r*, ensuring that the encoding of *r* is canonical.

decDoubleClass(x)

This returns the class (`enum decClass`) of the argument *x*.

decDoubleClassString(x)

This returns a description of the class of the argument *x* as a string (`const char *`).

decDoubleCompare(r, x, y, set)

Compares *x* and *y* numerically and places the result in *r*.

The result may be –1, 0, 1, or NaN (unordered); –1 indicates that *x* is less than *y*, 0 indicates that they are numerically equal, and 1 indicates that *x* is greater than *y*. NaN is returned only if *x* or *y* is a NaN.

decDoubleCompareSignal(r, x, y, set)

The same as decDoubleCompare, except that a quiet NaN argument is treated like a signaling NaN (causes `DEC_Invalid_operation` to be set).

decDoubleCompareTotal(r, x, y)

Compares *x* and *y* using the 754r total ordering (which takes into account the exponent) and places the result in *r*. No status is set (a signaling NaN is ordered between Infinity and NaN). The result will be –1, 0, or 1.

decDoubleCompareTotalMag(r, x, y)

The same as decDoubleCompareTotal except that the absolute values of the two arguments are used (as though modified by decDoubleCopyAbs).

decDoubleCopy(r, x)

Copies *x* to *r* quietly (no status is set). This is a bit-wise operation and so the result might not be canonical.

decDoubleCopyAbs(r, x)

Copies *x* to *r* quietly and sets the sign of *r* to 0 (no status is set). This is a bit-wise operation and so the result might not be canonical.

decDoubleCopyNegate(r, x)

Copies *x* to *r* quietly and inverts the sign of *r* (no status is set). This is a bit-wise operation and so the result might not be canonical.

decDoubleCopySign(r, x, y)

Copies *x* to *r* quietly with the sign of *r* set to the sign of *y* (no status is set). This is a bit-wise operation and so the result might not be canonical.

decDoubleDigits(x)

Returns the number of significant digits in *x* as an unsigned 32-bit integer (`uint32_t`). If *x* is a zero or is infinite, 1 is returned. If *x* is a NaN then the number of digits in the payload is returned.

decDoubleDivide(r, x, y, set)

Divides *x* by *y* and places the result in *r*.

decDoubleDivideInteger(r, x, y, set)

Divides *x* by *y* and places the integer part of the result (rounded towards zero) in *r* with exponent=0. If the result would not fit in *r* (because it would have more than `DECDOUBLE_Pmax` digits) then `DEC_Division_impossible` is set.

decDoubleFMA(r, x, y, z, set)

Calculates the fused multiply-add $x \times y + z$ and places the result in *r*. The multiply is carried out first and is exact, so this operation has only the one, final, rounding.

decDoubleFromBCD(r, exp, bcd, sign)

Sets *r* from an exponent *exp* (which may indicate a NaN or infinity), a BCD array *bcd*, and a *sign*.

exp (`int32_t`) is an in-range unbiased exponent or a special value in the form returned by decDoubleGetExponent (listed in `decQuad.h`).

bcd (`const uint8_t *`) is an array of `DECDOUBLE_Pmax` elements, one digit in each byte (BCD8 encoding); the first (most significant) digit is ignored if the result will be a NaN; all are ignored if the result is infinite. All bytes must be in the range 0–9.

sign (`int32_t`) is an integer which must be `DECFLOAT_Sign` to set the sign bit of *r* to 1, or 0 to set it to 0.

For speed, the arguments are not checked; no status is set by this function. The result is undefined if the arguments are invalid or out of range (that is, could not be produced by decDoubleToBCD).

(This function is also available in the decSingle module.)

decDoubleFromInt32(r, i)

Sets *r* from the signed 32-bit integer *i* (`int32_t`). The result is exact and no error is possible.

decDoubleFromNumber(r, dn, set)

This function is implemented as a macro and sets *r* from a decNumber, *dn*, using a decimal64 as a proxy as illustrated in Example 8 in the User's Guide (see page 15).

To use this macro, the `decimal64.h` header file must be included (see the text following the example for more details about compilation).

(This function is also available in the decSingle module.)

decDoubleFromPacked(r, exp, pack)

Sets *r* from an exponent *exp* (which may indicate a special value) and a packed BCD array, *pack*.

exp (`int32_t`) is an in-range unbiased exponent or a special value in the form returned by decDoubleGetExponent (listed in `decQuad.h`).

pack (`const uint8_t *`) is an array of `DECDOUBLE_Pmax` packed decimal digits (one digit per four-bit nibble) followed by a sign nibble, and (for decDouble and decQuad only) prefixed with an extra pad nibble (which is ignored); the sign nibble may be any of the six sign codes listed in `decQuad.h` and described for the decPacked module (see page 62), and digit nibbles must be in the range 0–9.

Like the decDoubleFromBCD function, the the first nibble of *pack* (after the pad nibble, if any) is ignored if the result will be a NaN, and all are ignored if the result is infinite.

For speed, the arguments are not checked; no status is set by this function. The result is undefined if the arguments are invalid or out of range (that is, could not be produced by decNumberToPacked), except that all six sign codes are permitted.

(This function is also available in the decSingle module.)

decDoubleFromString(r, string, set)

Sets *r* from a character string, *string* (`const char *`).

The length of the coefficient and the size of the exponent are checked by this routine, so rounding will be applied if necessary, and this may set status flags (underflow, overflow) will be reported, or rounding applied, as necessary.

There is no limit to the coefficient length for finite inputs; NaN payloads must be integers with no more than `DECDOUBLE_Pmax-1` digits. Exponents may have up to nine significant digits. The syntax of the *string* is fully checked; if it is not valid, the result will be a quiet NaN and an error flag will be set.

(This function is also available in the decSingle module.)

decDoubleFromUInt32(r, u)

Sets *r* from the unsigned 32-bit integer *u* (`uint32_t`). The result is exact and no error is possible.

decDoubleFromWider(r, dq, set)

Sets *r* from an instance, *dq*, of the next-wider format (`const decQuad *`). This narrowing function can cause rounding, overflow, *etc.*, but not Invalid operation (sNaNs are copied and do not signal).

(This function is also available in the decSingle module, but is not available in the decQuad module.)

decDoubleGetCoefficient(x, bcd)

Extracts the coefficient of *x* as a BCD integer into the array *bcd* (`uint8_t *`) and returns the sign as a signed 32-bit integer (`int32_t`). The returned value will be `DECFLOAT_Sign` if *x* has sign=1 or otherwise will be 0.

The digits of the coefficent are written, one digit per byte, into `DECDOUBLE_Pmax` elements of the *bcd* array. If *x* is a NaN the first byte will be zero (the remainder will be the payload), and if it is infinite then all of *bcd* will be zero.

(This function is also available in the decSingle module.)

decDoubleGetExponent(x)

Returns the exponent of *x* as a 32-bit integer (`int32_t`). If *x* is infinite or is a NaN (a special value) the first seven bits of the decDouble are returned, padded with 25 zero bits on the right and with the most significant (sign) bit set to 0. For example, -sNaN would return `0x7e000000` (`DECFLOAT_sNaN`). The possible return values for infinities and NaNs are listed in `decQuad.h`.

(This function is also available in the decSingle module.)

decDoubleInvert(r, x, set)

Carries out the digit-wise logical inversion of *x* and places the result in *r*.

The operand must be zero or positive (sign=0), an integer (finite with exponent=0) and comprise only zeros and/or ones; if not, `DEC_Invalid_operation` is set.

decDoubleIsCanonical(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if the encoding of *x* is canonical, or 0 otherwise.

decDoubleIsFinite(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is neither infinite nor a NaN, or 0 otherwise.

decDoubleIsInfinite(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if the encoding of *x* is an infinity, or 0 otherwise.

decDoubleIsInteger(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is finite and has exponent=0, or 0 otherwise.

decDoubleIsNaN(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is a NaN (quiet or signaling), or 0 otherwise.

decDoubleIsNormal(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is a normal number (that is, is finite, non-zero, and not subnormal), or 0 otherwise.

decDoubleIsSignaling(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is a signaling NaN, or 0 otherwise.

decDoubleIsSignalling(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is a signaling NaN, or 0 otherwise. (This is an alternative spelling of decDoubleIsSignaling.)

decDoubleIsSigned(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* has sign=1, or 0 otherwise.

decDoubleIsSubnormal(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is subnormal (that is, finite, non-zero, and with magnitude less than 10^{emin}), or 0 otherwise.

decDoubleIsZero(x)

Returns an unsigned integer (`uint32_t`) which will be 1 if *x* is a zero, or 0 otherwise.

decDoubleLogB(r, x, set)

Returns the adjusted exponent of *x*, according to 754r rules. That is, the exponent returned is calculated as if the decimal point followed the first significant digit (so, for example, if *x* were 123 then the result would be 2).

If *x* is infinite, the result is +Infinity. If *x* is a zero, the result is –Infinity, and the `DEC_Division_by_zero` flag is set. If *x* is less than zero, the absolute value of *x* is used. If *x*=1, the result is 0. NaNs are handled (propagated) as for arithmetic operations.

decDoubleMax(r, x, y, set)

If both arguments are numeric (not NaNs) this returns the larger of *x* and *y* (compared using decDoubleCompareTotal, to give a well-defined result).

If either (but not both of) *x* or *y* is a quiet NaN then the other argument is the result; otherwise NaNs are handled as for arithmetic operations.

decDoubleMaxMag(r, x, y, set)

The same as decDoubleMax except that the absolute values of the two arguments are used (as though modified by decDoubleCopyAbs).

decDoubleMin(r, x, y, set)

If both arguments are numeric (not NaNs) this returns the smaller of *x* and *y* (compared using decDoubleCompareTotal, to give a well-defined result).

If either (but not both of) *x* or *y* is a quiet NaN then the other argument is the result; otherwise NaNs are handled as for arithmetic operations.

decDoubleMinMag(r, x, y, set)

The same as decDoubleMin except that the absolute values of the two arguments are used (as though modified by decDoubleCopyAbs).

decDoubleMinus(r, x, set)

This has the same effect as 0–*x* where the exponent of the zero is the same as that of *x* (if *x* is finite). The effect is also the same as decFloatCopyNegate except that NaNs are handled as for arithmetic operations (the sign of a NaN is not affected, and an sNaN will signal), the result is canonical, and a zero result has sign=0.

decDoubleMultiply(r, x, y, set)

Multiplies *x* by *y* and places the result in *r*.

decDoubleNextMinus(r, x, set)

Returns the "next" decDouble to *x* in the direction of –Infinity according to 754r rules for *nextDown*. The only status possible is `DEC_Invalid_operation` (from an sNaN).

decDoubleNextPlus(r, x, set)

Returns the "next" decDouble to *x* in the direction of +Infinity according to 754r rules for *nextUp*. The only status possible is `DEC_Invalid_operation` (from an sNaN).

decDoubleNextToward(r, x, y, set)

Returns the "next" decDouble to *x* in the direction of *y* according to 754r rules for *nextAfter*.

If *x*=*y* the the result is decDoubleCopySign(r, x, y). If either operand is a NaN the result is as for arithmetic operations. Otherwise (the operands are numeric and different) the result of adding (or subtracting) an infinitesimal positive amount to *x* and rounding towards +Infinity (or –Infinity) is returned, depending on whether *y* is larger (or smaller) than *x*. The addition will set flags, except that if the result is normal (finite, non-zero, and not subnormal) no flags are set.

decDoubleOr(r, x, y, set)

Carries out the digit-wise logical inclusive Or of *x* and *y* and places the result in *r*.

The operands must be zero or positive (sign=0), an integer (finite with exponent=0) and comprise only zeros and/or ones; if not, `DEC_Invalid_operation` is set.

decDoublePlus(r, x, set)

This has the same effect as 0+*x* where the exponent of the zero is the same as that of *x* (if *x* is finite). The effect is also the same as decFloatCopy except that NaNs are handled as for arithmetic operations (the sign of a NaN is not affected, and an sNaN will signal), the result is canonical, and a zero result has sign=0.

decDoubleQuantize(r, x, y, set)

Returns *x* set to have the same quantum as *y*, if possible (that is, numerically the same value but rounded or padded if necessary to have the same exponent as *y*, for example to round a monetary quantity to cents). More details and an example are given with the decNumberQuantize function (see page 34).

decDoubleRadix(x)

Returns an unsigned integer (`uint32_t`) set to the base used for arithmetic in this module (always ten).

(This function is also available in the decSingle module.)

decDoubleReduce(r, x, set)

Returns a copy of *x* with its coefficient reduced to its shortest possible form without changing the value of the result. This removes all possible trailing zeros from the coefficient (some may remain when the number is very close to the most positive or most negative number). Infinities and NaNs are unchanged and no status is set unless *x* is an sNaN. If *x* is a zero the result exponent is 0.

decDoubleRemainder(r, x, y, set)

Integer-divides *x* by *y* and places the remainder from the division in *r*. That is, if the same *x* and *y* were given to the decDoubleDivideInteger and decDoubleRemainder functions, resulting in *int* and *rem* respectively, then the identity $x = (int \times y) + rem$ holds.

Note that, as for decDoubleDivideInteger, it must be possible to express the intermediate result (*int*) as an integer. That is, it must have no more than `DECDOUBLE_Pmax` digits. If it has too many then `DEC_Division_impossible` is raised.

decDoubleRemainderNear(r, x, y, set)

This is the same as decDoubleRemainder except that the nearest integer (or the nearest even integer if the remainder is equidistant from two) is used for *int* instead of the result from decDoubleDivideInteger. Again, that integer must fit.

decDoubleRotate(r, x, y, set)

The result is a copy of *x* with the digits of the coefficient rotated to the left (if *y* is positive) or to the right (if *y* is negative) without adjusting the exponent or the sign of *x*.

y is the count of positions to rotate and must be a finite integer (with exponent=0) in the range `-DECDOUBLE_Pmax` through `+DECDOUBLE_Pmax`. NaNs are propagated as usual. If *x* is infinite the result is Infinity of the same sign. No status is set unless *y* is invalid or an operand is an sNaN.

decDoubleSameQuantum(x, y)

Returns an unsigned integer (`uint32_t`) which will be 1 if the operands have the same exponent or are both NaNs (quiet or signaling) or both infinite. In all other cases, 0 is returned. No error or status is possible.

decDoubleScaleB(r, x, y, set)

This calculates $x \times 10^{y}$ and places the result in *r*. *y* must be an integer (finite with exponent=0) in the range ±2 × (`DECDOUBLE_Pmax` + `DECDOUBLE_Emax`), typically resulting from decDoubleLogB. Underflow and overflow might occur. NaNs propagate as usual.

decDoubleSetCoefficient(r, bcd, sign)

Sets the coefficient of *r* from a BCD integer in the array *bcd* (`uint8_t *`) and the signed 32-bit integer (`int32_t`) *sign*. *bcd* must have `DECDOUBLE_Pmax` elements in the range 0–9, and sign must be `DECFLOAT_Sign` to set the sign bit of *r* to 1, or 0 to set it to 0.

If *r* is a NaN the first byte of *bcd* will be ignored (the remainder will be the payload), and if it is infinite then all of *bcd* will be ignored (the coefficient will become zero).

For speed, the arguments are not checked; no status is set by this function. The result is undefined if the arguments are invalid or out of range (that is, could not have been produced by decDoubleGetCoefficient).

(This function is also available in the decSingle module.)

decDoubleSetExponent(r, set, exp)

Sets the exponent of *r* from the signed 32-bit integer (`int32_t`) *exp*. *exp* is either an in-range exponent or a special code as returned by decDoubleGetExponent. If *r* becomes infinite then its coefficient is set to zero, if it becomes NaN then the first digit of the coefficient is lost,[20] otherwise the coefficient is unchanged.

For speed, *exp* is not checked; however, underflow or overflow can result. The result is undefined if *exp* is not a value that could have been produced by decDoubleGetExponent.

(This function is also available in the decSingle module.)

[20] A NaN payload has one fewer digit than the coefficient of a finite number.

decDoubleShift(r, x, y, set)

The result is a copy of *x* with the digits of the coefficient shifted to the left (if *y* is positive) or to the right (if *y* is negative) without adjusting the exponent or the sign of *x*. Any digits “shifted in” will be 0.

y is the count of positions to shift and must be a finite integer (with exponent=0) in the range `-DECDOUBLE_Pmax` through `+DECDOUBLE_Pmax`. NaNs are propagated as usual. If *x* is infinite the result is Infinity of the same sign. No status is set unless *y* is invalid or an operand is an sNaN.

decDoubleShow(x, tag)

This function uses `printf` to display a readable rendering of *x*, showing both the encoding (in hexadecimal) and the value, and returns nothing (`void`). The string *tag* (`const char *`) is included in the display and may be used as an identifier for the displayed data.

This function is intended as a debug aid. It is not a programming interface – the format of the displayed data may change from release to release.

(This function is also available in the decSingle module.)

decDoubleSubtract(r, x, y, set)

Subtracts *y* from *x* and places the result in *r*.

decDoubleToBCD(x, exp, bcd)

Converts *x* into an exponent *exp* (`int32_t *`) and a BCD array *bcd* (`uint8_t *`). *exp* is set to the value that would be returned by decDoubleGetExponent(x), and *bcd* and the returned integer (`int32_t`) are as from decDoubleGetCoefficient(x, bcd).

(This function is also available in the decSingle module.)

decDoubleToEngString(x, string)

The same as decDoubleToString(x, string) except that if exponential notation is used the exponent will be a multiple of 3 (“engineering notation”).

(This function is also available in the decSingle module.)

decDoubleToInt32(x, set, round)

Returns a signed 32-bit integer (`int32_t`) which is the value of *x*, rounded to an integer if necessary using the explicit rounding mode *round* (`enum rounding`) instead of the rounding mode in *set*.

If *x* is infinite, is a NaN, or after rounding is outside the range of the result, then `DEC_Invalid_operation` is set. The `DEC_Inexact` flag is not set by this function, even if rounding ocurred.

decDoubleToInt32Exact(x, set, round)

The same as decDoubleToInt32 except that if rounding removes non-zero digits then the `DEC_Inexact` flag is set.

decDoubleToIntegralExact(r, x, set)

Returns the value of *x*, rounded to an integral value using the rounding mode in *set*.

If *x* is infinite, Infinity of the same sign is returned. If *x* is a NaN, the result is as for other arithmetic operations. If rounding removes non-zero digits then the `DEC_Inexact` flag is set.

decDoubleToIntegralValue(r, x, set, round)

Returns the value of *x*, rounded to an integral value using the explicit rounding mode *round* (`enum rounding`) instead of the rounding mode in *set*.

If *x* is infinite, Infinity of the same sign is returned. If *x* is a NaN, the result is as for other arithmetic operations. The `DEC_Inexact` flag is not set by this function, even if rounding ocurred.

decDoubleToNumber(x, dn)

This function is implemented as a macro and sets a decNumber, *dn*, from *x* using a decimal64 as a proxy as illustrated in Example 8 in the User's Guide (see page 15). The decNumber must have sufficient space for the digits in *x*.

To use this macro, the `decimal64.h` header file must be included (see the text following the example for more details about compilation).

A pointer to *dn* is returned (`decNumber *`).

(This function is also available in the decSingle module.)

decDoubleToPacked(x, exp, pack)

Converts *x* into an exponent *exp* (`int32_t *`) and a Packed BCD array *pack* (`uint8_t *`). *exp* is set to the value that would be returned by decDoubleGetExponent(x).

pack receives `DECDOUBLE_Pmax` packed decimal digits (one digit per four-bit nibble) followed by a sign nibble and prefixed (for decDouble and decQuad only) with an extra pad nibble (which is 0). The sign nibble will be `DECPMINUS` if *x* has sign=1 or `DECPPLUS` otherwise. The digit nibbles will be in the range 0–9.

A signed 32-bit integer (`int32_t`) is returned; it will be `DECFLOAT_Sign` if *x* has sign=1 or otherwise will be 0.

(This function is also available in the decSingle module.)

decDoubleToString(x, string)

Converts *x* to a zero-terminated string in the character array *string* (`char *`) and returns *string*. *string* must have at least `DECDOUBLE_String` elements (this count includes the terminator character).

Finite numbers will be converted to a string with exponential notation if the exponent is positive or if the magnitude of *x* is less than 1 and would require more than five zeros between the decimal point and the first significant digit.

Note that strings which are not simply numbers (one of `Infinity`, `-Infinity`, `NaN`, or `sNaN`) are possible. A NaN string may have a leading `-` sign and/or following payload digits. No digits follow the NaN string if the payload is 0.

(This function is also available in the decSingle module.)

decDoubleToUInt32(x, set, enum rounding)

Returns an unsigned 32-bit integer (`uint32_t`) which is the value of *x*, rounded to an integer if necessary using the explicit rounding mode *round* (`enum rounding`) instead of the rounding mode in *set*.

If *x* is infinite, is a NaN, or after rounding is outside the range of the result, then `DEC_Invalid_operation` is set. The `DEC_Inexact` flag is not set by this function, even if rounding ocurred.

Note that –0 converts to 0 and is valid, but all negative numbers are not valid.

decDoubleToUInt32Exact(x, set, enum rounding)

The same as decDoubleToUInt32 except that if rounding removes non-zero digits then the `DEC_Inexact` flag is set.

decDoubleToWider(x, dq)

Converts *x* into a structure, *dq*, of the next-wider format (`decQuad *`) and returns *dq*. Widening is always exact; no status is set (sNaNs are copied and do not signal). The result will be canonical if *x* is canonical, but otherwise might not be.

(This function is also available in the decSingle module, but is not available in the decQuad module.)

decDoubleVersion(void)

Returns a pointer to a character string (`const char *`) which includes the name and the version of the decNumber package.

(This function is also available in the decSingle module.)

decDoubleXor(r, x, y, set)

Carries out the digit-wise logical exclusive Or of *x* and *y* and places the result in *r*.

The operands must be zero or positive (sign=0), an integer (finite with exponent=0) and comprise only zeros and/or ones; if not, `DEC_Invalid_operation` is set.

decDoubleZero(r)

Sets *r* to the unsigned integer zero (that is, with the coefficient, the exponent, and the sign all set to 0).

(This function is also available in the decSingle module.)

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[21]

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.

[21] 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*$\times 2-1$. 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 35) 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*$\times 2-1$ 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 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 customizing, tuning, or testing the package. If you are just using the package for applications, using full IEEE arithmetic, you should not need to modify the parameters controlling these features unless compiling for a big-endian target, in which case the `DECLITEND` setting will need to be altered.

If any of these parameters is changed, all the decNumber source files being used must be recompiled to ensure correct operation.

Customization parameters

The decNumber package includes four compile-time parameters for customizing its use.

The first parameter controls the layout of the compressed decimal formats (see page 44). The storage of a number in these formats must follow the byte ordering ("endianness") of the computing platform; this parameter determines how the formats are loaded and stored. The parameter is set in the `decNumberLocal.h` file, and is:

`DECLITEND` This must be either 1 or 0. If 1, the target platform is assumed to be *little–endian* (for example, AMD and Intel x86 architecture machines are *little–endian*, where the byte containing the sign bit of the format is at the highest memory address). If 0, the target platform is assumed to be *big–endian* (for example, for IBM z-Series machines are *big–endian*, where the byte containing the sign bit of the format is at the lowest memory address).

Many compilers provide a compile-time definition for determining the endianness of the target platform, and `DECLITEND` can in that case be defined to use the provided definition.

If `DECCHECK` (see below) is set to 1 (highly recommended during development), any call to decContextDefault will check that `DECLITEND` is set correctly.

A second customization parameter allows the use of 64-bit integers to improve the performance of certain operations (notably multiplication and the mathematical functions), even when `DECDPUN` (see page 68) is less than 5. (64-bit integers are required for the decNumber module 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. Full 64-bit support is not assumed; only 32×32 to 64 and the inverse (divide) are used; most 32-bit compilers will be able to handle these efficiently without requiring 64-bit hardware.

Another customization parameter controls whether the status flags returned by decNumber are restricted to the five IEEE flags or comprise an extended set which gives more detail about invalid operations along with some extra flags (this does not affect performance). The parameter is set in the `decContext.h` file, and is:

`DECEXTFLAG` This must be either 1 or 0. If 1, the extended set of flags is used. If 0, only 5 bits are used, corresponding to the IEEE 754 and IEEE 854 flags.

The fourth customization 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%.

Tuning parameters

The decNumber package incorporates two compile-time parameters for tuning the performance of the decNumber module. These 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 26), 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-C-89 integers or library calls have to be used for 64-bit intermediate calculations.[22]

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.[23]

One or more buffers of at least `DECBUFFER` bytes will be allocated on the stack, depending on the function. 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.

[22] The decNumber library currently assumes that non-ANSI-C-89 64-bit integers are available if `DECDPUN` is greater than 4. See also the DECUSE64 code parameter.

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

Testing parameters

The decNumber package also incorporates three compile-time parameters which control the inclusion of extra code which provides for extra checking of input arguments, *etc.*, 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.* It is recommended that `DECCHECK` be set to 1 while developing an application that uses decNumber. These parameters are all set in the `decNumberLocal.h` file, and are:

`DECCHECK`	This must be either 1 or 0. If 1, extra checking code, including input structure reference checks, will be included in the module. The latter 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), a message may be displayed using `printf`, and any result will be a valid number of undefined value. This option is especially useful when testing programs that 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 decNumber development use only, so again should normally be left as 0.

Appendix A – Library performance

The decNumber module implements arbitrary-precision arithmetic with fully tailorable parameters (rounding precision, exponent range, and other factors can all be changed at run time). All decNumber operations can accept arbitrary-length operands. Further, decNumber uses a general-purpose internal format (tunable at compile time) which therefore requires conversions to and from any external format (such as strings, BCD, or the proposed IEEE 754 fixed-size decimal encodings).

As a result, the module has significant overheads compared to the dedicated decFloats modules (see page 48) which work directly on the fixed-size encodings. This appendix compares the performance of the decNumber module with the decDouble and decQuad implementations of the same operations. As the tables below show, there is a significant performance advantage in using the decFloats modules when arbitrary-precision operations are not required.

Description of the tables

In the following tables, timings for each operation are given in processor clock cycles. While generally a more useful indicator of comparative performance than "wall clock" times, cycle counts vary considerably with processor architecture. For example, the times below are cycles measured on an Intel Pentium M processor in an IBM X41T Thinkpad;[24] on a Pentium 4 or RISC processor most of the tests would show significantly higher cycle counts. The compiler used also makes a measurable difference. Details of the tests and compiler are given in the notes at the end of this appendix.

Throughout the tables, worst-case cycle times are shown for the main operations in the decDouble and decQuad modules, compared with the same operations using the decNumber module (which requires conversion of operands and results).

Worst-case timings are quoted because best-case timings are generally trivial special cases (such as NaN arguments) and "typical" instruction mixes are very application-dependent.

For each operation, the name of the operation is given, along with a brief description of the worst-case form of the operation. This is the worst case for the decFloats module (in some cases the worst case is different for the decNumber module).

[24] "Intel" and "Pentium" are trade marks of the Intel Corporation. "Thinkpad" is a trade mark of Lenovo.

decDouble performance tables

decDouble (64-bit) conversions		
Operation	**decDouble**	**decNumber**
Encoding to BCD (with exponent) 16-digit finite	44	481
BCD to encoding (with exponent) 16-digit finite	46	327
Encoding to string 16-digit, with exponent	89	133
Exact string to encoding (unrounded) 16-digit, with exponent	188	196
String to encoding (rounded (see page 74)) 16-digit, rounded, with exponent	262	548
Widen to decQuad 16-digit, with exponent	30	209
int32 to encoding From most negative int	39	199
Encoded integer to int32 To most negative int32	32	136

decDouble (64-bit) miscellaneous operations		
Operation	**decDouble**	**decNumber**
Class (classify datum) Negative small subnormal	37	113
Copies (Abs/Negate/Sign) CopySign, copy needed	25	338
Count significant digits Single digit	24	122
Logical And/Or/Xor/Invert (digitwise) 16-digit	23	510
Shift/Rotate Rotate 15 digits	154	583

decDouble (64-bit) computations		
Operation	**decDouble**	**decNumber**
Add (same-sign addition) 16-digit, unaligned, rounded	252	848
Subtract (different-signs addition) 16-digit, unaligned, rounded, borrow	292	
Compare 16-digit, unaligned, mismatch at end	133	442
CompareTotal 16-digit, unaligned, mismatch at end	139	594
Divide 16- by 16-digit (rounded)	848	1576
FMA (fused multiply-add) 16-digit, subtraction, rounded	797	1683
LogB (returns a decDouble) Negative result	48	279
MaxNum/MinNum 16-digit, unaligned, mismatch at end	155	656
Multiply 16×16-digit, round needed	368	1305
Quantize 16-digit, round all-nines	121	422
ScaleB (from decDoubles) Underflow	208	513
To integral value 16-digit, round all-nines	135	709

decQuad performance tables

decQuad (128-bit) conversions		
Operation	**decQuad**	**decNumber**
Encoding to BCD (with exponent) 34-digit finite	53	460
BCD to encoding (with exponent) 34-digit finite	75	307
Encoding to string 34-digit, with exponent	164	239
Exact string to encoding (unrounded) 34-digit, with exponent	289	597
String to encoding (rounded (see page 74)) 34-digit, rounded, with exponent	489	956
Narrow to decDouble 34-digit, all nines	145	612
int32 to encoding From most negative int	44	199
Encoded integer to int32 To most negative int32	32	156

decQuad (128-bit) miscellaneous operations		
Operation	**decQuad**	**decNumber**
Class (classify number) Negative small subnormal	53	133
Copies (Abs/Negate/Sign) CopySign, copy needed	28	380
Count significant digits Single digit	28	138
Logical And/Or/Xor/Invert (digitwise) 34-digit	28	622
Shift/Rotate Rotate 33 digits	222	812

decQuad (128-bit) computations		
Operation	**decQuad**	**decNumber**
Add (same-sign addition) 34-digit, aligned	433	1180
Subtract (different-signs addition) 34-digit, unaligned, rounded, borrow	468	
Compare 34-digit, unaligned, mismatch at end	187	1125
CompareTotal 34-digit, unaligned, mismatch at end	208	778
Divide 34- by 34-digit (rounded)	2045	3172
FMA (fused multiply-add) 34-digit, subtraction, rounded	1649	2707
LogB (returns a decQuad) Negative result	58	299
MaxNum/MinNum 34-digit, unaligned, mismatch at end	257	857
Multiply 34×34-digit, round needed	819	2235
Quantize 34-digit, round all-nines	197	670
ScaleB (from decQuads) Underflow	248	553
To integral value 34-digit, round all-nines	211	886

Notes

The following notes apply to all the tables in this appendix.

1. All timings were made on an IBM X41T Tablet PC (Pentium M, 1.5GHz, 1.5GB RAM) under Windows XP Tablet Edition with SP2; the modules were compiled using GCC version 3.4.4 with optimization settings `-O3 -march=i686`.
2. The default tuning parameters were used (DECUSE64=1, DECDPUN=3, etc.); some of these only affect decNumber.
3. Timings include call/return overhead, and for the decNumber module also include the costs of converting operand(s) to decNumbers and results back to the appropriate format using the decimal64 or decimal128 module.
4. “BCD” for decNumber is Packed BCD, using the decPacked module; for decFloats it is 8-bit BCD.

5. The worst case for each operation is not always obvious from the code and is implementation-dependent (for example, in the decFloats modules, an unaligned add is sometimes faster than an aligned add). It is possible that there may be unusual cases which are slower than the decFloats counts listed above, although a wide variety of micro-benchmarks have been tried.
6. A string-to-number conversion can theoretically have an arbitrarily large worst case as the string could contain any number of leading, trailing, or embedded zeros; the timings above measured cases where the input string's coefficient had up to eight more digits than the precision of the destination format.

Appendix B – 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 65).
- 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 has been added. This allows the compressed decimal formats (see page 44) 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 66) 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 3.37 (22 November 2006)

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

Changes in Version 3.40 (18 April 2007)

This is a major upgrade to decNumber to add logical and shifting functions together with generalizations of most of the new functions defined in the proposed revision of the IEEE 754 standard. (Note that as that standard has not yet been approved and published, it is still possible for changes to occur.)

The changes included in this update are:

- Thirty-four new functions have been added to the decNumber module (all names have the prefix **decNumber**): **And**, **CompareSignal**, **CompareTotalMag**, **CopyAbs**, **CopyNegate**, **CopySign**, **Class**, **ClassToString**, **FMA**, **FromInt**, **FromUInt**, **GetBCD**, **Invert**, **IsCanonical**, **IsFinite**, **IsNormal**, **IsSpecial**, **IsSubnormal**, **LogB**, **MaxMag**, **MinMag**, **NextMinus**, **NextPlus**, **NextToward**, **Or**, **Radix**, **Rotate**, **ScaleB**, **SetBCD**, **Shift**, **ToIntegralExact**, **ToInt32**, **ToUInt32**, **Xor**.
- Two new functions have been added to each of the three decimalNN modules: **decimalNNIsCanonical**, **decimalNNCanonical**.
- The `DECENDIAN` setting (in decNumberLocal.h) has been removed to improve performance; instead, you must set the `DECLITEND` parameter (see page 66) to 1 if compiling for a little-endian target, or to 0 if compiling for a big-endian target. If `DECCHECK` is set to 1 (highly recommended while testing), any call to decContextDefault will check that `DECLITEND` is set correctly.
- The `DECEXTFLAG` parameter (see page 66) has been added (in decContext.h). This controls whether the status flags returned by decNumber are restricted to the five IEEE flags or comprise an extended set which gives more detail about invalid operations along with some extra flags (this does not affect performance). The default is the extended set of flags, as in earlier versions of decNumber.

Changes in Version 3.41 (7 May 2007)

- Minor corrections (notably to the descriptions of the **FromString** functions) and clarifications have been made.

Changes in Version 3.50 (4 June 2007)

This is a major upgrade to decNumber which adds three new modules (`decSingle`, `decDouble`, and `decQuad`) with 175 new functions. These modules provide functions which work directly with the decimal32, decimal64, and decimal128 formats, to provide high performance when arbitrary-precision calculations are not needed.

In addition to the new modules, the changes included in this update are:

- Two new examples have been added to the User's Guide, showing how to use the new modules either stand-alone or in conjunction with the decNumber module.
- Eleven new functions have been added to the decContext module to match those defined in the proposed revision of the IEEE 754 standard. (Note that as that standard has not yet been approved and published, it is still possible for changes to be necessary in these.)
- Synonyms for `DEC_INIT_DECIMAL32`, *etc.*, have been provided to match the names of the new modules, called `DEC_INIT_DECSINGLE`, *etc.*

- The decClasses enumeration and strings have been moved from `decNumber.h` to `decContext.h` so that they can be used from all modules.
- The `DECVERSION` constant has been moved from `decNumber.h` to `decNumberLocal.h` so that it can be used from all modules.
- The decNumberNormalize function has been renamed decNumberReduce for clarity (it is still available in the code and header file under the old name, for compatibility).
- A new appendix comparing the performance of the decNumber module to the new decDouble and decQuad modules has been added.
- Numerous clarifications and editorial changes have been made.

Index