

**gmp**

**COLLABORATORS**

	<i>TITLE :</i> gmp		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		January 15, 2023	

**REVISION HISTORY**

<i>NUMBER</i>	<i>DATE</i>	<i>DESCRIPTION</i>	<i>NAME</i>

# Contents

<b>1</b>	<b>gmp</b>	<b>1</b>
1.1	gmp.guide	1
1.2	gmp.guide/Copying	2
1.3	gmp.guide/Introduction to MP	2
1.4	gmp.guide/Installing MP	3
1.5	gmp.guide/MP Basics	6
1.6	gmp.guide/Reporting Bugs	9
1.7	gmp.guide/Integer Functions	9
1.8	gmp.guide/Initializing Integers	10
1.9	gmp.guide/Assigning Integers	11
1.10	gmp.guide/Simultaneous Integer Init & Assign	12
1.11	gmp.guide/Converting Integers	12
1.12	gmp.guide/Integer Arithmetic	13
1.13	gmp.guide/Comparison Functions	18
1.14	gmp.guide/Integer Logic and Bit Fiddling	18
1.15	gmp.guide/I-O of Integers	19
1.16	gmp.guide/Miscellaneous Integer Functions	20
1.17	gmp.guide/Rational Number Functions	21
1.18	gmp.guide/Initializing Rationals	22
1.19	gmp.guide/Assigning Rationals	22
1.20	gmp.guide/Comparing Rationals	23
1.21	gmp.guide/Applying Integer Functions	24
1.22	gmp.guide/Miscellaneous Rational Functions	24
1.23	gmp.guide/Floating-point Functions	25
1.24	gmp.guide/Initializing Floats	26
1.25	gmp.guide/Assigning Floats	27
1.26	gmp.guide/Simultaneous Float Init & Assign	28
1.27	gmp.guide/Converting Floats	28
1.28	gmp.guide/Float Arithmetic	29
1.29	gmp.guide/Float Comparison	30

---

---

1.30	gmp.guide/I-O of Floats . . . . .	30
1.31	gmp.guide/Miscellaneous Float Functions . . . . .	31
1.32	gmp.guide/Low-level Functions . . . . .	32
1.33	gmp.guide/BSD Compatible Functions . . . . .	38
1.34	gmp.guide/Custom Allocation . . . . .	40
1.35	gmp.guide/Contributors . . . . .	41
1.36	gmp.guide/References . . . . .	42
1.37	gmp.guide/Concept Index . . . . .	42
1.38	gmp.guide/Function Index . . . . .	45

---

# Chapter 1

## gmp

### 1.1 gmp.guide

GNU MP

\*\*\*\*\*

This manual documents how to install and use the GNU multiple precision arithmetic library, version 2.0.2.

Copying

GMP Copying Conditions (LGPL).

Introduction to MP

Brief introduction to GNU MP.

Installing MP

How to configure and compile the MP library.

MP Basics

What every MP user should now.

Reporting Bugs

How to usefully report bugs.

Integer Functions

Functions for arithmetic on signed integers.

Rational Number Functions

Functions for arithmetic on rational numbers.

Floating-point Functions

Functions for arithmetic on floats.

Low-level Functions

Fast functions for natural numbers.

BSD Compatible Functions

All functions found in BSD MP.

---

Custom Allocation  
How to customize the internal allocation.

Contributors

References

Concept Index

Function Index

## 1.2 gmp.guide/Copying

GNU MP 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 GNU MP 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 GNU MP 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 GNU MP library are found in the Library General Public License that accompany the source code.

## 1.3 gmp.guide/Introduction to MP

Introduction to GNU MP

\*\*\*\*\*

---

GNU MP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. MP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

The speed of MP is achieved by using fullwords as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

There is carefully optimized assembly code for these CPUs: DEC Alpha, Amd 29000, HPPA 1.0 and 1.1, Intel Pentium and generic x86, Intel i960, Motorola MC68000, MC68020, MC88100, and MC88110, Motorola/IBM PowerPC, National NS32000, IBM POWER, MIPS R3000, R4000, SPARCv7, SuperSPARC, generic SPARCv8, and DEC VAX. Some optimizations also for ARM, Clipper, IBM ROMP (RT), and Pyramid AP/XP.

This version of MP is released under a more liberal license than previous versions. It is now permitted to link MP to non-free programs, as long as MP source code is provided when distributing the non-free program.

How to use this Manual  
=====

Everyone should read See  
MP Basics  
. If you need to install the  
library yourself, you need to read See  
Installing MP  
, too.

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

## 1.4 gmp.guide/Installing MP

Installing MP

\*\*\*\*\*

To build MP, you first have to configure it for your CPU and operating system. 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.

---

(If you're on an MS-DOS machine, you can build MP using 'make.bat'. It requires that djgpp is installed. It does not require configuration, nor is 'make' needed; 'make.bat' both configures and builds the library.)

Here are the steps needed to install the library on Unix systems:

1. In most cases, './configure --target=cpu-vendor-os', should work both for native and cross-compilation. If you get error messages, your machine might not be supported.

If you want to compile in a separate object directory, cd to that directory, and prefix the configure command with the path to the MP source directory. Not all 'make' programs have the necessary features to support this. In particular, SunOS and Solaris 'make' have bugs that makes them unable to build from a separate object directory. Use GNU 'make' instead.

In addition to the standard cpu-vendor-os tuples, MP recognizes sparc8 and supersparc as valid CPU names. Specifying these CPU names for relevant systems will improve performance significantly.

In general, if you want a library that runs as fast as possible, you should make sure you configure MP for the exact CPU type your system uses.

If you have 'gcc' in your 'PATH', it will be used by default. To override this, pass '-with-gcc=no' to 'configure'.

2. 'make'

This will compile MP, and create a library archive file 'libgmp.a' in the working directory.

3. 'make check'

This will make sure MP was built correctly. If you get error messages, please report this to 'bug-gmp@prep.ai.mit.edu'. (See

Reporting Bugs  
, for information on what to include in useful bug reports.)

4. 'make install'

This will copy the file 'gmp.h' and 'libgmp.a', as well as the info files, to '/usr/local' (or if you passed the '--prefix' option to 'configure', to the directory given as argument to '--prefix').

If you wish to build and install the BSD MP compatible functions, use 'make libmp.a' and 'make install-bsdmp'.

There are some other useful make targets:

\* 'doc'

---



Create a DVI version of the manual, in 'gmp.dvi' and a set of info files, in 'gmp.info', 'gmp.info-1', 'gmp.info-2', etc.

\* 'ps'

Create a Postscript version of the manual, in 'gmp.ps'.

\* 'html'

Create a HTML version of the manual, in 'gmp.html'.

\* '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'.

#### Known Build Problems

=====

GCC 2.7.2 (as well as 2.6.3) for the RS/6000 and PowerPC can not be used to compile MP, due to a bug in GCC. If you want to use GCC for these machines, you need to apply the patch below to GCC, or use a later version of the compiler.

If you are on a Sequent Symmetry, use the GNU assembler instead of the system's assembler, since the latter has serious bugs.

The system compiler on NeXT is a massacred and old gcc, even if the compiler calls itself 'cc'. This compiler cannot be used to build MP. You need to get a real gcc, and install that before you compile MP. (NeXT might have fixed this in newer releases of their system.)

The system C compiler under SunOS 4 has a bug that makes it miscompile mpq/get\_d.c. This will make 'make check' fail.

Please report other problems to 'bug-gmp@prep.ai.mit.edu'. See

#### Reporting Bugs

.

Patch to apply to GCC 2.6.3 and 2.7.2:

```
*** config/rs6000/rs6000.md Sun Feb 11 08:22:11 1996
--- config/rs6000/rs6000.md.new Sun Feb 18 03:33:37 1996
*****
*** 920,926 ****
      (set (match_operand:SI 0 "gpc_reg_operand" "=r")
          (not:SI (match_dup 1)))
      ""
```

```

!   "nor. %0,%2,%1"
    [(set_attr "type" "compare")]

(define_insn ""
--- 920,926 ----
    (set (match_operand:SI 0 "gpc_reg_operand" "=r")
        (not:SI (match_dup 1))))
    ""
!   "nor. %0,%1,%1"
    [(set_attr "type" "compare")]

(define_insn ""

```

## 1.5 gmp.guide/MP Basics

### MP Basics

\*\*\*\*\*

All declarations needed to use MP are collected in the include file `'gmp.h'`. It is designed to work with both C and C++ compilers.

### Nomenclature and Types

=====

In this manual, "integer" usually means a multiple precision integer, as defined by the MP library. The C data type for such integers is `'mpz_t'`. Here are some examples of how to declare such integers:

```

mpz_t sum;

struct foo { mpz_t x, y; };

mpz_t vec[20];

```

"Rational number" means a multiple precision fraction. The C data type for these fractions is `'mpq_t'`. For example:

```

mpq_t quotient;

```

"Floating point number" or "Float" for short, is an arbitrary precision mantissa with an limited precision exponent. The C data type for such objects is `'mpf_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'`.

### Function Classes

=====

There are six classes of functions in the MP library:

1. Functions for signed integer arithmetic, with names beginning with `'mpz_'`. The associated type is `'mpz_t'`. There are about 100 functions in this class.
2. Functions for rational number arithmetic, with names beginning with `'mpq_'`. The associated type is `'mpq_t'`. There are about 20 functions in this class, but the functions in the previous class can be used for performing arithmetic on the numerator and denominator separately.
3. Functions for floating-point arithmetic, with names beginning with `'mpf_'`. The associated type is `'mpf_t'`. There are about 50 functions in this class.
4. Functions compatible with Berkeley MP, such as `'itom'`, `'madd'`, and `'mult'`. The associated type is `'MINT'`.
5. Fast low-level functions that operate on natural numbers. These are used by the functions in the preceding groups, and you can also call them directly from very time-critical user programs. These functions' names begin with `'mpn_'`. There are about 30 (hard-to-use) functions in this class.

The associated type is array of `'mp_limb_t'`.

6. Miscellaneous functions. Functions for setting up custom allocation.

#### MP Variable Conventions

=====

As a general rule, all MP functions expect output arguments before input arguments. This notation is based on an analogy with the assignment operator. (The BSD MP compatibility functions disobey this rule, having the output argument(s) last.)

MP allows you to use the same variable for both input and output in the same expression. For example, the main function for integer multiplication, `'mpz_mul'`, can be used like this: `'mpz_mul (x, x, x)'`. This computes the square of X and puts the result back in X.

Before you can assign to an MP 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. Which function to use depends on the type of variable. See the chapters on integer functions, rational number functions, and floating-point functions for details.

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 MP variables. All functions in MP automatically allocate additional space when a variable does not already have enough space. They do not, however, reduce the space when a smaller number is stored in the object. Most of the time, this policy is best, since it avoids frequent re-allocation.

#### Useful Macros and Constants

=====

- Global Constant: `const int mp_bits_per_limb`  
The number of bits per limb.
- Macro: `__GNU_MP_VERSION`
- Macro: `__GNU_MP_VERSION_MINOR`  
The major and minor MP version, respectively, as integers.

#### Compatibility with Version 1.x

=====

This version of MP is upward compatible with previous versions of MP, with a few exceptions.

1. Integer division functions round the result differently. The old functions (`'mpz_div'`, `'mpz_divmod'`, `'mpz_mdiv'`, `'mpz_mdivmod'`, etc) now all use floor rounding (i.e., they round the quotient to  $-\infty$ ). There are a lot of new functions for integer division, giving the user better control over the rounding.
2. The function `'mpz_mod'` now compute the true `*mod*` function.
3. The functions `'mpz_powm'` and `'mpz_powm_ui'` now use `*mod*` for reduction.
4. The assignment functions for rational numbers do no longer canonicalize their results. In the case a non-canonical result could arise from an assignment, the user need to insert an explicit call to `'mpq_canonicalize'`. This change was made for efficiency.
5. Output generated by `'mpz_out_raw'` in this release cannot be read by `'mpz_inp_raw'` in previous releases. This change was made for making the file format truly portable between machines with different word sizes.
6. Several `'mpn'` functions have changed. But they were intentionally undocumented in previous releases.
7. The functions `'mpz_cmp_ui'`, `'mpz_cmp_si'`, and `'mpq_cmp_ui'` are now implemented as macros, and thereby sometimes evaluate their arguments multiple times.
8. The functions `'mpz_pow_ui'` and `'mpz_ui_pow_ui'` now yield 1 for  $0^0$ . (In version 1, they yielded 0.)

#### Getting the Latest Version of MP

---

=====

The latest version of the MP library is available by anonymous ftp from from 'prep.ai.mit.edu'. The file name is '/pub/gnu/gmp-M.N.tar.gz'. Many sites around the world mirror 'prep'; please use a mirror site near you.

## 1.6 gmp.guide/Reporting Bugs

### Reporting Bugs

\*\*\*\*\*

If you think you have found a bug in the MP library, please investigate it 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.

It is not uncommon that an observed problem is actually due to a bug in the compiler used when building MP; the MP code tends to explore interesting corners in compilers. Therefore, please include compiler version information in your bug report. This can be extracted using 'what 'which cc'', 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: 'bug-gmp@prep.ai.mit.edu'.

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.

## 1.7 gmp.guide/Integer Functions

### Integer Functions

\*\*\*\*\*

This chapter describes the MP functions for performing integer arithmetic. These functions start with the prefix 'mpz\_'.

---

Arbitrary precision integers are stored in objects of type `'mpz_t'`.

Initializing Integers  
Assigning Integers  
Simultaneous Integer Init & Assign  
Converting Integers  
Integer Arithmetic  
Comparison Functions  
Integer Logic and Bit Fiddling  
I-O of Integers  
Miscellaneous Integer Functions

## 1.8 gmp.guide/Initializing Integers

Initialization and Assignment Functions  
=====

The functions for integer arithmetic assume that all integer objects are initialized. You do that by calling the function `'mpz_init'`.

- Function: `void mpz_init (mpz_t INTEGER)`  
Initialize `INTEGER` with limb space and set the initial numeric value to 0. Each variable should normally only be initialized once, or at least cleared out (using `'mpz_clear'`) between each initialization.

Here is an example of using `'mpz_init'`:

```
{
  mpz_t integ;
  mpz_init (integ);
  ...
  mpz_add (integ, ...);
  ...
  mpz_sub (integ, ...);

  /* Unless the program is about to exit, do ... */
  mpz_clear (integ);
}
```

As you can see, you can store new values any number of times, once an object is initialized.

---

- Function: void mpz\_clear (mpz\_t INTEGER)  
Free the limb space occupied by INTEGER. Make sure to call this function for all 'mpz\_t' variables when you are done with them.
- Function: void \* \_mpz\_realloc (mpz\_t INTEGER, mp\_size\_t NEW\_ALLOC)  
Change the limb space allocation to NEW\_ALLOC limbs. This function is not normally called from user code, but it can be used to give memory back to the heap, or to increase the space of a variable to avoid repeated automatic re-allocation.
- Function: void mpz\_array\_init (mpz\_t INTEGER\_ARRAY[], size\_t ARRAY\_SIZE, mp\_size\_t FIXED\_NUM\_BITS)  
Allocate \*fixed\* limb space for all ARRAY\_SIZE integers in INTEGER\_ARRAY. The fixed allocation for each integer in the array is enough to store FIXED\_NUM\_BITS. If the fixed space will be insufficient for storing the result of a subsequent calculation, the result is unpredictable.

This function is useful for decreasing the working set for some algorithms that use large integer arrays.

There is no way to de-allocate the storage allocated by this function. Don't call 'mpz\_clear'!

## 1.9 gmp.guide/Assigning Integers

### Assignment Functions

---

These functions assign new values to already initialized integers (see

Initializing Integers  
).

- Function: void mpz\_set (mpz\_t ROP, mpz\_t OP)
- Function: void mpz\_set\_ui (mpz\_t ROP, unsigned long int OP)
- Function: void mpz\_set\_si (mpz\_t ROP, signed long int OP)
- Function: void mpz\_set\_d (mpz\_t ROP, double OP)
- Function: void mpz\_set\_q (mpz\_t ROP, mpq\_t OP)
- Function: void mpz\_set\_f (mpz\_t ROP, mpf\_t OP)  
Set the value of ROP from OP.
- Function: int mpz\_set\_str (mpz\_t ROP, char \*STR, int BASE)  
Set the value of ROP from STR, a '\0'-terminated C string in base BASE. White space is allowed in the string, and is simply ignored. The base may vary from 2 to 36. If BASE is 0, the actual base is determined from the leading characters: if the first two characters are '0x' or '0X', hexadecimal is assumed, otherwise if the first character is '0', octal is assumed, otherwise decimal is assumed.

This function returns 0 if the entire string up to the '\0' is a valid number in base BASE. Otherwise it returns -1.

---

## 1.10 gmp.guide/Simultaneous Integer Init & Assign

### Combined Initialization and Assignment Functions

---

For convenience, MP provides a parallel series of initialize-and-set functions which initialize the output and then store the value there. These functions' names have the form 'mpz\_init\_set...'

Here is an example of using one:

```
{
  mpz_t pie;
  mpz_init_set_str (pie, "3141592653589793238462643383279502884", 10);
  ...
  mpz_sub (pie, ...);
  ...
  mpz_clear (pie);
}
```

Once the integer has been initialized by any of the 'mpz\_init\_set...' functions, it can be used as the source or destination operand for the ordinary integer functions. Don't use an initialize-and-set function on a variable already initialized!

- Function: void mpz\_init\_set (mpz\_t ROP, mpz\_t OP)
- Function: void mpz\_init\_set\_ui (mpz\_t ROP, unsigned long int OP)
- Function: void mpz\_init\_set\_si (mpz\_t ROP, signed long int OP)
- Function: void mpz\_init\_set\_d (mpz\_t ROP, double OP)
- Initialize ROP with limb space and set the initial numeric value from OP.
- Function: int mpz\_init\_set\_str (mpz\_t ROP, char \*STR, int BASE)
- Initialize ROP and set its value like 'mpz\_set\_str' (see its documentation above for details).

If the string is a correct base BASE number, the function returns 0; if an error occurs it returns -1. ROP is initialized even if an error occurs. (I.e., you have to call 'mpz\_clear' for it.)

## 1.11 gmp.guide/Converting Integers

### Conversion Functions

---

This section describes functions for converting arbitrary precision integers to standard C types. Functions for converting \*to\* arbitrary precision integers are described in See  
Assigning Integers

---



and See

I-O of Integers

.

- Function: unsigned long int mpz\_get\_ui (mpz\_t OP)  
Return the least significant part from OP. This function combined with  
'mpz\_tdiv\_q\_2exp(..., OP, CHAR\_BIT\*sizeof(unsigned long int))' can  
be used to extract the limbs of an integer.

- Function: signed long int mpz\_get\_si (mpz\_t OP)  
If OP fits into a 'signed long int' return the value of OP.  
Otherwise return the least significant part of OP, with the same  
sign as OP.

If OP is too large to fit in a 'signed long int', the returned  
result is probably not very useful.

- Function: double mpz\_get\_d (mpz\_t OP)  
Convert OP to a double.

- Function: char \* mpz\_get\_str (char \*STR, int BASE, mpz\_t OP)  
Convert OP to a string of digits in base BASE. The base may vary  
from 2 to 36.

If STR is NULL, space for the result string is allocated using the  
default allocation function, and a pointer to the string is  
returned.

If STR is not NULL, it should point to a block of storage enough  
large for the result. To find out the right amount of space to  
provide for STR, use 'mpz\_sizeinbase (OP, BASE) + 2'. The two  
extra bytes are for a possible minus sign, and for the terminating  
null character.

## 1.12 gmp.guide/Integer Arithmetic

Arithmetic Functions

=====

- Function: void mpz\_add (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
- Function: void mpz\_add\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int  
OP2)  
Set ROP to OP1 + OP2.

- Function: void mpz\_sub (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
- Function: void mpz\_sub\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int  
OP2)  
Set ROP to OP1 - OP2.

- Function: void mpz\_mul (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
- Function: void mpz\_mul\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int

- OP2)  
Set ROP to OP1 times OP2.
- Function: void mpz\_mul\_2exp (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to OP1 times 2 raised to OP2. This operation can also be defined as a left shift, OP2 steps.
  - Function: void mpz\_neg (mpz\_t ROP, mpz\_t OP)  
Set ROP to -OP.
  - Function: void mpz\_abs (mpz\_t ROP, mpz\_t OP)  
Set ROP to the absolute value of OP.
  - Function: void mpz\_fac\_ui (mpz\_t ROP, unsigned long int OP)  
Set ROP to OP!, the factorial of OP.

#### Division functions

-----

Division is undefined if the divisor is zero, and passing a zero divisor to the divide or modulo functions, as well passing a zero mod argument to the 'mpz\_powm' and 'mpz\_powm\_ui' functions, will make these functions intentionally divide by zero. This gives the user the possibility to handle arithmetic exceptions in these functions in the same manner as other arithmetic exceptions.

There are three main groups of division functions:

- \* Functions that truncate the quotient towards 0. The names of these functions start with 'mpz\_tdiv'. The 't' in the name is short for 'truncate'.
- \* Functions that round the quotient towards -infinity. The names of these routines start with 'mpz\_fdiv'. The 'f' in the name is short for 'floor'.
- \* Functions that round the quotient towards +infinity. The names of these routines start with 'mpz\_cdiv'. The 'c' in the name is short for 'ceil'.

For each rounding mode, there are a couple of variants. Here 'q' means that the quotient is computed, while 'r' means that the remainder is computed. Functions that compute both the quotient and remainder have 'qr' in the name.

- Function: void mpz\_tdiv\_q (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)
- Function: void mpz\_tdiv\_q\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to [OP1/OP2]. The quotient is truncated towards 0.
- Function: void mpz\_tdiv\_r (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)
- Function: void mpz\_tdiv\_r\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to (OP1 - [OP1/OP2] \* OP2). Unless the remainder is zero, it has the same sign as the dividend.
- Function: void mpz\_tdiv\_qr (mpz\_t ROP1, mpz\_t ROP2, mpz\_t OP1, mpz\_t

- OP2)
- Function: void mpz\_tdiv\_qr\_ui (mpz\_t ROP1, mpz\_t ROP2, mpz\_t OP1, unsigned long int OP2)  
Divide OP1 by OP2 and put the quotient in ROP1 and the remainder in ROP2. The quotient is rounded towards 0. Unless the remainder is zero, it has the same sign as the dividend.  
  
If ROP1 and ROP2 are the same variable, the results are undefined.
  - Function: void mpz\_fdiv\_q (mpz\_t ROP1, mpz\_t OP1, mpz\_t OP2)
  - Function: void mpz\_fdiv\_q\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to OP1/OP2. The quotient is rounded towards -infinity.
  - Function: void mpz\_fdiv\_r (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)
  - Function: unsigned long int mpz\_fdiv\_r\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Divide OP1 by OP2 and put the remainder in ROP. Unless the remainder is zero, it has the same sign as the divisor.  
  
For 'mpz\_fdiv\_r\_ui' the remainder is small enough to fit in an 'unsigned long int', and is therefore returned.
  - Function: void mpz\_fdiv\_qr (mpz\_t ROP1, mpz\_t ROP2, mpz\_t OP1, mpz\_t OP2)
  - Function: unsigned long int mpz\_fdiv\_qr\_ui (mpz\_t ROP1, mpz\_t ROP2, mpz\_t OP1, unsigned long int OP2)  
Divide OP1 by OP2 and put the quotient in ROP1 and the remainder in ROP2. The quotient is rounded towards -infinity. Unless the remainder is zero, it has the same sign as the divisor.  
  
For 'mpz\_fdiv\_qr\_ui' the remainder is small enough to fit in an 'unsigned long int', and is therefore returned.  
  
If ROP1 and ROP2 are the same variable, the results are undefined.
  - Function: unsigned long int mpz\_fdiv\_ui (mpz\_t OP1, unsigned long int OP2)  
This function is similar to 'mpz\_fdiv\_r\_ui', but the remainder is only returned; it is not stored anywhere.
  - Function: void mpz\_cdiv\_q (mpz\_t ROP1, mpz\_t OP1, mpz\_t OP2)
  - Function: void mpz\_cdiv\_q\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to OP1/OP2. The quotient is rounded towards +infinity.
  - Function: void mpz\_cdiv\_r (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)
  - Function: unsigned long int mpz\_cdiv\_r\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Divide OP1 by OP2 and put the remainder in ROP. Unless the remainder is zero, it has the opposite sign as the divisor.  
  
For 'mpz\_cdiv\_r\_ui' the negated remainder is small enough to fit in an 'unsigned long int', and it is therefore returned.
  - Function: void mpz\_cdiv\_qr (mpz\_t ROP1, mpz\_t ROP2, mpz\_t OP1, mpz\_t OP2)
-

- Function: unsigned long int mpz\_cdiv\_qr\_ui (mpz\_t ROP1, mpz\_t ROP2, mpz\_t OP1, unsigned long int OP2)  
Divide OP1 by OP2 and put the quotient in ROP1 and the remainder in ROP2. The quotient is rounded towards +infinity. Unless the remainder is zero, it has the opposite sign as the divisor.  
  
For 'mpz\_cdiv\_qr\_ui' the negated remainder is small enough to fit in an 'unsigned long int', and it is therefore returned.  
  
If ROP1 and ROP2 are the same variable, the results are undefined.
  - Function: unsigned long int mpz\_cdiv\_ui (mpz\_t OP1, unsigned long int OP2)  
Return the negated remainder, similar to 'mpz\_cdiv\_r\_ui'. (The difference is that this function doesn't store the remainder anywhere.)
  - Function: void mpz\_mod (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)
  - Function: unsigned long int mpz\_mod\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to OP1 'mod' OP2. The sign of the divisor is ignored, and the result is always non-negative.  
  
For 'mpz\_mod\_ui' the remainder is small enough to fit in an 'unsigned long int', and is therefore returned.
  - Function: void mpz\_divexact (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
Set ROP to OP1/OP2. This function produces correct results only when it is known in advance that OP2 divides OP1.  
  
Since mpz\_divexact is much faster than any of the other routines that produce the quotient (see [References](#) [Jebelean](#)), it is the best choice for instances in which exact division is known to occur, such as reducing a rational to lowest terms.
  - Function: void mpz\_tdiv\_q\_2exp (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to OP1 divided by 2 raised to OP2. The quotient is rounded towards 0.
  - Function: void mpz\_tdiv\_r\_2exp (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Divide OP1 by (2 raised to OP2) and put the remainder in ROP. Unless it is zero, ROP will have the same sign as OP1.
  - Function: void mpz\_fdiv\_q\_2exp (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Set ROP to OP1 divided by 2 raised to OP2. The quotient is rounded towards -infinity.
  - Function: void mpz\_fdiv\_r\_2exp (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Divide OP1 by (2 raised to OP2) and put the remainder in ROP. The sign of ROP will always be positive.
-

This operation can also be defined as masking of the OP2 least significant bits.

#### Exponentialization Functions

-----

- Function: void mpz\_powm (mpz\_t ROP, mpz\_t BASE, mpz\_t EXP, mpz\_t MOD)
- Function: void mpz\_powm\_ui (mpz\_t ROP, mpz\_t BASE, unsigned long int EXP, mpz\_t MOD)  
Set ROP to (BASE raised to EXP) 'mod' MOD. If EXP is negative, the result is undefined.
- Function: void mpz\_pow\_ui (mpz\_t ROP, mpz\_t BASE, unsigned long int EXP)
- Function: void mpz\_ui\_pow\_ui (mpz\_t ROP, unsigned long int BASE, unsigned long int EXP)  
Set ROP to BASE raised to EXP. The case of  $0^0$  yields 1.

#### Square Root Functions

-----

- Function: void mpz\_sqrt (mpz\_t ROP, mpz\_t OP)  
Set ROP to the truncated integer part of the square root of OP.
- Function: void mpz\_sqrtrem (mpz\_t ROP1, mpz\_t ROP2, mpz\_t OP)  
Set ROP1 to the truncated integer part of the square root of OP, like 'mpz\_sqrt'. Set ROP2 to  $OP - ROP1 * ROP1$ , (i.e., zero if OP is a perfect square).  
  
If ROP1 and ROP2 are the same variable, the results are undefined.
- Function: int mpz\_perfect\_square\_p (mpz\_t OP)  
Return non-zero if OP is a perfect square, i.e., if the square root of OP is an integer. Return zero otherwise.

#### Number Theoretic Functions

-----

- Function: int mpz\_probab\_prime\_p (mpz\_t OP, int REPS)  
If this function returns 0, OP is definitely not prime. If it returns 1, then OP is 'probably' prime. The probability of a false positive is  $(1/4)**REPS$ . A reasonable value of reps is 25.  
  
An implementation of the probabilistic primality test found in Seminumerical Algorithms (see  
References  
Knuth).
  - Function: void mpz\_gcd (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
Set ROP to the greatest common divisor of OP1 and OP2.
  - Function: unsigned long int mpz\_gcd\_ui (mpz\_t ROP, mpz\_t OP1, unsigned long int OP2)  
Compute the greatest common divisor of OP1 and OP2. If ROP is not NULL, store the result there.  
  
If the result is small enough to fit in an 'unsigned long int', it
-

is returned. If the result does not fit, 0 is returned, and the result is equal to the argument OP1. Note that the result will always fit if OP2 is non-zero.

- Function: void mpz\_gcdext (mpz\_t G, mpz\_t S, mpz\_t T, mpz\_t A, mpz\_t B)  
Compute G, S, and T, such that  $AS + BT = G = \text{'gcd'}$  (A, B). If T is NULL, that argument is not computed.
- Function: int mpz\_invert (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
Compute the inverse of OP1 modulo OP2 and put the result in ROP. Return non-zero if an inverse exist, zero otherwise. When the function returns zero, do not assume anything about the value in ROP.
- Function: int mpz\_jacobi (mpz\_t OP1, mpz\_t OP2)
- Function: int mpz\_legendre (mpz\_t OP1, mpz\_t OP2)  
Compute the Jacobi and Legendre symbols, respectively.

## 1.13 gmp.guide/Comparison Functions

### Comparison Functions

=====

- Function: int mpz\_cmp (mpz\_t OP1, mpz\_t OP2)  
Compare OP1 and OP2. Return a positive value if  $OP1 > OP2$ , zero if  $OP1 = OP2$ , and a negative value if  $OP1 < OP2$ .
- Macro: int mpz\_cmp\_ui (mpz\_t OP1, unsigned long int OP2)
- Macro: int mpz\_cmp\_si (mpz\_t OP1, signed long int OP2)  
Compare OP1 and OP2. Return a positive value if  $OP1 > OP2$ , zero if  $OP1 = OP2$ , and a negative value if  $OP1 < OP2$ .

These functions are actually implemented as macros. They evaluate their arguments multiple times.

- Macro: int mpz\_sgn (mpz\_t OP)  
Return +1 if  $OP > 0$ , 0 if  $OP = 0$ , and -1 if  $OP < 0$ .

This function is actually implemented as a macro. It evaluates its arguments multiple times.

## 1.14 gmp.guide/Integer Logic and Bit Fiddling

### Logical and Bit Manipulation Functions

=====

These functions behave as if two's complement arithmetic were used (although sign-magnitude is used by the actual implementation).

- Function: void mpz\_and (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
Set ROP to OP1 logical-and OP2.
- Function: void mpz\_ior (mpz\_t ROP, mpz\_t OP1, mpz\_t OP2)  
Set ROP to OP1 inclusive-or OP2.
- Function: void mpz\_com (mpz\_t ROP, mpz\_t OP)  
Set ROP to the one's complement of OP.
- Function: unsigned long int mpz\_popcount (mpz\_t OP)  
For non-negative numbers, return the population count of OP. For negative numbers, return the largest possible value (MAX\_ULONG).
- Function: unsigned long int mpz\_hamdist (mpz\_t OP1, mpz\_t OP2)  
If OP1 and OP2 are both non-negative, return the hamming distance between the two operands. Otherwise, return the largest possible value (MAX\_ULONG).

It is possible to extend this function to return a useful value when the operands are both negative, but the current implementation returns MAX\_ULONG in this case. \*Do not depend on this behavior, since it will change in future versions of the library.\*

- Function: unsigned long int mpz\_scan0 (mpz\_t OP, unsigned long int STARTING\_BIT)  
Scan OP, starting with bit STARTING\_BIT, towards more significant bits, until the first clear bit is found. Return the index of the found bit.
- Function: unsigned long int mpz\_scan1 (mpz\_t OP, unsigned long int STARTING\_BIT)  
Scan OP, starting with bit STARTING\_BIT, towards more significant bits, until the first set bit is found. Return the index of the found bit.
- Function: void mpz\_setbit (mpz\_t ROP, unsigned long int BIT\_INDEX)  
Set bit BIT\_INDEX in OP1.
- Function: void mpz\_clrbit (mpz\_t ROP, unsigned long int BIT\_INDEX)  
Clear bit BIT\_INDEX in OP1.

## 1.15 gmp.guide/I-O of Integers

### Input and Output Functions

=====

Functions that perform input from a stdio stream, and functions that output to a stdio 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 'gmp.h', since that will allow 'gmp.h' to define prototypes for these functions.

- Function: `size_t mpz_out_str (FILE *STREAM, int BASE, mpz_t OP)`  
Output OP on stdio stream STREAM, as a string of digits in base BASE. The base may vary from 2 to 36.

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

- Function: `size_t mpz_inp_str (mpz_t ROP, FILE *STREAM, int BASE)`  
Input a possibly white-space preceded string in base BASE from stdio stream STREAM, and put the read integer in ROP. The base may vary from 2 to 36. If BASE is 0, the actual base is determined from the leading characters: if the first two characters are '0x' or '0X', hexadecimal is assumed, otherwise if the first character is '0', octal is assumed, otherwise decimal is assumed.

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

- Function: `size_t mpz_out_raw (FILE *STREAM, mpz_t OP)`  
Output OP on stdio stream STREAM, in raw binary format. The integer is written in a portable format, with 4 bytes of size information, and that many bytes of limbs. Both the size and the limbs are written in decreasing significance order (i.e., in big-endian).

The output can be read with 'mpz\_inp\_raw'.

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

The output of this can not be read by 'mpz\_inp\_raw' from GMP 1, because of changes necessary for compatibility between 32-bit and 64-bit machines.

- Function: `size_t mpz_inp_raw (mpz_t ROP, FILE *STREAM)`  
Input from stdio stream STREAM in the format written by 'mpz\_out\_raw', and put the result in ROP. Return the number of bytes read, or if an error occurred, return 0.

This routine can read the output from 'mpz\_out\_raw' also from GMP 1, in spite of changes necessary for compatibility between 32-bit and 64-bit machines.

## 1.16 gmp.guide/Miscellaneous Integer Functions

### Miscellaneous Functions

=====

- Function: `void mpz_random (mpz_t ROP, mp_size_t MAX_SIZE)`  
Generate a random integer of at most MAX\_SIZE limbs. The generated random number doesn't satisfy any particular requirements of



randomness. Negative random numbers are generated when `MAX_SIZE` is negative.

- Function: `void mpz_random2 (mpz_t ROP, mp_size_t MAX_SIZE)`  
Generate a random integer of at most `MAX_SIZE` limbs, with long strings of zeros and ones in the binary representation. 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.
- Function: `size_t mpz_size (mpz_t OP)`  
Return the size of `OP` measured in number of limbs. If `OP` is zero, the returned value will be zero.

\*This function is obsolete. It will disappear from future MP releases.\*

- Function: `size_t mpz_sizeinbase (mpz_t OP, int BASE)`  
Return the size of `OP` measured in number of digits in base `BASE`. The base may vary from 2 to 36. The returned value will be exact or 1 too big. If `BASE` is a power of 2, the returned value will always be exact.

This function is useful in order to allocate the right amount of space before converting `OP` to a string. The right amount of allocation is normally two more than the value returned by `'mpz_sizeinbase'` (one extra for a minus sign and one for the terminating `'\0'`).

## 1.17 gmp.guide/Rational Number Functions

### Rational Number Functions

\*\*\*\*\*

This chapter describes the MP functions for performing arithmetic on rational numbers. These functions start with the prefix `'mpq_'`.

Rational numbers are stored in objects of type `'mpq_t'`.

All rational arithmetic functions assume operands have a canonical form, and canonicalize their result. The canonical form means that the denominator and the numerator have no common factors, and that the denominator is positive. Zero has the unique representation `0/1`.

Pure assignment functions do not canonicalize the assigned variable. It is the responsibility of the user to canonicalize the assigned variable before any arithmetic operations are performed on that variable. \*Note that this is an incompatible change from version 1 of the library.\*

- Function: `void mpq_canonicalize (mpq_t OP)`  
Remove any factors that are common to the numerator and denominator of `OP`, and make the denominator positive.

Initializing Rationals  
 Assigning Rationals  
 Simultaneous Integer Init & Assign  
 Comparing Rationals  
 Applying Integer Functions  
 Miscellaneous Rational Functions

## 1.18 gmp.guide/Initializing Rationals

Initialization and Assignment Functions

=====

- Function: void mpq\_init (mpq\_t DEST\_RATIONAL)  
 Initialize DEST\_RATIONAL and set it to 0/1. Each variable should normally only be initialized once, or at least cleared out (using the function 'mpq\_clear') between each initialization.
- Function: void mpq\_clear (mpq\_t RATIONAL\_NUMBER)  
 Free the space occupied by RATIONAL\_NUMBER. Make sure to call this function for all 'mpq\_t' variables when you are done with them.
- Function: void mpq\_set (mpq\_t ROP, mpq\_t OP)
- Function: void mpq\_set\_z (mpq\_t ROP, mpz\_t OP)  
 Assign ROP from OP.
- Function: void mpq\_set\_ui (mpq\_t ROP, unsigned long int OP1, unsigned long int OP2)
- Function: void mpq\_set\_si (mpq\_t ROP, signed long int OP1, unsigned long int OP2)  
 Set the value of ROP to OP1/OP2. Note that if OP1 and OP2 have common factors, ROP has to be passed to 'mpq\_canonicalize' before any operations are performed on ROP.

## 1.19 gmp.guide/Assigning Rationals

Arithmetic Functions

=====

- Function: void mpq\_add (mpq\_t SUM, mpq\_t ADDEND1, mpq\_t ADDEND2)  
 Set SUM to ADDEND1 + ADDEND2.
- Function: void mpq\_sub (mpq\_t DIFFERENCE, mpq\_t MINUEND, mpq\_t SUBTRAHEND)

Set DIFFERENCE to MINUEND - SUBTRAHEND.

- Function: void mpq\_mul (mpq\_t PRODUCT, mpq\_t MULTIPLIER, mpq\_t MULTIPLICAND)  
Set PRODUCT to MULTIPLIER times MULTIPLICAND.
- Function: void mpq\_div (mpq\_t QUOTIENT, mpq\_t DIVIDEND, mpq\_t DIVISOR)  
Set QUOTIENT to DIVIDEND/DIVISOR.
- Function: void mpq\_neg (mpq\_t NEGATED\_OPERAND, mpq\_t OPERAND)  
Set NEGATED\_OPERAND to -OPERAND.
- Function: void mpq\_inv (mpq\_t INVERTED\_NUMBER, mpq\_t NUMBER)  
Set INVERTED\_NUMBER to 1/NUMBER. If the new denominator is zero, this routine will divide by zero.

## 1.20 gmp.guide/Comparing Rationals

### Comparison Functions

=====

- Function: int mpq\_cmp (mpq\_t OP1, mpq\_t OP2)  
Compare OP1 and OP2. Return a positive value if  $OP1 > OP2$ , zero if  $OP1 = OP2$ , and a negative value if  $OP1 < OP2$ .  
  
To determine if two rationals are equal, 'mpq\_equal' is faster than 'mpq\_cmp'.
- Macro: int mpq\_cmp\_ui (mpq\_t OP1, unsigned long int NUM2, unsigned long int DEN2)  
Compare OP1 and NUM2/DEN2. Return a positive value if  $OP1 > NUM2/DEN2$ , zero if  $OP1 = NUM2/DEN2$ , and a negative value if  $OP1 < NUM2/DEN2$ .  
  
This routine allows that NUM2 and DEN2 have common factors.  
  
This function is actually implemented as a macro. It evaluates its arguments multiple times.
- Macro: int mpq\_sgn (mpq\_t OP)  
Return +1 if  $OP > 0$ , 0 if  $OP = 0$ , and -1 if  $OP < 0$ .  
  
This function is actually implemented as a macro. It evaluates its arguments multiple times.
- Function: int mpq\_equal (mpq\_t OP1, mpq\_t OP2)  
Return non-zero if OP1 and OP2 are equal, zero if they are non-equal. Although 'mpq\_cmp' can be used for the same purpose, this function is much faster.

## 1.21 gmp.guide/Applying Integer Functions

### Applying Integer Functions to Rationals

The set of 'mpq' functions is quite small. In particular, there are no functions for either input or output. But there are two macros that allow us to apply any 'mpz' function on the numerator or denominator of a rational number. If these macros are used to assign to the rational number, 'mpq\_canonicalize' normally need to be called afterwards.

- Macro: mpz\_t mpq\_numref (mpq\_t OP)
  - Macro: mpz\_t mpq\_denref (mpq\_t OP)
- Return a reference to the numerator and denominator of OP, respectively. The 'mpz' functions can be used on the result of these macros.

## 1.22 gmp.guide/Miscellaneous Rational Functions

### Miscellaneous Functions

- Function: double mpq\_get\_d (mpq\_t OP)
- Convert OP to a double.

These functions assign between either the numerator or denominator of a rational, and an integer. Instead of using these functions, it is preferable to use the more general mechanisms 'mpq\_numref' and 'mpq\_denref', together with 'mpz\_set'.

- Function: void mpq\_set\_num (mpq\_t RATIONAL, mpz\_t NUMERATOR)
- Copy NUMERATOR to the numerator of RATIONAL. When this risks to make the numerator and denominator of RATIONAL have common factors, you have to pass RATIONAL to 'mpq\_canonicalize' before any operations are performed on RATIONAL.

This function is equivalent to 'mpz\_set (mpq\_numref (RATIONAL), NUMERATOR)'.

- Function: void mpq\_set\_den (mpq\_t RATIONAL, mpz\_t DENOMINATOR)
- Copy DENOMINATOR to the denominator of RATIONAL. When this risks to make the numerator and denominator of RATIONAL have common factors, or if the denominator might be negative, you have to pass RATIONAL to 'mpq\_canonicalize' before any operations are performed on RATIONAL.

\*In version 1 of the library, negative denominators were handled by copying the sign to the numerator. That is no longer done.\*

This function is equivalent to 'mpz\_set (mpq\_denref (RATIONAL), DENOMINATORS)'.

- Function: void mpq\_get\_num (mpz\_t NUMERATOR, mpq\_t RATIONAL)

Copy the numerator of RATIONAL to the integer NUMERATOR, to prepare for integer operations on the numerator.

This function is equivalent to `'mpz_set (NUMERATOR, mpq_numref (RATIONAL))'`.

- Function: `void mpq_get_den (mpz_t DENOMINATOR, mpq_t RATIONAL)`  
Copy the denominator of RATIONAL to the integer DENOMINATOR, to prepare for integer operations on the denominator.

This function is equivalent to `'mpz_set (DENOMINATOR, mpq_denref (RATIONAL))'`.

## 1.23 gmp.guide/Floating-point Functions

### Floating-point Functions

\*\*\*\*\*

This is a description of the *\*preliminary\** interface for floating-point arithmetic in GNU MP 2.

The floating-point functions expect arguments of type `'mpf_t'`.

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

There is one significant characteristic of floating-point numbers that has motivated a difference between this function class and other MP 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 precision of variables used as input is ignored.

The precision of a calculation is defined as follows: Compute the requested operation exactly (with "infinite precision"), and truncate the result to the destination variable precision. 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 MP floating-point functions are *\*not\** intended as a smooth extension to the IEEE P754 arithmetic. Specifically, the results obtained on one computer often differs from the results obtained on a computer with a different word size.

Initializing Floats

Assigning Floats

Simultaneous Float Init & Assign

Converting Floats

Float Arithmetic

Float Comparison

I-O of Floats

Miscellaneous Float Functions

## 1.24 gmp.guide/Initializing Floats

Initialization and Assignment Functions

=====

- Function: void mpf\_set\_default\_prec (unsigned long int PREC)  
Set the default precision to be *\*at least\** PREC bits. All subsequent calls to 'mpf\_init' will use this precision, but previously initialized variables are unaffected.

An 'mpf\_t' object must be initialized before storing the first value in it. The functions 'mpf\_init' and 'mpf\_init2' are used for that purpose.

- Function: void mpf\_init (mpf\_t X)  
Initialize X to 0. Normally, a variable should be initialized once only or at least be cleared, using 'mpf\_clear', between initializations. The precision of X is undefined unless a default precision has already been established by a call to 'mpf\_set\_default\_prec'.
- Function: void mpf\_init2 (mpf\_t X, unsigned long int PREC)  
Initialize X to 0 and set its precision to be *\*at least\** PREC bits. Normally, a variable should be initialized once only or at least be cleared, using 'mpf\_clear', between initializations.
- Function: void mpf\_clear (mpf\_t X)  
Free the space occupied by X. Make sure to call this function for all 'mpf\_t' variables when you are done with them.

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

```
{
  mpf_t x, y;
  mpf_init (x);      /* use default precision */
  mpf_init2 (y, 256); /* precision *at least* 256 bits */
  ...
  /* Unless the program is about to exit, do ... */
  mpf_clear (x);
  mpf_clear (y);
}
```

The following three 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.

- Function: void mpf\_set\_prec (mpf\_t ROP, unsigned long int PREC)  
Set the precision of ROP to be *at least* PREC bits. Since changing the precision involves calls to 'realloc', this routine should not be called in a tight loop.
- Function: unsigned long int mpf\_get\_prec (mpf\_t OP)  
Return the precision actually used for assignments of OP.
- Function: void mpf\_set\_prec\_raw (mpf\_t ROP, unsigned long int PREC)  
Set the precision of ROP to be *at least* PREC bits. This is a low-level function that does not change the allocation. The PREC argument must not be larger than the precision previously returned by 'mpf\_get\_prec'. It is crucial that the precision of ROP is ultimately reset to exactly the value returned by 'mpf\_get\_prec'.

## 1.25 gmp.guide/Assigning Floats

### Assignment Functions

---

These functions assign new values to already initialized floats (see

### Initializing Floats

).

- Function: void mpf\_set (mpf\_t ROP, mpf\_t OP)
- Function: void mpf\_set\_ui (mpf\_t ROP, unsigned long int OP)
- Function: void mpf\_set\_si (mpf\_t ROP, signed long int OP)
- Function: void mpf\_set\_d (mpf\_t ROP, double OP)
- Function: void mpf\_set\_z (mpf\_t ROP, mpz\_t OP)
- Function: void mpf\_set\_q (mpf\_t ROP, mpq\_t OP)  
Set the value of ROP from OP.
- Function: int mpf\_set\_str (mpf\_t ROP, char \*STR, int BASE)  
Set the value of ROP from the string in STR. The string is of the form 'M@N' or, if the base is 10 or less, alternatively 'MeN'. 'M' is the mantissa and 'N' is the exponent. The mantissa is always in the specified base. The exponent is either in the specified base or, if BASE is negative, in decimal.

The argument BASE may be in the ranges 2 to 36, or -36 to -2. Negative values are used to specify that the exponent is in decimal.

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.

White space is allowed in the string, and is simply ignored.

---

This function returns 0 if the entire string up to the '\0' is a valid number in base BASE. Otherwise it returns -1.

## 1.26 gmp.guide/Simultaneous Float Init & Assign

Combined Initialization and Assignment Functions

-----

For convenience, MP provides a parallel series of initialize-and-set functions which initialize the output and then store the value there. These functions' names have the form 'mpf\_init\_set...'

Once the float has been initialized by any of the 'mpf\_init\_set...' functions, it can be used as the source or destination operand for the ordinary float functions. Don't use an initialize-and-set function on a variable already initialized!

- Function: void mpf\_init\_set (mpf\_t ROP, mpf\_t OP)
  - Function: void mpf\_init\_set\_ui (mpf\_t ROP, unsigned long int OP)
  - Function: void mpf\_init\_set\_si (mpf\_t ROP, signed long int OP)
  - Function: void mpf\_init\_set\_d (mpf\_t ROP, double OP)
- Initialize ROP and set its value from OP.

The precision of ROP will be taken from the active default precision, as set by 'mpf\_set\_default\_prec'.

- Function: int mpf\_init\_set\_str (mpf\_t ROP, char \*STR, int BASE)
- Initialize ROP and set its value from the string in STR. See 'mpf\_set\_str' above for details on the assignment operation.

Note that ROP is initialized even if an error occurs. (I.e., you have to call 'mpf\_clear' for it.)

The precision of ROP will be taken from the active default precision, as set by 'mpf\_set\_default\_prec'.

## 1.27 gmp.guide/Converting Floats

Conversion Functions

=====

- Function: double mpf\_get\_d (mpf\_t OP)
- Convert OP to a double.
- Function: char \* mpf\_get\_str (char \*STR, mp\_exp\_t \*EXPPTR, int BASE, size\_t N\_DIGITS, mpf\_t OP)
- Convert OP to a string of digits in base BASE. The base may vary from 2 to 36. Generate at most N\_DIGITS significant digits, or if
-



`N_DIGITS` is 0, the maximum number of digits accurately representable by `OP`.

If `STR` is `NULL`, space for the mantissa is allocated using the default allocation function, and a pointer to the string is returned.

If `STR` is not `NULL`, it should point to a block of storage enough large for the mantissa, i.e., `N_DIGITS + 2`. The two extra bytes are for a possible minus sign, and for the terminating null character.

The exponent is written through the pointer `EXPPTR`.

If `N_DIGITS` is 0, the maximum number of digits meaningfully achievable from the precision of `OP` will be generated. Note that the space requirements for `STR` in this case will be impossible for the user to predetermine. Therefore, you need to pass `NULL` 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`.

## 1.28 gmp.guide/Float Arithmetic

### Arithmetic Functions

=====

- Function: `void mpf_add (mpf_t ROP, mpf_t OP1, mpf_t OP2)`
- Function: `void mpf_add_ui (mpf_t ROP, mpf_t OP1, unsigned long int OP2)`  
Set `ROP` to `OP1 + OP2`.
- Function: `void mpf_sub (mpf_t ROP, mpf_t OP1, mpf_t OP2)`
- Function: `void mpf_ui_sub (mpf_t ROP, unsigned long int OP1, mpf_t OP2)`
- Function: `void mpf_sub_ui (mpf_t ROP, mpf_t OP1, unsigned long int OP2)`  
Set `ROP` to `OP1 - OP2`.
- Function: `void mpf_mul (mpf_t ROP, mpf_t OP1, mpf_t OP2)`
- Function: `void mpf_mul_ui (mpf_t ROP, mpf_t OP1, unsigned long int OP2)`  
Set `ROP` to `OP1 times OP2`.

Division is undefined if the divisor is zero, and passing a zero divisor to the divide functions will make these functions intentionally divide by zero. This gives the user the possibility to handle arithmetic exceptions in these functions in the same manner as other arithmetic exceptions.

- Function: `void mpf_div (mpf_t ROP, mpf_t OP1, mpf_t OP2)`

- Function: void mpf\_ui\_div (mpf\_t ROP, unsigned long int OP1, mpf\_t OP2)
- Function: void mpf\_div\_ui (mpf\_t ROP, mpf\_t OP1, unsigned long int OP2)  
Set ROP to OP1/OP2.
- Function: void mpf\_sqrt (mpf\_t ROP, mpf\_t OP)
- Function: void mpf\_sqrt\_ui (mpf\_t ROP, unsigned long int OP)  
Set ROP to the square root of OP.
- Function: void mpf\_neg (mpf\_t ROP, mpf\_t OP)  
Set ROP to -OP.
- Function: void mpf\_abs (mpf\_t ROP, mpf\_t OP)  
Set ROP to the absolute value of OP.
- Function: void mpf\_mul\_2exp (mpf\_t ROP, mpf\_t OP1, unsigned long int OP2)  
Set ROP to OP1 times 2 raised to OP2.
- Function: void mpf\_div\_2exp (mpf\_t ROP, mpf\_t OP1, unsigned long int OP2)  
Set ROP to OP1 divided by 2 raised to OP2.

## 1.29 gmp.guide/Float Comparison

### Comparison Functions

=====

- Function: int mpf\_cmp (mpf\_t OP1, mpf\_t OP2)
- Function: int mpf\_cmp\_ui (mpf\_t OP1, unsigned long int OP2)
- Function: int mpf\_cmp\_si (mpf\_t OP1, signed long int OP2)  
Compare OP1 and OP2. Return a positive value if OP1 > OP2, zero if OP1 = OP2, and a negative value if OP1 < OP2.
- Function: int mpf\_eq (mpf\_t OP1, mpf\_t OP2, unsigned long int op3)  
Return non-zero if the first OP3 bits of OP1 and OP2 are equal, zero otherwise. I.e., test of OP1 and OP2 are approximately equal.
- Function: void mpf\_reldiff (mpf\_t ROP, mpf\_t OP1, mpf\_t OP2)  
Compute the relative difference between OP1 and OP2 and store the result in ROP.
- Macro: int mpf\_sgn (mpf\_t OP)  
Return +1 if OP > 0, 0 if OP = 0, and -1 if OP < 0.

This function is actually implemented as a macro. It evaluates its arguments multiple times.

## 1.30 gmp.guide/I-O of Floats

## Input and Output Functions

=====

Functions that perform input from a stdio stream, and functions that output to a stdio 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 'gmp.h', since that will allow 'gmp.h' to define prototypes for these functions.

- Function: `size_t mpf_out_str (FILE *STREAM, int BASE, size_t N_DIGITS, mpf_t OP)`

Output OP on stdio stream STREAM, as a string of digits in base BASE. 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 leading '0.' 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.

- Function: `size_t mpf_inp_str (mpf_t ROP, FILE *STREAM, int BASE)`
- Input a string in base BASE from stdio stream STREAM, 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'. 'M' is the mantissa and 'N' is the exponent. The mantissa is always in the specified base. The exponent is either in the specified base or, if BASE is negative, in decimal.

The argument BASE may be in the ranges 2 to 36, or -36 to -2. Negative values are used to specify that the exponent is in decimal.

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.

## 1.31 gmp.guide/Miscellaneous Float Functions

### Miscellaneous Functions

=====

- Function: `void mpf_random2 (mpf_t ROP, mp_size_t MAX_SIZE, mp_exp_t MAX_EXP)`

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.

## 1.32 gmp.guide/Low-level Functions

### Low-level Functions

\*\*\*\*\*

This chapter describes low-level MP functions, used to implement the high-level MP functions, but also intended for time-critical user code.

These functions start with the prefix `'mpn_'`.

The `'mpn'` functions are designed to be as fast as possible, *not* to provide a coherent calling interface. The different functions have somewhat similar interfaces, but there are variations that make them hard to use. These functions do as little as possible apart from the real multiple precision computation, so that no time is spent on things that not all callers need.

A source operand is specified by a pointer to the least significant limb and a limb count. A destination operand is specified by just a pointer. It is the responsibility of the caller to ensure that the destination has enough space for storing the result.

With this way of specifying operands, it is possible to perform computations on subranges of an argument, and store the result into a subrange of a destination.

A common requirement for all functions is that each source area needs at least one limb. No size argument may be zero.

The `'mpn'` functions is the base for the implementation of the `'mpz_'`, `'mpf_'`, and `'mpq_'` functions.

This example adds the number beginning at `SRC1_PTR` and the number beginning at `SRC2_PTR` and writes the sum at `DEST_PTR`. All areas have `SIZE` limbs.

```
cy = mpn_add_n (dest_ptr, src1_ptr, src2_ptr, size)
```

In the notation used here, a source operand is identified by the pointer to the least significant limb, and the limb count in braces. For example, `{sl_ptr, sl_size}`.

- Function: `mp_limb_t mpn_add_n (mp_limb_t * DEST_PTR, const mp_limb_t * SRC1_PTR, const mp_limb_t * SRC2_PTR, mp_size_t SIZE)`  
Add `{SRC1_PTR, SIZE}` and `{SRC2_PTR, SIZE}`, and write the `SIZE` least significant limbs of the result to `DEST_PTR`. Return carry, either 0 or 1.

This is the lowest-level function for addition. It is the preferred function for addition, since it is written in assembly for most targets. For addition of a variable to itself (i.e., SRC1\_PTR equals SRC2\_PTR, use 'mpn\_lshift' with a count of 1 for optimal speed.

- Function: mp\_limb\_t mpn\_add\_1 (mp\_limb\_t \* DEST\_PTR, const mp\_limb\_t \* SRC1\_PTR, mp\_size\_t SIZE, mp\_limb\_t SRC2\_LIMB)  
Add {SRC1\_PTR, SIZE} and SRC2\_LIMB, and write the SIZE least significant limbs of the result to DEST\_PTR. Return carry, either 0 or 1.

- Function: mp\_limb\_t mpn\_add (mp\_limb\_t \* DEST\_PTR, const mp\_limb\_t \* SRC1\_PTR, mp\_size\_t SRC1\_SIZE, const mp\_limb\_t \* SRC2\_PTR, mp\_size\_t SRC2\_SIZE)  
Add {SRC1\_PTR, SRC1\_SIZE} and {SRC2\_PTR, SRC2\_SIZE}, and write the SRC1\_SIZE least significant limbs of the result to DEST\_PTR. Return carry, either 0 or 1.

This function requires that SRC1\_SIZE is greater than or equal to SRC2\_SIZE.

- Function: mp\_limb\_t mpn\_sub\_n (mp\_limb\_t \* DEST\_PTR, const mp\_limb\_t \* SRC1\_PTR, const mp\_limb\_t \* SRC2\_PTR, mp\_size\_t SIZE)  
Subtract {SRC2\_PTR, SRC2\_SIZE} from {SRC1\_PTR, SIZE}, and write the SIZE least significant limbs of the result to DEST\_PTR. Return borrow, either 0 or 1.

This is the lowest-level function for subtraction. It is the preferred function for subtraction, since it is written in assembly for most targets.

- Function: mp\_limb\_t mpn\_sub\_1 (mp\_limb\_t \* DEST\_PTR, const mp\_limb\_t \* SRC1\_PTR, mp\_size\_t SIZE, mp\_limb\_t SRC2\_LIMB)  
Subtract SRC2\_LIMB from {SRC1\_PTR, SIZE}, and write the SIZE least significant limbs of the result to DEST\_PTR. Return borrow, either 0 or 1.

- Function: mp\_limb\_t mpn\_sub (mp\_limb\_t \* DEST\_PTR, const mp\_limb\_t \* SRC1\_PTR, mp\_size\_t SRC1\_SIZE, const mp\_limb\_t \* SRC2\_PTR, mp\_size\_t SRC2\_SIZE)  
Subtract {SRC2\_PTR, SRC2\_SIZE} from {SRC1\_PTR, SRC1\_SIZE}, and write the SRC1\_SIZE least significant limbs of the result to DEST\_PTR. Return borrow, either 0 or 1.

This function requires that SRC1\_SIZE is greater than or equal to SRC2\_SIZE.

- Function: void mpn\_mul\_n (mp\_limb\_t \* DEST\_PTR, const mp\_limb\_t \* SRC1\_PTR, const mp\_limb\_t \* SRC2\_PTR, mp\_size\_t SIZE)  
Multiply {SRC1\_PTR, SIZE} and {SRC2\_PTR, SIZE}, and write the \*entire\* result to DEST\_PTR.

The destination has to have space for 2SIZE limbs, even if the significant result might be one limb smaller.

- Function: `mp_limb_t mpn_mul_1 (mp_limb_t * DEST_PTR, const mp_limb_t * SRC1_PTR, mp_size_t SIZE, mp_limb_t SRC2_LIMB)`  
Multiply `{SRC1_PTR, SIZE}` and `SRC2_LIMB`, and write the `SIZE` least significant limbs of the product to `DEST_PTR`. Return the most significant limb of the product.

This is a low-level function that is a building block for general multiplication as well as other operations in MP. It is written in assembly for most targets.

Don't call this function if `SRC2_LIMB` is a power of 2; use `'mpn_lshift'` with a count equal to the logarithm of `SRC2_LIMB` instead, for optimal speed.

- Function: `mp_limb_t mpn_addmul_1 (mp_limb_t * DEST_PTR, const mp_limb_t * SRC1_PTR, mp_size_t SIZE, mp_limb_t SRC2_LIMB)`  
Multiply `{SRC1_PTR, SIZE}` and `SRC2_LIMB`, and add the `SIZE` least significant limbs of the product to `{DEST_PTR, SIZE}` and write the result to `DEST_PTR`. Return the most significant limb of the product, plus carry-out from the addition.

This is a low-level function that is a building block for general multiplication as well as other operations in MP. It is written in assembly for most targets.

- Function: `mp_limb_t mpn_submul_1 (mp_limb_t * DEST_PTR, const mp_limb_t * SRC1_PTR, mp_size_t SIZE, mp_limb_t SRC2_LIMB)`  
Multiply `{SRC1_PTR, SIZE}` and `SRC2_LIMB`, and subtract the `SIZE` least significant limbs of the product from `{DEST_PTR, SIZE}` and write the result to `DEST_PTR`. Return the most significant limb of the product, minus borrow-out from the subtraction.

This is a low-level function that is a building block for general multiplication and division as well as other operations in MP. It is written in assembly for most targets.

- Function: `mp_limb_t mpn_mul (mp_limb_t * DEST_PTR, const mp_limb_t * SRC1_PTR, mp_size_t SRC1_SIZE, const mp_limb_t * SRC2_PTR, mp_size_t SRC2_SIZE)`  
Multiply `{SRC1_PTR, SRC1_SIZE}` and `{SRC2_PTR, SRC2_SIZE}`, and write the result to `DEST_PTR`. Return the most significant limb of the result.

The destination has to have space for `SRC1_SIZE + SRC1_SIZE` limbs, even if the result might be one limb smaller.

This function requires that `SRC1_SIZE` is greater than or equal to `SRC2_SIZE`. The destination must be distinct from either input operands.

- Function: `mp_size_t mpn_divrem (mp_limb_t * R1P, mp_size_t XSIZE, mp_limb_t * RS2P, mp_size_t RS2SIZE, const mp_limb_t * S3P, mp_size_t S3SIZE)`  
Divide `{RS2P, RS2SIZE}` by `{S3P, S3SIZE}`, and write the quotient at `R1P`, with the exception of the most significant limb, which is returned. The remainder replaces the dividend at `RS2P`.

In addition to an integer quotient, XSIZE fraction limbs are developed, and stored after the integral limbs. For most usages, XSIZE will be zero.

It is required that RS2SIZE is greater than or equal to S3SIZE. It is required that the most significant bit of the divisor is set.

If the quotient is not needed, pass RS2P + S3SIZE as R1P. Aside from that special case, no overlap between arguments is permitted.

Return the most significant limb of the quotient, either 0 or 1.

The area at R1P needs to be RS2SIZE - S3SIZE + XSIZE limbs large.

- Function: `mp_limb_t mpn_divrem_1 (mp_limb_t * R1P, mp_size_t XSIZE, mp_limb_t * S2P, mp_size_t S2SIZE, mp_limb_t S3LIMB)`  
Divide {S2P, S2SIZE} by S3LIMB, and write the quotient at R1P. Return the remainder.

In addition to an integer quotient, XSIZE fraction limbs are developed, and stored after the integral limbs. For most usages, XSIZE will be zero.

The areas at R1P and S2P have to be identical or completely separate, not partially overlapping.

- Function: `mp_size_t mpn_divmod (mp_limb_t * R1P, mp_limb_t * RS2P, mp_size_t RS2SIZE, const mp_limb_t * S3P, mp_size_t S3SIZE)`  
\*This interface is obsolete. It will disappear from future releases. Use 'mpn\_divrem' in its stead.\*
- Function: `mp_limb_t mpn_divmod_1 (mp_limb_t * R1P, mp_limb_t * S2P, mp_size_t S2SIZE, mp_limb_t S3LIMB)`  
\*This interface is obsolete. It will disappear from future releases. Use 'mpn\_divrem\_1' in its stead.\*
- Function: `mp_limb_t mpn_mod_1 (mp_limb_t * S1P, mp_size_t S1SIZE, mp_limb_t S2LIMB)`  
Divide {S1P, S1SIZE} by S2LIMB, and return the remainder.
- Function: `mp_limb_t mpn_preinv_mod_1 (mp_limb_t * S1P, mp_size_t S1SIZE, mp_limb_t S2LIMB, mp_limb_t S3LIMB)`  
\*This interface is obsolete. It will disappear from future releases. Use 'mpn\_mod\_1' in its stead.\*
- Function: `mp_limb_t mpn_bdivmod (mp_limb_t * DEST_PTR, mp_limb_t * S1P, mp_size_t S1SIZE, const mp_limb_t * S2P, mp_size_t S2SIZE, unsigned long int D)`  
The function puts the low  $[D/\text{BITS\_PER\_MP\_LIMB}]$  limbs of  $Q = \{S1P, S1SIZE\} / \{S2P, S2SIZE\} \bmod 2^D$  at DEST\_PTR, and returns the high  $D \bmod \text{BITS\_PER\_MP\_LIMB}$  bits of  $Q$ .

$\{S1P, S1SIZE\} - Q * \{S2P, S2SIZE\} \bmod 2^{(S1SIZE * \text{BITS\_PER\_MP\_LIMB})}$  is placed at S1P. Since the low  $[D/\text{BITS\_PER\_MP\_LIMB}]$  limbs of this difference are zero, it is possible to overwrite the low limbs at S1P with this difference, provided  $\text{DEST\_PTR} \leq S1P$ .

This function requires that  $S1SIZE * BITS\_PER\_MP\_LIMB \geq D$ , and that  $\{S2P, S2SIZE\}$  is odd.

\*This interface is preliminary. It might change incompatibly in future revisions.\*

- Function: `mp_limb_t mpn_lshift (mp_limb_t * DEST_PTR, const mp_limb_t * SRC_PTR, mp_size_t SRC_SIZE, unsigned long int COUNT)`  
Shift  $\{SRC\_PTR, SRC\_SIZE\}$  COUNT bits to the left, and write the SRC\_SIZE least significant limbs of the result to DEST\_PTR. COUNT might be in the range 1 to  $n - 1$ , on an n-bit machine. The bits shifted out to the left are returned.

Overlapping of the destination space and the source space is allowed in this function, provided  $DEST\_PTR \geq SRC\_PTR$ .

This function is written in assembly for most targets.

- Function: `mp_limb_t mpn_rshift (mp_limb_t * DEST_PTR, const mp_limb_t * SRC_PTR, mp_size_t SRC_SIZE, unsigned long int COUNT)`  
Shift  $\{SRC\_PTR, SRC\_SIZE\}$  COUNT bits to the right, and write the SRC\_SIZE most significant limbs of the result to DEST\_PTR. COUNT might be in the range 1 to  $n - 1$ , on an n-bit machine. The bits shifted out to the right are returned.

Overlapping of the destination space and the source space is allowed in this function, provided  $DEST\_PTR \leq SRC\_PTR$ .

This function is written in assembly for most targets.

- Function: `int mpn_cmp (const mp_limb_t * SRC1_PTR, const mp_limb_t * SRC2_PTR, mp_size_t SIZE)`  
Compare  $\{SRC1\_PTR, SIZE\}$  and  $\{SRC2\_PTR, SIZE\}$  and return a positive value if  $src1 > src2$ , 0 if they are equal, and a negative value if  $src1 < src2$ .

- Function: `mp_size_t mpn_gcd (mp_limb_t * DEST_PTR, mp_limb_t * SRC1_PTR, mp_size_t SRC1_SIZE, mp_limb_t * SRC2_PTR, mp_size_t SRC2_SIZE)`  
Puts at DEST\_PTR the greatest common divisor of  $\{SRC1\_PTR, SRC1\_SIZE\}$  and  $\{SRC2\_PTR, SRC2\_SIZE\}$ ; both source operands are destroyed by the operation. The size in limbs of the greatest common divisor is returned.

$\{SRC1\_PTR, SRC1\_SIZE\}$  must be odd, and  $\{SRC2\_PTR, SRC2\_SIZE\}$  must have at least as many bits as  $\{SRC1\_PTR, SRC1\_SIZE\}$ .

\*This interface is preliminary. It might change incompatibly in future revisions.\*

- Function: `mp_limb_t mpn_gcd_1 (const mp_limb_t * SRC1_PTR, mp_size_t SRC1_SIZE, mp_limb_t SRC2_LIMB)`  
Return the greatest common divisor of  $\{SRC1\_PTR, SRC1\_SIZE\}$  and SRC2\_LIMB, where SRC2\_LIMB (as well as SRC1\_SIZE) must be different from 0.



- Function: `mp_size_t mpn_gcdext (mp_limb_t * R1P, mp_limb_t * R2P, mp_limb_t * S1P, mp_size_t S1SIZE, mp_limb_t * S2P, mp_size_t S2SIZE)`  
Puts at R1P the greatest common divisor of {S1P, S1SIZE} and {S2P, S2SIZE}. The first cofactor is written at R2P. Both source operands are destroyed by the operation. The size in limbs of the greatest common divisor is returned.

\*This interface is preliminary. It might change incompatibly in future revisions.\*

- Function: `mp_size_t mpn_sqrtrem (mp_limb_t * R1P, mp_limb_t * R2P, const mp_limb_t * SP, mp_size_t SIZE)`  
Compute the square root of {SP, SIZE} and put the result at R1P. Write the remainder at R2P, unless R2P is NULL.

Return the size of the remainder, whether R2P was NULL or non-NULL. Iff the operand was a perfect square, the return value will be 0.

The areas at R1P and SP have to be distinct. The areas at R2P and SP have to be identical or completely separate, not partially overlapping.

The area at R1P needs to have space for `ceil(SIZE/2)` limbs. The area at R2P needs to be SIZE limbs large.

\*This interface is preliminary. It might change incompatibly in future revisions.\*

- Function: `mp_size_t mpn_get_str (unsigned char *STR, int BASE, mp_limb_t * S1P, mp_size_t S1SIZE)`  
Convert {S1P, S1SIZE} to a raw unsigned char array in base BASE. The string is not in ASCII; to convert it to printable format, add the ASCII codes for '0' or 'A', depending on the base and range. There may be leading zeros in the string.

The area at S1P is clobbered.

Return the number of characters in STR.

The area at STR has to have space for the largest possible number represented by a S1SIZE long limb array, plus one extra character.

- Function: `mp_size_t mpn_set_str (mp_limb_t * R1P, const char *STR, size_t strsize, int BASE)`  
Convert the raw unsigned char array at STR of length STRSIZE to a limb array {S1P, S1SIZE}. The base of STR is BASE.

Return the number of limbs stored in R1P.

- Function: `unsigned long int mpn_scan0 (const mp_limb_t * S1P, unsigned long int BIT)`  
Scan S1P from bit position BIT for the next clear bit.

It is required that there be a clear bit within the area at S1P at or beyond bit position BIT, so that the function has something to

---

return.

\*This interface is preliminary. It might change incompatibly in future revisions.\*

- Function: unsigned long int mpn\_scan1 (const mp\_limb\_t \* S1P, unsigned long int BIT)  
Scan S1P from bit position BIT for the next set bit.

It is required that there be a set bit within the area at S1P at or beyond bit position BIT, so that the function has something to return.

\*This interface is preliminary. It might change incompatibly in future revisions.\*

- Function: void mpn\_random2 (mp\_limb\_t \* R1P, mp\_size\_t R1SIZE)  
Generate a random number of length R1SIZE with long strings of zeros and ones in the binary representation, and store it at R1P.

The generated random numbers are intended for testing the correctness of the implementation of the 'mpn' routines.

- Function: unsigned long int mpn\_popcount (const mp\_limb\_t \* S1P, unsigned long int SIZE)  
Count the number of set bits in {S1P, SIZE}.
- Function: unsigned long int mpn\_hamdist (const mp\_limb\_t \* S1P, const mp\_limb\_t \* S2P, unsigned long int SIZE)  
Compute the hamming distance between {S1P, SIZE} and {S2P, SIZE}.
- Function: int mpn\_perfect\_square\_p (const mp\_limb\_t \* S1P, mp\_size\_t SIZE)  
Return non-zero iff {S1P, SIZE} is a perfect square.

### 1.33 gmp.guide/BSD Compatible Functions

Berkeley MP Compatible Functions

\*\*\*\*\*

These functions are intended to be fully compatible with the Berkeley MP library which is available on many BSD derived U\*x systems.

The original Berkeley MP library has a usage restriction: you cannot use the same variable as both source and destination in a single function call. The compatible functions in GNU MP do not share this restriction--inputs and outputs may overlap.

It is not recommended that new programs are written using these functions. Apart from the incomplete set of functions, the interface for initializing 'MINT' objects is more error prone, and the 'pow' function collides with 'pow' in 'libm.a'.

Include the header 'mp.h' to get the definition of the necessary

types and functions. If you are on a BSD derived system, make sure to include GNU `'mp.h'` if you are going to link the GNU `'libmp.a'` to your program. This means that you probably need to give the `-I<dir>` option to the compiler, where `<dir>` is the directory where you have GNU `'mp.h'`.

- Function: `MINT * itom (signed short int INITIAL_VALUE)`  
Allocate an integer consisting of a 'MINT' object and dynamic limb space. Initialize the integer to INITIAL\_VALUE. Return a pointer to the 'MINT' object.
- Function: `MINT * xtom (char *INITIAL_VALUE)`  
Allocate an integer consisting of a 'MINT' object and dynamic limb space. Initialize the integer from INITIAL\_VALUE, a hexadecimal, '\0'-terminate C string. Return a pointer to the 'MINT' object.
- Function: `void move (MINT *SRC, MINT *DEST)`  
Set DEST to SRC by copying. Both variables must be previously initialized.
- Function: `void madd (MINT *SRC_1, MINT *SRC_2, MINT *DESTINATION)`  
Add SRC\_1 and SRC\_2 and put the sum in DESTINATION.
- Function: `void msub (MINT *SRC_1, MINT *SRC_2, MINT *DESTINATION)`  
Subtract SRC\_2 from SRC\_1 and put the difference in DESTINATION.
- Function: `void mult (MINT *SRC_1, MINT *SRC_2, MINT *DESTINATION)`  
Multiply SRC\_1 and SRC\_2 and put the product in DESTINATION.
- Function: `void mdiv (MINT *DIVIDEND, MINT *DIVISOR, MINT *QUOTIENT, MINT *REMAINDER)`
- Function: `void sdiv (MINT *DIVIDEND, signed short int DIVISOR, MINT *QUOTIENT, signed short int *REMAINDER)`  
Set QUOTIENT to DIVIDEND/DIVISOR, and REMAINDER to DIVIDEND mod DIVISOR. The quotient is rounded towards zero; the remainder has the same sign as the dividend unless it is zero.

Some implementations of these functions work differently--or not at all--for negative arguments.

- Function: `void msqrt (MINT *OPERAND, MINT *ROOT, MINT *REMAINDER)`  
Set ROOT to the truncated integer part of the square root of OPERAND. Set REMAINDER to OPERAND-ROOT\*ROOT, (i.e., zero if OPERAND is a perfect square).

If ROOT and REMAINDER are the same variable, the results are undefined.

- Function: `void pow (MINT *BASE, MINT *EXP, MINT *MOD, MINT *DEST)`  
Set DEST to (BASE raised to EXP) modulo MOD.
- Function: `void rpow (MINT *BASE, signed short int EXP, MINT *DEST)`  
Set DEST to BASE raised to EXP.
- Function: `void gcd (MINT *OPERAND1, MINT *OPERAND2, MINT *RES)`  
Set RES to the greatest common divisor of OPERAND1 and OPERAND2.
- Function: `int mcmp (MINT *OPERAND1, MINT *OPERAND2)`

Compare OPERAND1 and OPERAND2. Return a positive value if OPERAND1 > OPERAND2, zero if OPERAND1 = OPERAND2, and a negative value if OPERAND1 < OPERAND2.

- Function: void min (MINT \*DEST)  
Input a decimal string from 'stdin', and put the read integer in DEST. SPC and TAB are allowed in the number string, and are ignored.
- Function: void mout (MINT \*SRC)  
Output SRC to 'stdout', as a decimal string. Also output a newline.
- Function: char \* mtox (MINT \*OPERAND)  
Convert OPERAND to a hexadecimal string, and return a pointer to the string. The returned string is allocated using the default memory allocation function, 'malloc' by default.
- Function: void mfree (MINT \*OPERAND)  
De-allocate, the space used by OPERAND. \*This function should only be passed a value returned by 'itom' or 'xtom'.\*

## 1.34 gmp.guide/Custom Allocation

Custom Allocation

\*\*\*\*\*

By default, the MP functions use 'malloc', 'realloc', and 'free' for memory allocation. If 'malloc' or 'realloc' fails, the MP library terminates execution after printing a fatal error message to standard error.

For some applications, you may wish to allocate memory in other ways, or you may not want to have a fatal error when there is no more memory available. To accomplish this, you can specify alternative memory allocation functions.

- Function: void mp\_set\_memory\_functions (  
void \*(\*ALLOC\_FUNC\_PTR) (size\_t),  
void \*(\*REALLOC\_FUNC\_PTR) (void \*, size\_t, size\_t),  
void (\*FREE\_FUNC\_PTR) (void \*, size\_t))  
Replace the current allocation functions from the arguments. If an argument is NULL, the corresponding default function is retained.  
  
\*Make sure to call this function in such a way that there are no active MP objects that were allocated using the previously active allocation function! Usually, that means that you have to call this function before any other MP function.\*

The functions you supply should fit the following declarations:

- Function: void \* allocate\_function (size\_t ALLOC\_SIZE)  
This function should return a pointer to newly allocated space

with at least `ALLOC_SIZE` storage units.

- Function: `void * reallocate_function (void *PTR, size_t OLD_SIZE, size_t NEW_SIZE)`

This function should return a pointer to newly allocated space of at least `NEW_SIZE` storage units, after copying at least the first `OLD_SIZE` storage units from `PTR`. It should also de-allocate the space at `PTR`.

You can assume that the space at `PTR` was formerly returned from `'allocate_function'` or `'reallocate_function'`, for a request for `OLD_SIZE` storage units.

- Function: `void deallocate_function (void *PTR, size_t SIZE)`  
De-allocate the space pointed to by `PTR`.

You can assume that the space at `PTR` was formerly returned from `'allocate_function'` or `'reallocate_function'`, for a request for `SIZE` storage units.

(A "storage unit" is the unit in which the `'sizeof'` operator returns the size of an object, normally an 8 bit byte.)

## 1.35 gmp.guide/Contributors

### Contributors

\*\*\*\*\*

I would like to thank Gunnar Sjoedin and Hans Riesel for their help with mathematical problems, Richard Stallman for his help with design issues and for revising the first version of this manual, Brian Beuning and Doug Lea for their testing of early versions of the library.

John Amanatides of York University in Canada contributed the function `'mpz_probab_prime_p'`.

Paul Zimmermann of Inria sparked the development of GMP 2, with his comparisons between bignum packages.

Ken Weber (Kent State University, Universidade Federal do Rio Grande do Sul) contributed `'mpz_gcd'`, `'mpz_divexact'`, `'mpn_gcd'`, and `'mpn_bdivmod'`, partially supported by CNPq (Brazil) grant 301314194-2.

Per Bothner of Cygnus Support helped to set up MP to use Cygnus' configure. He has also made valuable suggestions and tested numerous intermediary releases.

Joachim Hollman was involved in the design of the `'mpf'` interface, and in the `'mpz'` design revisions for version 2.

Bennet Yee contributed the functions `'mpz_jacobi'` and `'mpz_legendre'`.

Andreas Schwab contributed the files `'mpn/m68k/lshift.S'` and `'mpn/m68k/rshift.S'`.

---

The development of floating point functions of GNU MP 2, were supported in part by the ESPRIT-BRA (Basic Research Activities) 6846 project POSSO (POLynomial System SOLving).

GNU MP 2 was finished and released by TMG Datakonsult, Sodermannagatan 5, 116 23 STOCKHOLM, SWEDEN, in cooperation with the IDA Center for Computing Sciences, USA.

## 1.36 gmp.guide/References

### References

\*\*\*\*\*

- \* Donald E. Knuth, "The Art of Computer Programming", vol 2, "Seminumerical Algorithms", 2nd edition, Addison-Wesley, 1981.
- \* John D. Lipson, "Elements of Algebra and Algebraic Computing", The Benjamin Cummings Publishing Company Inc, 1981.
- \* Richard M. Stallman, "Using and Porting GCC", Free Software Foundation, 1995.
- \* Peter L. Montgomery, "Modular Multiplication Without Trial Division", in Mathematics of Computation, volume 44, number 170, April 1985.
- \* Torbjorn Granlund and Peter L. Montgomery, "Division by Invariant Integers using Multiplication", in Proceedings of the SIGPLAN PLDI'94 Conference, June 1994.
- \* Tudor Jebelean, "An algorithm for exact division", Journal of Symbolic Computation, v. 15, 1993, pp. 169-180.
- \* Kenneth Weber, "The accelerated integer GCD algorithm", ACM Transactions on Mathematical Software, v. 21 (March), 1995, pp. 111-122.

## 1.37 gmp.guide/Concept Index

### Concept Index

\*\*\*\*\*

gmp.h  
MP Basics

mp.h  
BSD Compatible Functions

---

---

- Arithmetic functions <1>
  - Integer Arithmetic
- Arithmetic functions
  - Float Arithmetic
- Bit manipulation functions
  - Integer Logic and Bit Fiddling
- BSD MP compatible functions
  - BSD Compatible Functions
- Comparison functions
  - Float Comparison
- Conditions for copying GNU MP
  - Copying
- Conversion functions <1>
  - Converting Integers
- Conversion functions
  - Converting Floats
- Copying conditions
  - Copying
- Float arithmetic functions
  - Float Arithmetic
- Float assignment functions
  - Assigning Floats
- Float comparisons functions
  - Float Comparison
- Float functions
  - Floating-point Functions
- Float input and output functions
  - I-O of Floats
- Floating-point functions
  - Floating-point Functions
- Floating-point number
  - MP Basics
- I/O functions <1>
  - I-O of Floats
- I/O functions
  - I-O of Integers
- Initialization and assignment functions <1>
  - Simultaneous Float Init & Assign

---

Initialization and assignment functions  
Simultaneous Integer Init & Assign

Input functions <1>  
I-O of Integers

Input functions  
I-O of Floats

Installation  
Installing MP

Integer  
MP Basics

Integer arithmetic functions  
Integer Arithmetic

Integer assignment functions  
Assigning Integers

Integer conversion functions  
Converting Integers

Integer functions  
Integer Functions

Integer input and output functions  
I-O of Integers

Limb  
MP Basics

Logical functions  
Integer Logic and Bit Fiddling

Low-level functions  
Low-level Functions

Miscellaneous float functions  
Miscellaneous Float Functions

Miscellaneous integer functions  
Miscellaneous Integer Functions

Output functions <1>  
I-O of Floats

Output functions  
I-O of Integers

Rational number  
MP Basics

Rational number functions  
Rational Number Functions

---



Reporting bugs  
  Reporting Bugs

User-defined precision  
  Floating-point Functions

## 1.38 gmp.guide/Function Index

Function and Type Index

\*\*\*\*\*

mp\_limb\_t  
  MP Basics

mpf\_t  
  MP Basics

mpq\_t  
  MP Basics

mpz\_t  
  MP Basics

\_\_GNU\_MP\_VERSION  
  MP Basics

\_\_GNU\_MP\_VERSION\_MINOR  
  MP Basics

\_mpz\_realloc  
  Initializing Integers

allocate\_function  
  Custom Allocation

deallocate\_function  
  Custom Allocation

gcd  
  BSD Compatible Functions

itom  
  BSD Compatible Functions

madd  
  BSD Compatible Functions

mcmp  
  BSD Compatible Functions

---

---

mdiv  
  BSD Compatible Functions

mfree  
  BSD Compatible Functions

min  
  BSD Compatible Functions

mout  
  BSD Compatible Functions

move  
  BSD Compatible Functions

mp\_set\_memory\_functions  
  Custom Allocation

mpf\_abs  
  Float Arithmetic

mpf\_add  
  Float Arithmetic

mpf\_add\_ui  
  Float Arithmetic

mpf\_clear  
  Initializing Floats

mpf\_cmp  
  Float Comparison

mpf\_cmp\_si  
  Float Comparison

mpf\_cmp\_ui  
  Float Comparison

mpf\_div  
  Float Arithmetic

mpf\_div\_2exp  
  Float Arithmetic

mpf\_div\_ui  
  Float Arithmetic

mpf\_eq  
  Float Comparison

mpf\_get\_d  
  Converting Floats

mpf\_get\_prec  
  Initializing Floats

---

---

mpf\_get\_str  
Converting Floats

mpf\_init  
Initializing Floats

mpf\_init2  
Initializing Floats

mpf\_init\_set  
Simultaneous Float Init & Assign

mpf\_init\_set\_d  
Simultaneous Float Init & Assign

mpf\_init\_set\_si  
Simultaneous Float Init & Assign

mpf\_init\_set\_str  
Simultaneous Float Init & Assign

mpf\_init\_set\_ui  
Simultaneous Float Init & Assign

mpf\_inp\_str  
I-O of Floats

mpf\_mul  
Float Arithmetic

mpf\_mul\_2exp  
Float Arithmetic

mpf\_mul\_ui  
Float Arithmetic

mpf\_neg  
Float Arithmetic

mpf\_out\_str  
I-O of Floats

mpf\_random2  
Miscellaneous Float Functions

mpf\_reldiff  
Float Comparison

mpf\_set  
Assigning Floats

mpf\_set\_d  
Assigning Floats

mpf\_set\_default\_prec  
Initializing Floats

---

---

mpf\_set\_prec  
    Initializing Floats

mpf\_set\_prec\_raw  
    Initializing Floats

mpf\_set\_q  
    Assigning Floats

mpf\_set\_si  
    Assigning Floats

mpf\_set\_str  
    Assigning Floats

mpf\_set\_ui  
    Assigning Floats

mpf\_set\_z  
    Assigning Floats

mpf\_sgn  
    Float Comparison

mpf\_sqrt  
    Float Arithmetic

mpf\_sqrt\_ui  
    Float Arithmetic

mpf\_sub  
    Float Arithmetic

mpf\_sub\_ui  
    Float Arithmetic

mpf\_ui\_div  
    Float Arithmetic

mpf\_ui\_sub  
    Float Arithmetic

mpn\_add  
    Low-level Functions

mpn\_add\_1  
    Low-level Functions

mpn\_add\_n  
    Low-level Functions

mpn\_addmul\_1  
    Low-level Functions

mpn\_bdivmod  
    Low-level Functions

---

---

mpn\_cmp  
Low-level Functions

mpn\_divmod  
Low-level Functions

mpn\_divmod\_1  
Low-level Functions

mpn\_divrem  
Low-level Functions

mpn\_divrem\_1  
Low-level Functions

mpn\_gcd  
Low-level Functions

mpn\_gcd\_1  
Low-level Functions

mpn\_gcdext  
Low-level Functions

mpn\_get\_str  
Low-level Functions

mpn\_hamdist  
Low-level Functions

mpn\_lshift  
Low-level Functions

mpn\_mod\_1  
Low-level Functions

mpn\_mul  
Low-level Functions

mpn\_mul\_1  
Low-level Functions

mpn\_mul\_n  
Low-level Functions

mpn\_perfect\_square\_p  
Low-level Functions

mpn\_popcount  
Low-level Functions

mpn\_preinv\_mod\_1  
Low-level Functions

mpn\_random2  
Low-level Functions

---

---

mpn\_rshift  
Low-level Functions

mpn\_scan0  
Low-level Functions

mpn\_scan1  
Low-level Functions

mpn\_set\_str  
Low-level Functions

mpn\_sqrtrem  
Low-level Functions

mpn\_sub  
Low-level Functions

mpn\_sub\_1  
Low-level Functions

mpn\_sub\_n  
Low-level Functions

mpn\_submul\_1  
Low-level Functions

mpq\_add  
Assigning Rationals

mpq\_canonicalize  
Rational Number Functions

mpq\_clear  
Initializing Rationals

mpq\_cmp  
Comparing Rationals

mpq\_cmp\_ui  
Comparing Rationals

mpq\_denref  
Applying Integer Functions

mpq\_div  
Assigning Rationals

mpq\_equal  
Comparing Rationals

mpq\_get\_d  
Miscellaneous Rational Functions

mpq\_get\_den  
Miscellaneous Rational Functions

---

---

mpq\_get\_num  
Miscellaneous Rational Functions

mpq\_init  
Initializing Rationals

mpq\_inv  
Assigning Rationals

mpq\_mul  
Assigning Rationals

mpq\_neg  
Assigning Rationals

mpq\_numref  
Applying Integer Functions

mpq\_set  
Initializing Rationals

mpq\_set\_den  
Miscellaneous Rational Functions

mpq\_set\_num  
Miscellaneous Rational Functions

mpq\_set\_si  
Initializing Rationals

mpq\_set\_ui  
Initializing Rationals

mpq\_set\_z  
Initializing Rationals

mpq\_sgn  
Comparing Rationals

mpq\_sub  
Assigning Rationals

mpz\_abs  
Integer Arithmetic

mpz\_add  
Integer Arithmetic

mpz\_add\_ui  
Integer Arithmetic

mpz\_and  
Integer Logic and Bit Fiddling

mpz\_array\_init  
Initializing Integers

---

---

mpz\_cdiv\_q  
Integer Arithmetic

mpz\_cdiv\_q\_ui  
Integer Arithmetic

mpz\_cdiv\_qr  
Integer Arithmetic

mpz\_cdiv\_qr\_ui  
Integer Arithmetic

mpz\_cdiv\_r  
Integer Arithmetic

mpz\_cdiv\_r\_ui  
Integer Arithmetic

mpz\_cdiv\_ui  
Integer Arithmetic

mpz\_clear  
Initializing Integers

mpz\_clrbit  
Integer Logic and Bit Fiddling

mpz\_cmp  
Comparison Functions

mpz\_cmp\_si  
Comparison Functions

mpz\_cmp\_ui  
Comparison Functions

mpz\_com  
Integer Logic and Bit Fiddling

mpz\_divexact  
Integer Arithmetic

mpz\_fac\_ui  
Integer Arithmetic

mpz\_fdiv\_q  
Integer Arithmetic

mpz\_fdiv\_q\_2exp  
Integer Arithmetic

mpz\_fdiv\_q\_ui  
Integer Arithmetic

mpz\_fdiv\_qr  
Integer Arithmetic

---



---

mpz\_fdiv\_qr\_ui  
Integer Arithmetic

mpz\_fdiv\_r  
Integer Arithmetic

mpz\_fdiv\_r\_2exp  
Integer Arithmetic

mpz\_fdiv\_r\_ui  
Integer Arithmetic

mpz\_fdiv\_ui  
Integer Arithmetic

mpz\_gcd  
Integer Arithmetic

mpz\_gcd\_ui  
Integer Arithmetic

mpz\_gcdext  
Integer Arithmetic

mpz\_get\_d  
Converting Integers

mpz\_get\_si  
Converting Integers

mpz\_get\_str  
Converting Integers

mpz\_get\_ui  
Converting Integers

mpz\_hamdist  
Integer Logic and Bit Fiddling

mpz\_init  
Initializing Integers

mpz\_init\_set  
Simultaneous Integer Init & Assign

mpz\_init\_set\_d  
Simultaneous Integer Init & Assign

mpz\_init\_set\_si  
Simultaneous Integer Init & Assign

mpz\_init\_set\_str  
Simultaneous Integer Init & Assign

mpz\_init\_set\_ui  
Simultaneous Integer Init & Assign

---

---

mpz\_inp\_raw  
I-O of Integers

mpz\_inp\_str  
I-O of Integers

mpz\_invert  
Integer Arithmetic

mpz\_ior  
Integer Logic and Bit Fiddling

mpz\_jacobi  
Integer Arithmetic

mpz\_legendre  
Integer Arithmetic

mpz\_mod  
Integer Arithmetic

mpz\_mod\_ui  
Integer Arithmetic

mpz\_mul  
Integer Arithmetic

mpz\_mul\_2exp  
Integer Arithmetic

mpz\_mul\_ui  
Integer Arithmetic

mpz\_neg  
Integer Arithmetic

mpz\_out\_raw  
I-O of Integers

mpz\_out\_str  
I-O of Integers

mpz\_perfect\_square\_p  
Integer Arithmetic

mpz\_popcount  
Integer Logic and Bit Fiddling

mpz\_pow\_ui  
Integer Arithmetic

mpz\_powm  
Integer Arithmetic

mpz\_powm\_ui  
Integer Arithmetic

---

---

mpz\_probab\_prime\_p  
Integer Arithmetic

mpz\_random  
Miscellaneous Integer Functions

mpz\_random2  
Miscellaneous Integer Functions

mpz\_scan0  
Integer Logic and Bit Fiddling

mpz\_scan1  
Integer Logic and Bit Fiddling

mpz\_set  
Assigning Integers

mpz\_set\_d  
Assigning Integers

mpz\_set\_f  
Assigning Integers

mpz\_set\_q  
Assigning Integers

mpz\_set\_si  
Assigning Integers

mpz\_set\_str  
Assigning Integers

mpz\_set\_ui  
Assigning Integers

mpz\_setbit  
Integer Logic and Bit Fiddling

mpz\_sgn  
Comparison Functions

mpz\_size  
Miscellaneous Integer Functions

mpz\_sizeinbase  
Miscellaneous Integer Functions

mpz\_sqrt  
Integer Arithmetic

mpz\_sqrtrem  
Integer Arithmetic

mpz\_sub  
Integer Arithmetic

---

---

mpz\_sub\_ui  
Integer Arithmetic

mpz\_tdiv\_q  
Integer Arithmetic

mpz\_tdiv\_q\_2exp  
Integer Arithmetic

mpz\_tdiv\_q\_ui  
Integer Arithmetic

mpz\_tdiv\_qr  
Integer Arithmetic

mpz\_tdiv\_qr\_ui  
Integer Arithmetic

mpz\_tdiv\_r  
Integer Arithmetic

mpz\_tdiv\_r\_2exp  
Integer Arithmetic

mpz\_tdiv\_r\_ui  
Integer Arithmetic

mpz\_ui\_pow\_ui  
Integer Arithmetic

msqrt  
BSD Compatible Functions

msub  
BSD Compatible Functions

mtox  
BSD Compatible Functions

mult  
BSD Compatible Functions

pow  
BSD Compatible Functions

realloc\_function  
Custom Allocation

rpow  
BSD Compatible Functions

sdiv  
BSD Compatible Functions

xtom  
BSD Compatible Functions

---

