MPFR

The Multiple Precision Floating-Point Reliable Library
Edition 2.0.1
April 2002

The MPFR team, LORIA/INRIA Lorraine

Published by the Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA

MPFR Copying Conditions

This library is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. The library is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this library that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the library, that you receive source code or else can get it if you want it, that you can change this library or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the MPFR library, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the MPFR library. If it is modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license for the MPFR library are found in the Lesser General Public License that accompanies the source code. See the file COPYING.LIB.

1 Introduction to MPFR

MPFR is a portable library written in C for arbitrary precision arithmetic on reliable floating-point numbers. It is based on the GNU MP library. It aims to extend the class of floating-point numbers provided by the GNU MP library by *reliable* floating-point numbers. It may replace the GNU MP floating-point numbers in a future release. The main differences with the `mpf` class are:

- the `mpfr` code is portable, i.e. the result of any operation does not depend (or should not) on the machine word size `mp_bits_per_limb` (32 or 64 on most machines);
- the precision in bits can be set exactly to any valid value for each variable (including very small precision);
- `mpfr` provides the four rounding modes from the IEEE 754 standard.

In particular, with a precision of 53 bits, `mpfr` should be able to exactly reproduce all computations with double-precision machine floating-point numbers (`double` type in C), except the default exponent range is much wider and subnormal numbers are not implemented.

This version of MPFR is released under the GNU Lesser General Public License. It is permitted to link MPFR to non-free programs, as long as when distributing them the MPFR source code and a means to re-link with a modified MPFR is provided.

1.1 How to use this Manual

Everyone should read Chapter 4 [MPFR Basics], page 5. If you need to install the library yourself, you need to read Chapter 2 [Installing MPFR], page 3, too.

The rest of the manual can be used for later reference, although it is probably a good idea to glance through it.

2 Installing MPFR

To build MPFR, you first have to install GNU MP (version 4.0.1 or higher) on your computer. You need a C compiler, preferably GCC, but any reasonable compiler should work. And you need a standard Unix 'make' program, plus some other standard Unix utility programs.

Here are the steps needed to install the library on Unix systems (more details are provided in the 'INSTALL' file):

1. In most cases, './configure --with-gmp=/usr/local/gmp' should work, where the directory '/usr/local/gmp' is where you have installed GNU MP. When you install GNU MP, you have to copy the files 'config.h', 'gmp-impl.h', 'gmp-mparam.h' and 'longlong.h' from the GNU MP source directory to '/usr/local/gmp/include'; these additional files are needed by MPFR. If you get error messages, you might check that you use the same compiler and compile options as for GNU MP (see the 'INSTALL' file).
2. 'make'

   This will compile MPFR, and create a library archive file 'libmpfr.a' in the working directory.
3. 'make check'

   This will make sure MPFR was built correctly. If you get error messages, please report this to 'mpfr@loria.fr'. (See Chapter 3 [Reporting Bugs], page 4, for information on what to include in useful bug reports.)
4. 'make install'

   This will copy the files 'mpfr.h' and 'mpf2mpfr.h', and 'libmpfr.a', to the directories '/usr/local/include' and '/usr/local/lib' respectively (or if you passed the '--prefix' option to 'configure', to the directory given as argument to '--prefix'). This will also install 'mpfr.info' in '/usr/local/info'.

There are some other useful make targets:

- 'mpfr.dvi' or 'dvi'

  Create a DVI version of the manual, in 'mpfr.dvi'.
- 'mpfr.ps'

  Create a Postscript version of the manual, in 'mpfr.ps'.
- 'clean'

  Delete all object files and archive files, but not the configuration files.
- 'distclean'

  Delete all files not included in the distribution.
- 'uninstall' Delete all files copied by 'make install'.

2.1 Known Build Problems

MPFR suffers from all bugs from the GNU MP library, plus many many more.

Please report other problems to 'mpfr@loria.fr'. See Chapter 3 [Reporting Bugs], page 4. Some bug fixes are available on the MPFR web page 'http://www.loria.fr/projets/mpfr/' or 'http://www.mpfr.org/'.

3 Reporting Bugs

If you think you have found a bug in the MPFR library, first have a look on the MPFR web page '`http://www.loria.fr/projets/mpfr/`' or '`http://www.mpfr.org/`': perhaps this bug is already known, in which case you will find a workaround for it. Otherwise, please investigate and report it. We have made this library available to you, and it is not to ask too much from you, to ask you to report the bugs that you find.

There are a few things you should think about when you put your bug report together.

You have to send us a test case that makes it possible for us to reproduce the bug. Include instructions on how to run the test case.

You also have to explain what is wrong; if you get a crash, or if the results printed are incorrect and in that case, in what way.

Please include compiler version information in your bug report. This can be extracted using '`cc -V`' on some machines, or, if you're using gcc, '`gcc -v`'. Also, include the output from '`uname -a`'.

If your bug report is good, we will do our best to help you to get a corrected version of the library; if the bug report is poor, we won't do anything about it (aside of chiding you to send better bug reports).

Send your bug report to: '`mpfr@loria.fr`'.

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please send a note to the same address.

4 MPFR Basics

All declarations needed to use MPFR are collected in the include file '`mpfr.h`'. It is designed to work with both C and C++ compilers. You should include that file in any program using the MPFR library:

```
#include "mpfr.h"
```

4.1 Nomenclature and Types

Floating-point number or *Float* for short, is an arbitrary precision mantissa with a limited precision exponent. The C data type for such objects is `mpfr_t`. A floating-point number can have three special values: Not-a-Number (NaN) or plus or minus Infinity. NaN represents a value which cannot be represented in the floating-point format, like 0 divided by 0, or Infinity minus Infinity.

The *Precision* is the number of bits used to represent the mantissa of a floating-point number; the corresponding C data type is `mp_prec_t`. The precision can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`. In the current implementation, `MPFR_PREC_MIN` is equal to 2 and `MPFR_PREC_MAX` is equal to `ULONG_MAX`.

The *rounding mode* specifies the way to round the result of a floating-point operation, in case the exact result can not be represented exactly in the destination mantissa; the corresponding C data type is `mp_rnd_t`.

A *limb* means the part of a multi-precision number that fits in a single word. (We chose this word because a limb of the human body is analogous to a digit, only larger, and containing several digits.) Normally a limb contains 32 or 64 bits. The C data type for a limb is `mp_limb_t`.

4.2 Function Classes

There is only one class of functions in the MPFR library:

1. Functions for floating-point arithmetic, with names beginning with `mpfr_`. The associated type is `mpfr_t`.

4.3 MPFR Variable Conventions

As a general rule, all MPFR functions expect output arguments before input arguments. This notation is based on an analogy with the assignment operator.

MPFR allows you to use the same variable for both input and output in the same expression. For example, the main function for floating-point multiplication, `mpfr_mul`, can be used like this: `mpfr_mul (x, x, x, rnd_mode)`. This computes the square of x with rounding mode `rnd_mode` and puts the result back in x.

Before you can assign to an MPFR variable, you need to initialize it by calling one of the special initialization functions. When you're done with a variable, you need to clear it out, using one of the functions for that purpose.

A variable should only be initialized once, or at least cleared out between each initialization. After a variable has been initialized, it may be assigned to any number of times.

For efficiency reasons, avoid to initialize and clear out a variable in loops. Instead, initialize it before entering the loop, and clear it out after the loop has exited.

You don't need to be concerned about allocating additional space for MPFR variables, since any variable has a mantissa of fixed size. Hence unless you change its precision, or clear and reinitialize it, a floating-point variable will have the same allocated space during all its life.

4.4 Compatibility with MPF

A header file 'mpf2mpfr.h' is included in the distribution of MPFR for compatibility with the GNU MP class MPF. After inserting the following two lines after the `#include "gmp.h"` line,

`#include "mpfr.h"`

`#include "mpf2mpfr.h"`

any program written for MPF can be linked directly with MPFR without any changes. All operations are then performed with the default MPFR rounding mode, which can be reset with `mpfr_set_default_rounding_mode`.

mp_rnd_t **__gmp_default_rounding_mode** Global Variable

The default rounding mode (to nearest initially).

4.5 Getting the Latest Version of MPFR

The latest version of MPFR is available from 'http://www.loria.fr/projets/mpfr/' or 'http://www.mpfr.org/'.

5 Floating-point Functions

The floating-point functions expect arguments of type `mpfr_t`.

The MPFR floating-point functions have an interface that is similar to the GNU MP integer functions. The function prefix for floating-point operations is `mpfr_`.

There is one significant characteristic of floating-point numbers that has motivated a difference between this function class and other MPFR function classes: the inherent inexactness of floating-point arithmetic. The user has to specify the precision of each variable. A computation that assigns a variable will take place with the precision of the assigned variable; the cost of that computation should not depend from the precision of variables used as input on average.

The precision of a calculation is defined as follows: Compute the requested operation exactly (with "infinite precision"), and round the result to the destination variable precision with the given rounding mode. Even if the user has asked for a very high precision, MP will not calculate with superfluous digits. For example, if two low-precision numbers of nearly equal magnitude are added, the precision of the result will be limited to what is required to represent the result accurately.

The MPFR floating-point functions are intended to be a smooth extension of the IEEE P754 arithmetic. The results obtained on one computer should not differ from the results obtained on a computer with a different word size.

5.1 Rounding Modes

The following four rounding modes are supported:

- `GMP_RNDN`: round to nearest
- `GMP_RNDZ`: round towards zero
- `GMP_RNDU`: round towards plus infinity
- `GMP_RNDD`: round towards minus infinity

The '`round to nearest`' mode works as in the IEEE P754 standard: in case the number to be rounded lies exactly in the middle of two representable numbers, it is rounded to the one with the least significant bit set to zero. For example, the number 5, which is represented by (101) in binary, is rounded to (100)=4 with a precision of two bits, and not to (110)=6. This rule avoids the *drift* phenomenon mentioned by Knuth in volume 2 of The Art of Computer Programming (section 4.2.2, pages 221-222).

Most MPFR functions take as first argument the destination variable, as second and following arguments the input variables, as last argument a rounding mode, and have a return value of type `int`. If this value is zero, it means that the value stored in the destination variable is the exact result of the corresponding mathematical function. If the returned value is positive (resp. negative), it means the value stored in the destination variable is greater (resp. lower) than the exact result. For example with the `GMP_RNDU` rounding mode, the returned value is usually positive, except when the result is exact, in which case it is zero. In the case of an infinite result, it is considered as inexact when it was obtained by overflow, and exact otherwise. A NaN result (Not-a-Number) always corresponds to an inexact return value.

`void` **mpfr_set_default_rounding_mode** (`mp_rnd_t` *rnd*) Function

Sets the default rounding mode to *rnd*. The default rounding mode is to nearest initially.

`int` **mpfr_round_prec** (`mpfr_t` *x*, `mp_rnd_t` *rnd*, `mp_prec_t` *prec*) Function

Rounds *x* according to *rnd* with precision *prec*, which may be different from that of *x*. If *prec* is greater or equal to the precision of *x*, then new space is allocated for the mantissa, and it is filled with zeroes. Otherwise, the mantissa is rounded to precision *prec* with the given direction. In both cases, the precision of *x* is changed to *prec*. The returned value is zero when the result is exact, positive when it is greater than the original value of *x*, and negative when it is smaller. The precision *prec* can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`.

`void` **mpfr_set_machine_rnd_mode** (`mp_rnd_t` *rnd*) Function

Set the machine rounding mode to *rnd*. This function is provided only when the operating system supports the ISOC9X standard interface for setting rounding modes (i.e. through the header file <fenv.h>).

`char *` **mpfr_print_rnd_mode** (`mp_rnd_t` *rnd*) Function

Returns the input string (GMP_RNDD, GMP_RNDU, GMP_RNDN, GMP_RNDZ) corresponding to the rounding mode *rnd* or a null pointer if *rnd* is an invalid rounding mode.

5.2 Exceptions

Note: Overflow handling is still experimental and currently implemented very partially. If an overflow occurs internally at the wrong place, anything can happen (crash, wrong results, etc).

`mp_exp_t` **mpfr_get_emin** (`void`) Function
`mp_exp_t` **mpfr_get_emax** (`void`) Function

Return the (current) smallest and largest exponents allowed for a floating-point variable.

`int` **mpfr_set_emin** (`mp_exp_t` *exp*) Function
`int` **mpfr_set_emax** (`mp_exp_t` *exp*) Function

Set the smallest and largest exponents allowed for a floating-point variable. Return a non-zero value when *exp* is not in the range accepted by the implementation (in that case the smallest or largest exponent is not changed), and zero otherwise. If the user changes the exponent range, it is her/his responsibility to check that all current floating-point variables are in the new allowed range (for example using `mpfr_check_range`, otherwise the subsequent behaviour will be undefined, in the sense of the ISO C standard.

`int` **mpfr_check_range** (`mpfr_t` *x*, `mp_rnd_t` *rnd*) Function

Return zero if the exponent of *x* is in the current allowed range (see `mpfr_get_emin` and `mpfr_get_emax`), otherwise reset *x* according to the current floating-point system and the rounding mode *rnd*, and return a positive value if the rounded result is larger than the original one, a negative value otherwise (the result cannot be exact in that case).

void **mpfr_clear_underflow** (void) Function
void **mpfr_clear_overflow** (void) Function
void **mpfr_clear_nanflag** (void) Function
void **mpfr_clear_inexflag** (void) Function

Clear the underflow, overflow, invalid, and inexact flags.

void **mpfr_clear_flags** (void) Function

Clear all global flags (underflow, overflow, inexact, invalid).

int **mpfr_underflow_p** (void) Function
int **mpfr_overflow_p** (void) Function
int **mpfr_nanflag_p** (void) Function
int **mpfr_inexflag_p** (void) Function

Return the corresponding (underflow, overflow, invalid, inexact) flag, which is non-zero iff the flag is set.

5.3 Initialization and Assignment Functions

void **mpfr_set_default_prec** (mp_prec_t *prec*) Function

Set the default precision to be **exactly** *prec* bits. The precision of a variable means the number of bits used to store its mantissa. All subsequent calls to `mpfr_init` will use this precision, but previously initialized variables are unaffected. This default precision is set to 53 bits initially. The precision can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`.

mp_prec_t **mpfr_get_default_prec** () Function

Returns the default MPFR precision in bits.

An `mpfr_t` object must be initialized before storing the first value in it. The functions `mpfr_init` and `mpfr_init2` are used for that purpose.

void **mpfr_init** (mpfr_t *x*) Function

Initialize *x*, and set its value to NaN. Normally, a variable should be initialized once only or at least be cleared, using `mpfr_clear`, between initializations. The precision of *x* is the default precision, which can be changed by a call to `mpfr_set_default_prec`.

void **mpfr_init2** (mpfr_t *x*, mp_prec_t *prec*) Function

Initialize *x*, set its precision to be **exactly** *prec* bits, and set its value to NaN. Normally, a variable should be initialized once only or at least be cleared, using `mpfr_clear`, between initializations. To change the precision of a variable which has already been initialized, use `mpfr_set_prec` instead. The precision *prec* can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`.

void **mpfr_clear** (mpfr_t *x*) Function

Free the space occupied by *x*. Make sure to call this function for all `mpfr_t` variables when you are done with them.

Here is an example on how to initialize floating-point variables:

```
{
  mpfr_t x, y;
  mpfr_init (x); /* use default precision */
  mpfr_init2 (y, 256); /* precision exactly 256 bits */
  ...
  /* Unless the program is about to exit, do ... */
  mpfr_clear (x);
  mpfr_clear (y);
}
```

The following two functions are useful for changing the precision during a calculation. A typical use would be for adjusting the precision gradually in iterative algorithms like Newton-Raphson, making the computation precision closely match the actual accurate part of the numbers.

`int` **mpfr_set_prec** (`mpfr_t` *x*, `mp_prec_t` *prec*) — Function

Reset the precision of *x* to be **exactly** *prec* bits. The previous value stored in *x* is lost. It is equivalent to a call to `mpfr_clear(x)` followed by a call to `mpfr_init2(x, prec)`, but more efficient as no allocation is done in case the current allocated space for the mantissa of *x* is enough. The precision *prec* can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`. It returns a non-zero value iff the memory allocation failed.

In case you want to keep the previous value stored in *x*, use `mpfr_round_prec` instead.

`mp_prec_t` **mpfr_get_prec** (`mpfr_t` *x*) — Function

Return the precision actually used for assignments of *x*, i.e. the number of bits used to store its mantissa.

`void` **mpfr_set_prec_raw** (`mpfr_t` *x*, `unsigned long int` *p*) — Function

Reset the precision of *x* to be **exactly** *prec* bits. The only difference with `mpfr_set_prec` is that *p* is assumed to be small enough so that the mantissa fits into the current allocated memory space for *x*. Otherwise an error will occur.

5.4 Assignment Functions

These functions assign new values to already initialized floats (see Section 5.3 [Initializing Floats], page 9).

`int` **mpfr_set** (`mpfr_t` *rop*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) — Function
`int` **mpfr_set_ui** (`mpfr_t` *rop*, `unsigned long int` *op*, `mp_rnd_t` *rnd*) — Function
`int` **mpfr_set_si** (`mpfr_t` *rop*, `long int` *op*, `mp_rnd_t` *rnd* — Function
`int` **mpfr_set_d** (`mpfr_t` *rop*, `double` *op*, `mp_rnd_t` *rnd*) — Function
`int` **mpfr_set_z** (`mpfr_t` *rop*, `mpz_t` *op*, `mp_rnd_t` *rnd*) — Function
`int` **mpfr_set_q** (`mpfr_t` *rop*, `mpq_t` *op*, `mp_rnd_t` *rnd*) — Function

Set the value of *rop* from *op*, rounded to the precision of *rop* towards the given direction *rnd*. Please note that even a `long int` may have to be rounded, if the destination precision is less than the machine word width. The return value is zero when *rop*=*op*, positive when *rop*>*op*, and negative when *rop*<*op*. For `mpfr_set_d`, be careful that the input number *op* may not be exactly representable as a double-precision number (this happens for 0.1 for instance), in which case it is first rounded by the C compiler to a double-precision number, and then only to a mpfr floating-point number.

`int` **mpfr_set_str** (`mpfr_t` *x*, `char` **s*, `int` *base*, `mp_rnd_t` *rnd*) Function

Set *x* to the value of the string *s* in base *base* (between 2 and 36), rounded in direction *rnd* to the precision of *x*. The exponent is read in decimal. This function returns −1 if an internal overflow occurred (for instance, because the exponent is too large). Otherwise it returns 0 if the base is valid and if the entire string up to the final '\0' is a valid number in base *base*, and 1 if the input is incorrect.

`void` **mpfr_set_str_raw** (`mpfr_t` *x*, `char` **s*) Function

Set *x* to the value of the binary number in string *s*, which has to be of the form `+/-`xxxx.xxxxxxEyy. The exponent is read in decimal, but is interpreted as the power of two to be multiplied by the mantissa. The mantissa length of *s* has to be less or equal to the precision of *x*, otherwise an error occurs. If *s* starts with `N`, it is interpreted as NaN (Not-a-Number); if it starts with `I` after the sign, it is interpreted as infinity, with the corresponding sign.

`int` **mpfr_set_f** (`mpfr_t` *x*, `mpf_t` *y*, `mp_rnd_t` *rnd*) Function

Set *x* to the GNU MP floating-point number *y*, rounded with the *rnd* mode and the precision of *x*. The returned value is zero when *x*=*y*, positive when *x*>*y*, and negative when *x*<*y*.

`void` **mpfr_set_inf** (`mpfr_t` *x*, `int` *sign*) Function
`void` **mpfr_set_nan** (`mpfr_t` *x*) Function

Set the variable *x* to infinity or NaN (Not-a-Number) respectively. In `mpfr_set_inf`, *x* is set to plus infinity iff *sign* is positive.

`void` **mpfr_swap** (`mpfr_t` *x*, `mpfr_t` *y*) Function

Swap the values *x* and *y* efficiently. Warning: the precisions are exchanged too; in case the precisions are different, `mpfr_swap` is thus not equivalent to three `mpfr_set` calls using a third auxiliary variable.

5.5 Combined Initialization and Assignment Functions

int **mpfr_init_set** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Macro
int **mpfr_init_set_ui** (mpfr_t *rop*, unsigned long int *op*, mp_rnd_t *rnd*) Macro
int **mpfr_init_set_si** (mpfr_t *rop*, signed long int *op*, mp_rnd_t *rnd*) Macro
int **mpfr_init_set_d** (mpfr_t *rop*, double *op*, mp_rnd_t *rnd*) Macro
int **mpfr_init_set_f** (mpfr_t *rop*, mpf_t *op*, mp_rnd_t *rnd*) Macro
int **mpfr_init_set_z** (mpfr_t *rop*, mpz_t *op*, mp_rnd_t *rnd*) Macro
int **mpfr_init_set_q** (mpfr_t *rop*, mpq_t *op*, mp_rnd_t *rnd*) Macro

Initialize *rop* and set its value from *op*, rounded to direction *rnd*. The precision of *rop* will be taken from the active default precision, as set by mpfr_set_default_prec. The return value if zero if *rop*=*op*, positive if *rop*>*op*, and negative when *rop*<*op*.

int **mpfr_init_set_str** (mpfr_t *x*, char **s*, int *base*, mp_rnd_t *rnd*) Function

Initialize *x* and set its value from the string *s* in base *base*, rounded to direction *rnd*. See mpfr_set_str.

5.6 Conversion Functions

double **mpfr_get_d** (mpfr_t *op*, mp_rnd_t *rnd*) Function

Convert *op* to a double, using the rounding mode *rnd*.

double **mpfr_get_d1** (mpfr_t *op*) Function

Convert *op* to a double, using the default MPFR rounding mode (see function mpfr_set_default_rounding_mode).

mp_exp_t **mpfr_get_z_exp** (mpz_t *z*, mpfr_t *op*) Function

Puts the mantissa of *op* into *z*, and returns the exponent *exp* such that *op* equals *z* multiplied by two exponent *exp*.

char * **mpfr_get_str** (char **str*, mp_exp_t **expptr*, int *base*, size_t *n_digits*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Convert *op* to a string of digits in base *base*, with rounding in direction *rnd*. The base may vary from 2 to 36. Generate exactly *n_digits* significant digits.

If *n_digits* is 0, it writes the maximum possible number of digits giving an exact rounding in the given base *base* with the direction *rnd*. In other words, if *op* was the exact rounding of a real number in direction *rnd*, then the written value is also an exact rounding in base *base* of that real number with the same precision. An error occurs when one is unable to determine the leading digit, which can happen especially if the precision of *op* is small.

If *str* is a null pointer, space for the mantissa is allocated using the default allocation function, and a pointer to the string is returned. In that case, the user should her/himself free the corresponding memory with (*_mp_free_func)(s, strlen(s) + 1).

If *str* is not a null pointer, it should point to a block of storage large enough for the mantissa, i.e., *n_digits* + 2 or more. The extra two bytes are for a possible minus sign, and for the terminating null character.

If the input number is a real number, the exponent is written through the pointer *expptr* (the current minimal exponent for 0).

If *n_digits* is 0, note that the space requirements for *str* in this case will be impossible for the user to predetermine. Therefore, one needs to pass a null pointer for the string argument whenever *n_digits* is 0.

The generated string is a fraction, with an implicit radix point immediately to the left of the first digit. For example, the number 3.1416 would be returned as "31416" in the string and 1 written at *expptr*.

A pointer to the string is returned, unless there is an error, in which case a null pointer is returned.

5.7 Basic Arithmetic Functions

int **mpfr_add** (mpfr_t *rop*, mpfr_t *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_add_ui** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_add_z** (mpfr_t *rop*, mpfr_t *op1*, mpz_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_add_q** (mpfr_t *rop*, mpfr_t *op1*, mpq_t *op2*, mp_rnd_t *rnd*) Function

Set *rop* to $op1 + op2$ rounded in the direction *rnd*. The return value is zero if *rop* is exactly $op1 + op2$, positive if *rop* is larger than $op1 + op2$, and negative if *rop* is smaller than $op1 + op2$.

int **mpfr_sub** (mpfr_t *rop*, mpfr_t *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_ui_sub** (mpfr_t *rop*, unsigned long int *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_sub_ui** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_sub_z** (mpfr_t *rop*, mpfr_t *op1*, mpz_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_sub_q** (mpfr_t *rop*, mpfr_t *op1*, mpq_t *op2*, mp_rnd_t *rnd*) Function

Set *rop* to $op1 - op2$ rounded in the direction *rnd*. The return value is zero if *rop* is exactly $op1 - op2$, positive if *rop* is larger than $op1 - op2$, and negative if *rop* is smaller than $op1 - op2$.

int **mpfr_mul** (mpfr_t *rop*, mpfr_t *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_mul_ui** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_mul_z** (mpfr_t *rop*, mpfr_t *op1*, mpz_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_mul_q** (mpfr_t *rop*, mpfr_t *op1*, mpq_t *op2*, mp_rnd_t *rnd*) Function

Set *rop* to $op1 \times op2$ rounded in the direction *rnd*. Return 0 if the result is exact, a positive value if $rop > op1 \times op2$, a negative value otherwise.

int **mpfr_div** (mpfr_t *rop*, mpfr_t *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_ui_div** (mpfr_t *rop*, unsigned long int *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_div_ui** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_div_z** (mpfr_t *rop*, mpfr_t *op1*, mpz_t *op2*, mp_rnd_t *rnd*) Function
int **mpfr_div_q** (mpfr_t *rop*, mpfr_t *op1*, mpq_t *op2*, mp_rnd_t *rnd*) Function

Set *rop* to *op1*/*op2* rounded in the direction *rnd*. These functions return 0 if the division is exact, a positive value when *rop* is larger than *op1* divided by *op2*, and a negative value otherwise.

int **mpfr_sqrt** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_sqrt_ui** (mpfr_t *rop*, unsigned long int *op*, mp_rnd_t *rnd*) Function

Set *rop* to $\sqrt{op}$ rounded in the direction *rnd*. Set *rop* to NaN if *op* is negative. Return 0 if the operation is exact, a non-zero value otherwise.

int **mpfr_pow_ui** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_ui_pow_ui** (mpfr_t *rop*, unsigned long int *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function

Set *rop* to *op1* raised to *op2*. The computation is done by binary exponentiation. Return 0 if the result is exact, a non-zero value otherwise (but the sign of the return value has no meaning).

int **mpfr_ui_pow** (mpfr_t *rop*, unsigned long int *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function

Set *rop* to *op1* raised to *op2*, rounded to the direction *rnd* with the precision of *rop*. Return zero iff the result is exact, a positive value when the result is greater than *op1* to the power *op2*, and a negative value when it is smaller.

int **mpfr_pow_si** (mpfr_t *rop*, mpfr_t *op1*, long int *op2*, mp_rnd_t *rnd*) Function

Set *rop* to *op1* raised to the power *op2*, rounded to the direction *rnd* with the precision of *rop*. Return zero iff the result is exact.

int **mpfr_pow** (mpfr_t *rop*, mpfr_t *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function

Set *rop* to *op1* raised to the power *op2*, rounded to the direction *rnd* with the precision of *rop*. If *op1* is negative then *rop* is set to NaN, even if *op2* is an integer. Return zero iff the result is exact.

int **mpfr_neg** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to $-op$ rounded in the direction *rnd*. Just changes the sign if *rop* and *op* are the same variable.

int **mpfr_abs** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to the absolute value of *op*, rounded in the direction *rnd*. Return 0 if the result is exact, a positive value if *rop* is larger than the absolute value of *op*, and a negative value otherwise.

int **mpfr_mul_2exp** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_mul_2ui** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_mul_2si** (mpfr_t *rop*, mpfr_t *op1*, long int *op2*, mp_rnd_t *rnd*) Function

Set *rop* to $op1 \times 2^{op2}$ rounded to the direction *rnd*. Just increases the exponent by *op2* when *rop* and *op1* are identical. Return zero when *rop*=*op1*, a positive value when *rop*>*op1*, and a negative value when *rop*<*op1*. Note: The `mpfr_mul_2exp` function is defined for compatibility reasons; you should use `mpfr_mul_2ui` (or `mpfr_mul_2si`) instead.

int **mpfr_div_2exp** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_div_2ui** (mpfr_t *rop*, mpfr_t *op1*, unsigned long int *op2*, mp_rnd_t *rnd*) Function
int **mpfr_div_2si** (mpfr_t *rop*, mpfr_t *op1*, long int *op2*, mp_rnd_t *rnd*) Function

Set *rop* to $op1/2^{op2}$ rounded to the direction *rnd*. Just decreases the exponent by *op2* when *rop* and *op1* are identical. Return zero when *rop*=*op1*, a positive value when *rop*>*op1*, and a negative value when *rop*<*op1*. Note: The `mpfr_div_2exp` function is defined for compatibility reasons; you should use `mpfr_div_2ui` (or `mpfr_div_2si`) instead.

5.8 Comparison Functions

int **mpfr_cmp** (mpfr_t *op1*, mpfr_t *op2*) Function
int **mpfr_cmp_ui** (mpfr_t *op1*, unsigned long int *op2*) Function
int **mpfr_cmp_si** (mpfr_t *op1*, signed long int *op2*) Function

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$. Both *op1* and *op2* are considered to their full own precision, which may differ. In case *op1* and *op2* are of same sign but different, the absolute value returned is one plus the absolute difference of their exponents. It is not allowed that one of the operands is NaN (Not-a-Number).

int **mpfr_cmp_ui_2exp** (mpfr_t *op1*, unsigned long int *op2*, int *e*) Function
int **mpfr_cmp_si_2exp** (mpfr_t *op1*, long int *op2*, int *e*) Function

Compare *op1* and *op2* multiplied by two to the power *e*.

int **mpfr_eq** (mpfr_t *op1*, mpfr_t *op2*, unsigned long int op3) Function

Return non-zero if the first *op3* bits of *op1* and *op2* are equal, zero otherwise. I.e., tests if *op1* and *op2* are approximately equal.

int **mpfr_nan_p** (mpfr_t *op*) Function

Return non-zero if *op* is Not-a-Number (NaN), zero otherwise.

int **mpfr_inf_p** (mpfr_t *op*) Function

Return non-zero if *op* is plus or minus infinity, zero otherwise.

`int` **mpfr_number_p** (`mpfr_t` *op*) Function

Return non-zero if *op* is an ordinary number, i.e. neither Not-a-Number nor plus or minus infinity.

`void` **mpfr_reldiff** (`mpfr_t` *rop*, `mpfr_t` *op1*, `mpfr_t` *op2*, `mp_rnd_t` *rnd*) Function

Compute the relative difference between *op1* and *op2* and store the result in *rop*. This function does not guarantee the exact rounding on the relative difference; it just computes abs(*op1*−*op2*)/*op1*, using the rounding mode *rnd* for all operations.

`int` **mpfr_sgn** (`mpfr_t` *op*) Function

Return a positive value if $op > 0$, zero if $op = 0$, and a negative value if $op < 0$. Its result is not specified when *op* is NaN (Not-a-Number).

5.9 Special Functions

`int` **mpfr_log** (`mpfr_t` *rop*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) Function

Set *rop* to the natural logarithm of *op*, rounded to the direction *rnd* with the precision of *rop*. Return zero when the result is exact (this occurs in fact only when *op* is 0, 1, or +infinity) and a non-zero value otherwise (except for rounding to nearest, the sign of the return value is that of *rop*-log(*op*).

`int` **mpfr_exp** (`mpfr_t` *rop*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) Function

Set *rop* to the exponential of *op*, rounded to the direction *rnd* with the precision of *rop*. Return zero when the result is exact (this occurs in fact only when *op* is -infinity, 0, or +infinity), a positive value when the result is greater than the exponential of *op*, and a negative value when it is smaller.

`int` **mpfr_exp2** (`mpfr_t` *rop*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) Function

Set *rop* to 2 power of *op*, rounded to the direction *rnd* with the precision of *rop*. Return zero iff the result is exact (this occurs in fact only when *op* is -infinity, 0, or +infinity), a positive value when the result is greater than the exponential of *op*, and a negative value when it is smaller.

`int` **mpfr_cos** (`mpfr_t` *cop*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) Function
`int` **mpfr_sin** (`mpfr_t` *sop*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) Function
`int` **mpfr_tan** (`mpfr_t` *top*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) Function

Set *cop* to the cosine of *op*, *sop* to the sine of *op*, *top* to the tangent of *op*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact (this occurs in fact only when *op* is 0 i.e. the sine is 0, the cosine is 1, and the tangent is 0).

`int` **mpfr_sin_cos** (`mpfr_t` *sop*, `mpfr_t` *cop*, `mpfr_t` *op*, `mp_rnd_t` *rnd*) Function

Set simultaneously *sop* to the sine of *op* and *cop* to the cosine of *op*, rounded to the direction *rnd* with their corresponding precisions. Return 0 iff both results are exact.

int **mpfr_acos** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_asin** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_atan** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to the arc-cosine, arc-sine or arc-tangent of *op*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact.

int **mpfr_cosh** (mpfr_t *cop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_sinh** (mpfr_t *sop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_tanh** (mpfr_t *top*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *cop* to the hyperbolic cosine of *op*, *sop* to the hyperbolic sine of *op*, *top* to the hyperbolic tangent of *op*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact (this occurs in fact only when *op* is 0 i.e. the result is 1).

int **mpfr_acosh** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_asinh** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_atanh** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to the inverse hyperbolic cosine, sine or tangent of *op*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact.

int **mpfr_fac_ui** (mpfr_t *rop*, unsigned long int *op*, mp_rnd_t *rnd*) Function

Set *rop* to the factorial of the unsigned long int *op*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact.

int **mpfr_log1p** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to the logarithm of one plus *op*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact (this occurs in fact only when *op* is 0 i.e. the result is 0).

int **mpfr_expm1** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to the exponential of *op* minus one, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact (this occurs in fact only when *op* is 0 i.e. the result is 0).

int **mpfr_log2** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_log10** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to the log[t] (t=2 or 10)(log x / log t) of *op*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact (this occurs in fact only when *op* is 1 i.e. the result is 0).

int **mpfr_fma** (mpfr_t *rop*, mpfr_t *opx*,mpfr_t *opy*,mpfr_t *opz*, mp_rnd_t *rnd*) Function

Set *rop* to *opx* * *opy* + *opz*, rounded to the direction *rnd* with the precision of *rop*. Return 0 iff the result is exact, a positive value if *rop* is larger than *opx* * *opy* + *opz*, and a negative value otherwise.

void **mpfr_agm** (mpfr_t *rop*, mpfr_t *op1*, mpfr_t *op2*, mp_rnd_t *rnd*) Function

Set *rop* to the arithmetic-geometric mean of *op1* and *op2*, rounded to the direction *rnd* with the precision of *rop*.

void **mpfr_const_log2** (mpfr_t *rop*, mp_rnd_t *rnd*) Function

Set *rop* to the logarithm of 2 rounded to the direction *rnd* with the precision of *rop*. This function stores the computed value to avoid another calculation if a lower or equal precision is requested.

void **mpfr_const_pi** (mpfr_t *rop*, mp_rnd_t *rnd*) Function

Set *rop* to the value of Pi rounded to the direction *rnd* with the precision of *rop*. This function uses the Borwein, Borwein, Plouffe formula which directly gives the expansion of Pi in base 16.

void **mpfr_const_euler** (mpfr_t *rop*, mp_rnd_t *rnd*) Function

Set *rop* to the value of Euler's constant 0.577... rounded to the direction *rnd* with the precision of *rop*.

5.10 Input and Output Functions

Functions that perform input from a standard input/output stream, and functions that output to a standard input/output stream. Passing a null pointer for a *stream* argument to any of these functions will make them read from `stdin` and write to `stdout`, respectively.

When using any of these functions, it is a good idea to include '`stdio.h`' before '`mpfr.h`', since that will allow '`mpfr.h`' to define prototypes for these functions.

size_t **mpfr_out_str** (FILE **stream*, int *base*, size_t *n_digits*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Output *op* on stdio stream *stream*, as a string of digits in base *base*, rounded to direction *rnd*. The base may vary from 2 to 36. Print at most *n_digits* significant digits, or if *n_digits* is 0, the maximum number of digits accurately representable by *op*.

In addition to the significant digits, a decimal point at the right of the first digit and a trailing exponent, in the form '`eNNN`', are printed. If *base* is greater than 10, '`@`' will be used instead of '`e`' as exponent delimiter.

Return the number of bytes written, or if an error occurred, return 0.

size_t **mpfr_inp_str** (mpfr_t *rop*, FILE **stream*, int *base*, mp_rnd_t *rnd*) Function

Input a string in base *base* from stdio stream *stream*, rounded in direction *rnd*, and put the read float in *rop*. The string is of the form '`M@N`' or, if the base is 10 or less, alternatively '`MeN`' or '`MEN`'. '`M`' is the mantissa and '`N`' is the exponent. The mantissa is always in the specified base. The exponent is in decimal.

The argument *base* may be in the range 2 to 36.

Unlike the corresponding `mpz` function, the base will not be determined from the leading characters of the string if *base* is 0. This is so that numbers like '`0.23`' are not interpreted as octal.

Return the number of bytes read, or if an error occurred, return 0.

void **mpfr_print_binary** (mpfr_t *float*) Function

Output *float* on stdout in raw binary format (the exponent is in decimal, yet). The last bits from the least significant limb which do not belong to the mantissa are printed between square brackets; they should always be zero.

5.11 Miscellaneous Functions

int **mpfr_rint** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function
int **mpfr_ceil** (mpfr_t *rop*, mpfr_t *op*) Function
int **mpfr_floor** (mpfr_t *rop*, mpfr_t *op*) Function
int **mpfr_round** (mpfr_t *rop*, mpfr_t *op*) Function
int **mpfr_trunc** (mpfr_t *rop*, mpfr_t *op*) Function

Set *rop* to *op* rounded to an integer. `mpfr_ceil` rounds to the next higher representable integer, `mpfr_floor` to the next lower, `mpfr_round` to the nearest representable integer, rounding halfway cases away from zero, and `mpfr_trunc` to the representable integer towards zero. `mpfr_rint` behaves like one of these four functions, depending on the rounding mode. The returned value is zero when the result is exact, positive when it is greater than the original value of *op*, and negative when it is smaller. More precisely, the returned value is 0 when *op* is an integer representable in *rop*, 1 or -1 when *op* is an integer that is not representable in *rop*, 2 or -2 when *op* is not an integer.

void **mpfr_urandomb** (mpfr_t *rop*, gmp_randstate_t *state*) Function

Generate a uniformly distributed random float in the interval 0 <= X < 1.

void **mpfr_random** (mpfr_t *rop*) Function

Generate a uniformly distributed random float in the interval 0 <= X < 1.

void **mpfr_random2** (mpfr_t *rop*, mp_size_t *max_size*, mp_exp_t *max_exp*) Function

Generate a random float of at most *max_size* limbs, with long strings of zeros and ones in the binary representation. The exponent of the number is in the interval $-exp$ to *exp*. This function is useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs. Negative random numbers are generated when *max_size* is negative.

5.12 Internals

These types and functions were mainly designed for the implementation of `mpfr`, but may be useful for users too. However no upward compatibility is guaranteed. You need to include `mpfr-impl.h` to use them.

The `mpfr_t` type consists of four fields. The `_mpfr_prec` field is used to store the precision of the variable (in bits); this is not less than 2.

The `_mpfr_size` field is used to store the number of allocated limbs, with the high bits reserved to store the sign (bit 31), the NaN flag (bit 30), and the Infinity flag (bit 29); thus bits 0 to 28 remain for the number of allocated limbs, with a maximal value of 536870911. A NaN is indicated by the NaN flag set, and no other fields are relevant. An Infinity is indicated by the NaN flag clear and the Inf flag set; the sign bit of an Infinity indicates the sign, the limb data and the exponent are not relevant.

The `_mpfr_exp` field stores the exponent. An exponent of 0 means a radix point just above the most significant limb. Non-zero values are a multiplier $2\hat{}n$ relative to that point.

Finally, the `_mpfr_d` is a pointer to the limbs, least significant limbs stored first. The number zero is represented with its most significant limb set to zero, i.e. `_mpfr_d[n-1]` where n=ceil(`_mpfr_prec`/`BITS_PER_MP_LIMB`). The number of limbs in use is controlled by `_mpfr_prec`, namely ceil(`_mpfr_prec`/`BITS_PER_MP_LIMB`). Zero is represented by the most significant limb being zero, other limb data and the exponent are not relevant ("not relevant" implies that the corresponding objects may contain invalid values, thus should not be evaluated even if they are not taken into account). Non-zero values always have the most significant bit of the most significant limb set to 1. When the precision is not a whole number of limbs, the excess bits at the low end of the data are zero. When the precision has been lowered by `mpfr_set_prec`, the space allocated at `_mpfr_d` remains as given by `_mpfr_size`, but `_mpfr_prec` indicates how much of that space is actually used.

`int` **mpfr_add_one_ulp** (`mpfr_t` *x*, `mp_rnd_t` *rnd*) Function

Add one unit in last place (ulp) to *x* if *x* is finite and positive, subtract one ulp if *x* is finite and negative; otherwise, *x* is not changed. The return value is zero unless an overflow occurs, in which case the `mpfr_add_one_ulp` function behaves like a conventional addition.

`int` **mpfr_sub_one_ulp** (`mpfr_t` *x*, `mp_rnd_t` *rnd*) Function

Subtract one ulp to *x* if *x* is finite and positive, add one ulp if *x* is finite and negative; otherwise, *x* is not changed. The return value is zero unless an underflow occurs, in which case the `mpfr_sub_one_ulp` function behaves like a conventional subtraction.

`int` **mpfr_can_round** (`mpfr_t` *b*, `mp_exp_t` *err*, `mp_rnd_t` *rnd1*, `mp_rnd_t` *rnd2*, `mp_prec_t` *prec*) Function

Assuming *b* is an approximation of an unknown number *x* in direction *rnd1* with error at most two to the power E(b)-*err* where E(b) is the exponent of *b*, returns 1 if one is able to round exactly *x* to precision *prec* with direction *rnd2*, and 0 otherwise. This function **does not modify** its arguments.

Contributors

The main developers consist of Guillaume Hanrot, Vincent Lefvre and Paul Zimmermann.

We would like to thank Jean-Michel Muller and Joris van der Hoeven for very fruitful discussions at the beginning of that project, Torbjorn Granlund and Kevin Ryde for their help about design issues and their suggestions for an easy integration into GNU MP, and Nathalie Revol for her careful reading of this documentation.

Sylvie Boldo from ENS-Lyon, France, contributed the functions `mpfr_agm` and `mpfr_log`. Emmanuel Jeandel, from ENS-Lyon too, contributed the generic hypergeometric code in `generic.c`, as well as the `mpfr_exp3`, a first implementation of the sine and cosine, and improved versions of `mpfr_const_log2` and `mpfr_const_pi`. Mathieu Dutour contributed the functions `mpfr_atan` and `mpfr_asin`, David Daney contributed the hyperbolic and inverse hyperbolic functions, the base-2 exponential, and the factorial function. Fabrice Rouillier contributed the original version of '`mul_ui.c`', the '`gmp_op.c`' file, and helped to the Windows porting.

References

- Torbjorn Granlund, "GNU MP: The GNU Multiple Precision Arithmetic Library", version 4.0.1, 2001.
- IEEE standard for binary floating-point arithmetic, Technical Report ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985: IEEE Standards Board; approved July 26, 1985: American National Standards Institute, 18 pages.
- Donald E. Knuth, "The Art of Computer Programming", vol 2, "Seminumerical Algorithms", 2nd edition, Addison-Wesley, 1981.

Concept Index

Function and Type Index

M

Table of Contents