GNU MP

The GNU Multiple Precision Arithmetic Library
Edition 2.0.2
June 1996

by Torbjörn Granlund, TMG Datakonsult

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

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 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.

1.1 How to use this Manual

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

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

2 Installing 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, your 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 Slowaris '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 Chapter 4 [Reporting Bugs], page 8, 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’.

2.1 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 Chapter 4 [Reporting Bugs], page 8.

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 ""
```

3 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.

3.1 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`.

3.2 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 is 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.

3.3 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.

3.4 Useful Macros and Constants

`const int` **mp_bits_per_limb** Global Constant

The number of bits per limb.

__GNU_MP_VERSION Macro
__GNU_MP_VERSION_MINOR Macro

The major and minor MP version, respectively, as integers.

3.5 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 −infinity). 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 implementated 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.)

3.6 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.

4 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.

5 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`.

5.1 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`.

`void` **mpz_init** (`mpz_t` *integer*) Function

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.

`void` **mpz_clear** (`mpz_t` *integer*) Function

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

`void *` **_mpz_realloc** (`mpz_t` *integer*, `mp_size_t` *new_alloc*) Function

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.

`void` **mpz_array_init** (`mpz_t` *integer_array*[], `size_t` *array_size*, `mp_size_t` *fixed_num_bits*) Function

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`!

5.1.1 Assignment Functions

These functions assign new values to already initialized integers (see Section 5.1 [Initializing Integers], page 9).

`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*) Function

Set the value of *rop* from *op*.

`int` **mpz_set_str** (`mpz_t` *rop*, `char` **str*, `int` *base*) Function

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.

5.1.2 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!

`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*) Function

Initialize *rop* with limb space and set the initial numeric value from *op*.

`int` **mpz_init_set_str** (`mpz_t` *rop*, `char *`*str*, `int` *base*) Function

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.)

5.2 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 Section 5.1.1 [Assigning Integers], page 10 and Section 5.6 [I/O of Integers], page 17.

`unsigned long int` **mpz_get_ui** (`mpz_t` *op*) Function

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.

`signed long int` **mpz_get_si** (`mpz_t` *op*) Function

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.

`double` **mpz_get_d** (`mpz_t` *op*) Function

Convert *op* to a double.

`char *` **mpz_get_str** (`char *`*str*, `int` *base*, `mpz_t` *op*) Function

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.

5.3 Arithmetic Functions

`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*) Function

Set *rop* to $op1 + op2$.

`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*) Function

Set *rop* to $op1 - op2$.

`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*) Function

Set *rop* to *op1* × *op2*.

`void` **mpz_mul_2exp** (`mpz_t` *rop*, `mpz_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to *op1* × 2^{op2}. This operation can also be defined as a left shift, *op2* steps.

`void` **mpz_neg** (`mpz_t` *rop*, `mpz_t` *op*) Function

Set *rop* to −*op*.

`void` **mpz_abs** (`mpz_t` *rop*, `mpz_t` *op*) Function

Set *rop* to the absolute value of *op*.

`void` **mpz_fac_ui** (`mpz_t` *rop*, `unsigned long int` *op*) Function

Set *rop* to *op*!, the factorial of *op*.

5.3.1 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.

`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*) Function

Set *rop* to [*op1*/*op2*]. The quotient is truncated towards 0.

`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*) Function

Set *rop* to (*op1* - [*op1*/*op2*] * *op2*). Unless the remainder is zero, it has the same sign as the dividend.

`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*) Function

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.

`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*) Function

Set *rop* to $\lfloor op1/op2 \rfloor$. (I.e., round the quotient towards $-\infty$.)

`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*) Function

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.

`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*) Function

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.

`unsigned long int` **mpz_fdiv_ui** (`mpz_t` *op1*, `unsigned long int` *op2*) Function

This function is similar to `mpz_fdiv_r_ui`, but the remainder is only returned; it is not stored anywhere.

`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*) Function

Set *rop* to $\lceil op1/op2 \rceil$. (I.e., round the quotient towards $+\infty$.)

`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*) Function

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.

`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*) Function

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.

`unsigned long int` **mpz_cdiv_ui** (`mpz_t` *op1*, `unsigned long int` *op2*) Function

Return the negated remainder, similar to `mpz_cdiv_r_ui`. (The difference is that this function doesn't store the remainder anywhere.)

`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*) Function

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.

`void` **mpz_divexact** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

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], page 39 Jebelean), it is the best choice for instances in which exact division is known to occur, such as reducing a rational to lowest terms.

`void` **mpz_tdiv_q_2exp** (`mpz_t` *rop*, `mpz_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to $op1/2^{op2}$. The quotient is rounded towards 0.

`void` **mpz_tdiv_r_2exp** (`mpz_t` *rop*, `mpz_t` *op1*, `unsigned long int` *op2*) Function

Divide *op1* by 2^{op2} and put the remainder in *rop*. Unless it is zero, *rop* will have the same sign as *op1*.

`void` **mpz_fdiv_q_2exp** (`mpz_t` *rop*, `mpz_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to $\lfloor op1/2^{op2} \rfloor$. The quotient is rounded towards $-\infty$.

`void` **mpz_fdiv_r_2exp** (`mpz_t` *rop*, `mpz_t` *op1*, `unsigned long int` *op2*) Function

Divide *op1* by 2^{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.

5.3.2 Exponentialization Functions

`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*) Function

Set *rop* to (*base* raised to *exp*) `mod` *mod*. If *exp* is negative, the result is undefined.

`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*) Function

Set *rop* to *base* raised to *exp*. The case of 0^0 yields 1.

5.3.3 Square Root Functions

`void` **mpz_sqrt** (`mpz_t` *rop*, `mpz_t` *op*) Function

Set *rop* to $\lfloor\sqrt{op}\rfloor$, the truncated integer part of the square root of *op*.

`void` **mpz_sqrtrem** (`mpz_t` *rop1*, `mpz_t` *rop2*, `mpz_t` *op*) Function

Set *rop1* to $\lfloor\sqrt{op}\rfloor$, like `mpz_sqrt`. Set *rop2* to $(op - rop1^2)$, (i.e., zero if *op* is a perfect square).

If *rop1* and *rop2* are the same variable, the results are undefined.

`int` **mpz_perfect_square_p** (`mpz_t` *op*) Function

Return non-zero if *op* is a perfect square, i.e., if the square root of *op* is an integer. Return zero otherwise.

5.3.4 Number Theoretic Functions

`int` **mpz_probab_prime_p** (`mpz_t` *op*, `int` *reps*) Function

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], page 39 Knuth).

`void` **mpz_gcd** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to the greatest common divisor of *op1* and *op2*.

`unsigned long int` **mpz_gcd_ui** (`mpz_t` *rop*, `mpz_t` *op1*, `unsigned long int` *op2*) Function

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.

`void` **mpz_gcdext** (`mpz_t` *g*, `mpz_t` *s*, `mpz_t` *t*, `mpz_t` *a*, `mpz_t` *b*) Function

Compute *g*, *s*, and *t*, such that $as + bt = g =$ `gcd` (a, b). If *t* is NULL, that argument is not computed.

`int` **mpz_invert** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

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*.

`int` **mpz_jacobi** (`mpz_t` *op1*, `mpz_t` *op2*) Function

`int` **mpz_legendre** (`mpz_t` *op1*, `mpz_t` *op2*) Function

Compute the Jacobi and Legendre symbols, respectively.

5.4 Comparison Functions

`int` **mpz_cmp** (`mpz_t` *op1*, `mpz_t` *op2*) Function

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

`int` **mpz_cmp_ui** (`mpz_t` *op1*, `unsigned long int` *op2*) Macro

`int` **mpz_cmp_si** (`mpz_t` *op1*, `signed long int` *op2*) Macro

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.

`int` **mpz_sgn** (`mpz_t` *op*) Macro

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.

5.5 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).

`void` **mpz_and** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to *op1* logical-and *op2*.

`void` **mpz_ior** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to *op1* inclusive-or *op2*.

`void` **mpz_com** (`mpz_t` *rop*, `mpz_t` *op*) Function

Set *rop* to the one's complement of *op*.

`unsigned long int` **mpz_popcount** (`mpz_t` *op*) Function

For non-negative numbers, return the population count of *op*. For negative numbers, return the largest possible value (*MAX_ULONG*).

`unsigned long int` **mpz_hamdist** (`mpz_t` *op1*, `mpz_t` *op2*) Function

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.**

`unsigned long int` **mpz_scan0** (`mpz_t` *op*, `unsigned long int` *starting_bit*) Function

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.

`unsigned long int` **mpz_scan1** (`mpz_t` *op*, `unsigned long int` *starting_bit*) Function

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.

`void` **mpz_setbit** (`mpz_t` *rop*, `unsigned long int` *bit_index*) Function

Set bit *bit_index* in *op1*.

`void` **mpz_clrbit** (`mpz_t` *rop*, `unsigned long int` *bit_index*) Function

Clear bit *bit_index* in *op1*.

5.6 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.

`size_t` **mpz_out_str** (`FILE` **stream*, `int` *base*, `mpz_t` *op*) Function

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.

`size_t` **mpz_inp_str** (`mpz_t` *rop*, `FILE` **stream*, `int` *base*) Function

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.

`size_t` **mpz_out_raw** (`FILE` **stream*, `mpz_t` *op*) Function

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.

`size_t` **mpz_inp_raw** (`mpz_t` *rop*, `FILE` **stream*) Function

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.

5.7 Miscellaneous Functions

`void` **mpz_random** (`mpz_t` *rop*, `mp_size_t` *max_size*) Function

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.

`void` **mpz_random2** (`mpz_t` *rop*, `mp_size_t` *max_size*) Function

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.

`size_t` **mpz_size** (`mpz_t` *op*) Function

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.

`size_t` **mpz_sizeinbase** (`mpz_t` *op*, `int` *base*) Function

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').

6 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 from 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.**

`void` **mpq_canonicalize** (`mpq_t` *op*) Function

Remove any factors that are common to the numerator and denominator of *op*, and make the denominator positive.

6.1 Initialization and Assignment Functions

`void` **mpq_init** (`mpq_t` *dest_rational*) Function

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.

`void` **mpq_clear** (`mpq_t` *rational_number*) Function

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

`void` **mpq_set** (`mpq_t` *rop*, `mpq_t` *op*) Function
`void` **mpq_set_z** (`mpq_t` *rop*, `mpz_t` *op*) Function

Assign *rop* from *op*.

`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*) Function

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*.

6.2 Arithmetic Functions

`void` **mpq_add** (`mpq_t` *sum*, `mpq_t` *addend1*, `mpq_t` *addend2*) Function

Set *sum* to *addend1* + *addend2*.

`void` **mpq_sub** (`mpq_t` *difference*, `mpq_t` *minuend*, `mpq_t` *subtrahend*) Function

Set *difference* to *minuend* − *subtrahend*.

`void` **mpq_mul** (`mpq_t` *product*, `mpq_t` *multiplier*, `mpq_t` *multiplicand*) Function

Set *product* to *multiplier* × *multiplicand*.

`void` **mpq_div** (`mpq_t` *quotient*, `mpq_t` *dividend*, `mpq_t` *divisor*) Function

Set *quotient* to *dividend*/*divisor*.

`void` **mpq_neg** (`mpq_t` *negated_operand*, `mpq_t` *operand*) Function

Set *negated_operand* to −*operand*.

`void` **mpq_inv** (`mpq_t` *inverted_number*, `mpq_t` *number*) Function

Set *inverted_number* to 1/*number*. If the new denominator is zero, this routine will divide by zero.

6.3 Comparison Functions

`int` **mpq_cmp** (`mpq_t` *op1*, `mpq_t` *op2*) Function

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`.

`int` **mpq_cmp_ui** (`mpq_t` *op1*, `unsigned long int` *num2*, `unsigned long int` *den2*) Macro

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.

`int` **mpq_sgn** (`mpq_t` *op*) Macro

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.

`int` **mpq_equal** (`mpq_t` *op1*, `mpq_t` *op2*) Function

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.

6.4 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.

`mpz_t` **mpq_numref** (`mpq_t` *op*) Macro

`mpz_t` **mpq_denref** (`mpq_t` *op*) Macro

Return a reference to the numerator and denominator of *op*, respectively. The `mpz` functions can be used on the result of these macros.

6.5 Miscellaneous Functions

`double` **mpq_get_d** (`mpq_t` *op*) Function

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`.

`void` **mpq_set_num** (`mpq_t` *rational*, `mpz_t` *numerator*) Function

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*`)`.

`void` **mpq_set_den** (`mpq_t` *rational*, `mpz_t` *denominator*) Function

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*`)`.

`void` **mpq_get_num** (`mpz_t` *numerator*, `mpq_t` *rational*) Function

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*`))`.

`void` **mpq_get_den** (`mpz_t` *denominator*, `mpq_t` *rational*) Function

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*`))`.

7 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.

7.1 Initialization and Assignment Functions

`void` **mpf_set_default_prec** (`unsigned long int` *prec*) Function

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.

`void` **mpf_init** (`mpf_t` *x*) Function

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`.

`void` **mpf_init2** (`mpf_t` *x*, `unsigned long int` *prec*) Function

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.

`void` **mpf_clear** (`mpf_t` *x*) Function

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.

`void` **mpf_set_prec** (`mpf_t` *rop*, `unsigned long int` *prec*) Function

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.

`unsigned long int` **mpf_get_prec** (`mpf_t` *op*) Function

Return the precision actually used for assignments of *op*.

`void` **mpf_set_prec_raw** (`mpf_t` *rop*, `unsigned long int` *prec*) Function

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 that 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`.

7.1.1 Assignment Functions

These functions assign new values to already initialized floats (see Section 7.1 [Initializing Floats], page 22).

`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*) Function

Set the value of *rop* from *op*.

`int` **mpf_set_str** (`mpf_t` *rop*, `char` **str*, `int` *base*) Function

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.

7.1.2 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!

`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*) Function

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`.

`int` **mpf_init_set_str** (`mpf_t` *rop*, `char` **str*, `int` *base*) Function

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`.

7.2 Conversion Functions

`double` **mpf_get_d** (`mpf_t` *op*) Function

Convert *op* to a double.

`char *` **mpf_get_str** (`char` **str*, `mp_exp_t` **expptr*, `int` *base*, `size_t` *n_digits*, `mpf_t` *op*) Function

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*.

7.3 Arithmetic Functions

`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*) Function

Set *rop* to $op1 + op2$.

`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*) Function

Set *rop* to $op1 - op2$.

`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*) Function

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.

`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*) Function

Set *rop* to $op1/op2$.

`void` **mpf_sqrt** (`mpf_t` *rop*, `mpf_t` *op*) Function
`void` **mpf_sqrt_ui** (`mpf_t` *rop*, `unsigned long int` *op*) Function

Set *rop* to $\sqrt{op}$.

`void` **mpf_neg** (`mpf_t` *rop*, `mpf_t` *op*) Function

Set *rop* to $-op$.

`void` **mpf_abs** (`mpf_t` *rop*, `mpf_t` *op*) Function

Set *rop* to the absolute value of *op*.

`void` **mpf_mul_2exp** (`mpf_t` *rop*, `mpf_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to *op1* $\times 2^{op2}$.

`void` **mpf_div_2exp** (`mpf_t` *rop*, `mpf_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to *op1*$/2^{op2}$.

7.4 Comparison Functions

`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*) Function

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

`int` **mpf_eq** (`mpf_t` *op1*, `mpf_t` *op2*, `unsigned long int op3`) Function

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

`void` **mpf_reldiff** (`mpf_t` *rop*, `mpf_t` *op1*, `mpf_t` *op2*) Function

Compute the relative difference between *op1* and *op2* and store the result in *rop*.

`int` **mpf_sgn** (`mpf_t` *op*) Macro

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.

7.5 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.

`size_t` **mpf_out_str** (`FILE *`*stream*, `int` *base*, `size_t` *n_digits*, `mpf_t` *op*) Function

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.

`size_t` **mpf_inp_str** (`mpf_t` *rop*, `FILE` **stream*, `int` *base*) Function

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.

7.6 Miscellaneous Functions

`void` **mpf_random2** (`mpf_t` *rop*, `mp_size_t` *max_size*, `mp_exp_t` *max_exp*) Function

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

8 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, {s1_ptr, s1_size}.

`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*) Function

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.

`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*) Function

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.

`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*) Function

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*.

`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*) Function

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.

`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*) Function

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.

`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*) Function

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*.

`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*) Function

Multiply {*src1_ptr*, *size*} and {*src2_ptr*, *size*}, and write the **entire** result to *dest_ptr*.

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

`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*) Function

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.

`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*) Function

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* *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.

`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*) Function

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.

`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*) Function

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.

`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*) Function

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.

`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*) Function

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.

`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*) Function

This interface is obsolete. It will disappear from future releases. Use `mpn_divrem` in its stead.

`mp_limb_t` **mpn_divmod_1** (`mp_limb_t *` *r1p*, `mp_limb_t *` *s2p*, `mp_size_t` *s2size*, `mp_limb_t` *s3limb*) Function

This interface is obsolete. It will disappear from future releases. Use `mpn_divrem_1` in its stead.

`mp_limb_t` **mpn_mod_1** (`mp_limb_t *` *s1p*, `mp_size_t` *s1size*, `mp_limb_t` *s2limb*) Function

Divide {*s1p*, *s1size*} by *s2limb*, and return the remainder.

`mp_limb_t` **mpn_preinv_mod_1** (`mp_limb_t *` *s1p*, `mp_size_t` *s1size*, `mp_limb_t` *s2limb*, `mp_limb_t` *s3limb*) Function

This interface is obsolete. It will disappear from future releases. Use `mpn_mod_1` in its stead.

`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*) Function

The function puts the low [*d*/*BITS_PER_MP_LIMB*] limbs of *q* = {*s1p*, *s1size*}/{*s2p*, *s2size*} mod 2^*d* at *dest_ptr*, and returns the high *d* mod *BITS_PER_MP_LIMB* bits of *q*.

{*s1p*, *s1size*} - *q* * {*s2p*, *s2size*} mod 2^(*s1size***BITS_PER_MP_LIMB*) is placed at *s1p*. Since the low [*d*/*BITS_PER_MP_LIMB*] limbs of this difference are zero, it is possible to overwrite the low limbs at *s1p* with this difference, provided *dest_ptr* <= *s1p*.

This function requires that *s1size* * *BITS_PER_MP_LIMB* >= *D*, and that {*s2p*, *s2size*} is odd.

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

`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*) Function

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* >= *src_ptr*.

This function is written in assembly for most targets.

`mp_limp_t` **mpn_rshift** (`mp_limb_t *` *dest_ptr*, `const mp_limb_t *` *src_ptr*, `mp_size_t` *src_size*, `unsigned long int` *count*) Function

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* <= *src_ptr*.

This function is written in assembly for most targets.

`int` **mpn_cmp** (`const mp_limb_t *` *src1_ptr*, `const mp_limb_t *` *src2_ptr*, `mp_size_t` *size*) Function

Compare {*src1_ptr*, *size*} and {*src2_ptr*, *size*} and return a positive value if src1 > src2, 0 of they are equal, and a negative value if src1 < src2.

`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*) Function

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.

`mp_limb_t` **mpn_gcd_1** (`const mp_limb_t *` *src1_ptr*, `mp_size_t` *src1_size*, `mp_limb_t` *src2_limb*) Function

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.

`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*) Function

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.

`mp_size_t` **mpn_sqrtrem** (`mp_limb_t *` *r1p*, `mp_limb_t *` *r2p*, `const mp_limb_t *` *sp*, `mp_size_t` *size*) Function

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 $\lceil size/2 \rceil$ limbs. The area at *r2p* needs to be *size* limbs large.

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

`mp_size_t` **mpn_get_str** (`unsigned char` **str*, `int` *base*, `mp_limb_t *` *s1p*, `mp_size_t` *s1size*) Function

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.

`mp_size_t` **mpn_set_str** (`mp_limb_t *` *r1p*, `const char` **str*, `size_t` `strsize`, `int` *base*) Function

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*.

`unsigned long int` **mpn_scan0** (`const mp_limb_t *` *s1p*, `unsigned long int` *bit*) Function

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.

`unsigned long int` **mpn_scan1** (`const mp_limb_t *` *s1p*, `unsigned long int` *bit*) Function

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.

`void` **mpn_random2** (`mp_limb_t *` *r1p*, `mp_size_t` *r1size*) Function

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.

`unsigned long int` **mpn_popcount** (`const mp_limb_t *` *s1p*, `unsigned long int` *size*) Function

Count the number of set bits in {*s1p*, *size*}.

`unsigned long int` **mpn_hamdist** (`const mp_limb_t *` *s1p*, `const mp_limb_t *` *s2p*, `unsigned long int` *size*) Function

Compute the hamming distance between {*s1p*, *size*} and {*s2p*, *size*}.

`int` **mpn_perfect_square_p** (`const mp_limb_t *` *s1p*, `mp_size_t` *size*) Function

Return non-zero iff {*s1p*, *size*} is a perfect square.

9 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*ix 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 you 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`'.

`MINT *` **itom** (`signed short int` *initial_value*) Function

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.

`MINT *` **xtom** (`char *`*initial_value*) Function

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.

`void` **move** (`MINT *`*src*, `MINT *`*dest*) Function

Set *dest* to *src* by copying. Both variables must be previously initialized.

`void` **madd** (`MINT *`*src_1*, `MINT *`*src_2*, `MINT *`*destination*) Function

Add *src_1* and *src_2* and put the sum in *destination*.

`void` **msub** (`MINT *`*src_1*, `MINT *`*src_2*, `MINT *`*destination*) Function

Subtract *src_2* from *src_1* and put the difference in *destination*.

`void` **mult** (`MINT *`*src_1*, `MINT *`*src_2*, `MINT *`*destination*) Function

Multiply *src_1* and *src_2* and put the product in *destination*.

`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*) Function

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.

`void` **msqrt** (`MINT` **operand*, `MINT` **root*, `MINT` **remainder*) Function

Set *root* to $\lfloor\sqrt{operand}\rfloor$, like `mpz_sqrt`. Set *remainder* to $(operand - root^2)$, (i.e., zero if *operand* is a perfect square).

If *root* and *remainder* are the same variable, the results are undefined.

`void` **pow** (`MINT` **base*, `MINT` **exp*, `MINT` **mod*, `MINT` **dest*) Function

Set *dest* to (*base* raised to *exp*) modulo *mod*.

`void` **rpow** (`MINT` **base*, `signed short int` *exp*, `MINT` **dest*) Function

Set *dest* to *base* raised to *exp*.

`void` **gcd** (`MINT` **operand1*, `MINT` **operand2*, `MINT` **res*) Function

Set *res* to the greatest common divisor of *operand1* and *operand2*.

`int` **mcmp** (`MINT` **operand1*, `MINT` **operand2*) Function

Compare *operand1* and *operand2*. Return a positive value if *operand1* `>` *operand2*, zero if *operand1* = *operand2*, and a negative value if *operand1* `<` *operand2*.

`void` **min** (`MINT` **dest*) Function

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.

`void` **mout** (`MINT` **src*) Function

Output *src* to `stdout`, as a decimal string. Also output a newline.

`char *` **mtox** (`MINT` **operand*) Function

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.

`void` **mfree** (`MINT` **operand*) Function

De-allocate, the space used by *operand*. **This function should only be passed a value returned by `itom` or `xtom`.**

10 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.

`void` **mp_set_memory_functions** (Function
`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:

`void *` **allocate_function** (`size_t` *alloc_size*) Function

This function should return a pointer to newly allocated space with at least *alloc_size* storage units.

`void *` **reallocate_function** (`void *`*ptr*`, size_t` *old_size*`, size_t` *new_size*) Function

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.

`void` **deallocate_function** (`void *`*ptr*`, size_t` *size*) Function

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.)

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.

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.

Concept Index

Function and Type Index

Table of Contents