gmp.info

COLLABORATORS			
	TITLE : gmp.info		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		September 19, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

gmp.info

1.1 gmp.info

```
                 Function Index
```

1.2 gmp.info/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 GMP 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 GMP 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 GMP library are found
in the General Public License that accompany the source code.
```

1.3 gmp.info/Intro

```
                Introduction to MP
******************

   GNU MP is a portable library for arbitrary precision integer and
rational number arithmetic.(1)  It aims to provide the fastest possible
arithmetic for all applications that need more than two words of
integer precision.

   Most often, applications tend to use just a few words of precision;
but some applications may need thousands of words.  GNU MP is designed
to give good performance for both kinds of applications, by choosing
algorithms based on the sizes of the operands.
```

There are five groups of functions in the MP library:

1. Functions for signed integer arithmetic, with names beginning with mpz_.

2. Functions for rational number arithmetic, with names beginning with mpq_.

3. Functions compatible with Berkeley MP, such as itom, madd, and mult.

4. 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_.

5. Miscellaneous functions.

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.) Multi-precision numbers, whether output or input, are always passed as addresses to the declared type.

Nomenclature

Thanks

---------- Footnotes ----------

(1) The limit of the precision is set by the available memory in your computer.

1.4 gmp.info/Nomenclature

Nomenclature and Data Types
===========================

In this manual, integer means a multiple precision integer, as used in the MP package. The C data type for such integers is MP_INT. For example:

```
MP_INT sum;

struct foo { MP_INT x, y; };

MP_INT vec[20];
```

Rational number means a multiple precision fraction. The C data type for these fractions is MP_RAT. For example:

```
     MP_RAT quotient;
```

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 bits.

1.5 gmp.info/Thanks

Thanks
======

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 this manual, Brian Beuning and Doug Lea for their testing of various versions of the library, and Joachim Hollman for his many valuable suggestions.

Special thanks to Brian Beuning, he has shaked out many bugs from early versions of the code!

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

1.6 gmp.info/Initialization

Initialization
****

Before you can use a variable or object of type MP_INT or MP_RAT, you must initialize it. This fills in the components that point to dynamically allocated space for the limbs of the number.

When you are finished using the object, you should clear out the object. This frees the dynamic space that it points to, so the space can be used again.

Once you have initialized the object, you need not be concerned about allocating additional space. The functions in the MP package automatically allocate additional space when the object does not already have enough space. They do not, however, reduce the space in use when a smaller number is stored in the object. Most of the time, this policy is best, since it avoids frequent re-allocation. If you want to reduce the space in an object to the minimum needed, you can do _mpz_realloc (&object, mpz_size (&object)).

The functions to initialize numbers are mpz_init (for MP_INT) and mpq_init (for MP_RAT).

mpz_init allocates space for the limbs, and stores a pointer to that space in the MP_INT object. It also stores the value 0 in the object.

In the same manner, mpq_init allocates space for the numerator and denominator limbs, and stores pointers to these spaces in the MP_RAT object.

To clear out a number object, use mpz_clear and mpq_clear, respectively.

Here is an example of use:

```
{
  MP_INT temp;
  mpz_init (&temp);

  ... store and read values in temp zero or more times ...

  mpz_clear (&temp):
}
```

You might be tempted to copy an integer from one object to another like this:

```
MP_INT x, y;

x = y;
```

Although valid C, this is an error. Rather than copying the integer value from y to x it will make the two variables share storage. Subsequent assignments to one variable would change the other mysteriously. And if you were to clear out both variables subsequently, you would confuse malloc and cause your program to crash.

To copy the value properly, you must use the function mpz_set. (see

Assigning Integers
)

1.7 gmp.info/Integer Functions

Integer Functions
****

This chapter describes the MP functions for performing integer arithmetic.

The integer functions use arguments and values of type pointer-to-MP_INT (see
Nomenclature
). The type MP_INT is a structure, but applications should not refer directly to its components. Include the header gmp.h to get the definition of MP_INT.

1.8 gmp.info/Initializing Integers

```
                Initializing Integer Objects
============================

   Most of the functions for integer arithmetic assume that the output
is stored in an object already initialized.  For example, mpz_add
stores the result of addition (see
                Integer Arithmetic
                ).  Thus, you must
initialize the object before storing the first value in it.  You can do
this separately by calling the function mpz_init.

 - Function: void mpz_init (MP_INT *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:

     {
       MP_INT integ;
       mpz_init (&integ);
       ...
       mpz_add (&integ, ...);
       ...
       mpz_sub (&integ, ...);

       /* Unless you are now exiting the program, 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 (MP_INT *integer )
```

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

 - Function: void * _mpz_realloc (MP_INT *integer , mp_size 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 (MP_INT integer_array [], size_t
          array_size , mp_size fixed_num_limbs )
     Allocate fixed limb space for all array_size integers in
     integer_array.  The fixed allocation for each integer in the array
     is fixed_num_limbs.  This function is useful for decreasing the
     working set for some algorithms that use large integer arrays.  If
     the fixed space will be insufficient for storing the result of a
     subsequent calculation, the result is unpredictable.

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

1.9 gmp.info/Assigning Integers

```
                Integer Assignment Functions
----------------------------

   These functions assign new values to already initialized integers
(see
                Initializing Integers
                ).

 - Function: void mpz_set (MP_INT *dest_integer , MP_INT *src_integer )
     Assign dest_integer from src_integer.

 - Function: void mpz_set_ui (MP_INT *integer , unsigned long int
          initial_value )
     Set the value of integer from initial_value.

 - Function: void mpz_set_si (MP_INT *integer , signed long int
          initial_value )
     Set the value of integer from initial_value.

 - Function: int mpz_set_str (MP_INT *integer , char *initial_value ,
          int base )
     Set the value of integer from initial_value, 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.info/Simultaneous Integer Init & Assign

```
Combined Initialization and Assignment Functions
------------------------------------------------
```

For your convenience, MP provides a parallel series of initialize-and-set arithmetic 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:

```
{
  MP_INT integ;
  mpz_init_set_str (&integ, "3141592653589793238462643383279502884", 10);
  ...
  mpz_sub (&integ, ...);

  mpz_clear (&integ);
}
```

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 (MP_INT *dest_integer , MP_INT *src_integer)

  Initialize dest_integer with limb space and set the initial numeric value from src_integer.

- Function: void mpz_init_set_ui (MP_INT *dest_integer , unsigned long int src_ulong)

  Initialize dest_integer with limb space and set the initial numeric value from src_ulong.

- Function: void mpz_init_set_si (MP_INT *dest_integer , signed long int src_slong)

  Initialize dest_integer with limb space and set the initial numeric value from src_slong.

- Function: int mpz_init_set_str (MP_INT *dest_integer , char *src_cstring , int base)

  Initialize dest_integer with limb space and set the initial numeric value from src_cstring, a '\0'-terminated C string in base base. The base may vary from 2 to 36. There may be white space in the string.

  If the string is a correct base base number, the function returns 0; if an error occurs it returns -1. dest_integer is initialized even if an error occurs. (I.e., you have to call mpz_clear for

```
     it.)
```

1.11 gmp.info/Converting Integers

```
                Conversion Functions
====================

 - Function: unsigned long int mpz_get_ui (MP_INT *src_integer )
     Return the least significant limb from src_integer.  This function
     together with
     mpz_div_2exp(..., src_integer, CHAR_BIT*sizeof(unsigned long int))
     can be used to extract the limbs of an integer efficiently.

 - Function: signed long int mpz_get_si (MP_INT *src_integer )
     If src_integer fits into a signed long int return the value of
     src_integer.  Otherwise return the least significant bits of
     src_integer, with the same sign as src_integer.

 - Function: char * mpz_get_str (char *string , int base , MP_INT
          *integer )
     Convert integer to a '\0'-terminated C string in string, using
     base base.  The base may vary from 2 to 36.  If string is NULL,
     space for the string is allocated using the default allocation
     function.

     If string 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 string, use mpz_sizeinbase (integer, base) +
     2.  The "+ 2" is for a possible minus sign, and for the
     terminating null character.  (see
                Miscellaneous Functions
                ).

     This function returns a pointer to the result string.
```

1.12 gmp.info/Integer Arithmetic

```
Integer Arithmetic Functions
============================

 - Function: void mpz_add (MP_INT *sum , MP_INT *addend1 , MP_INT
          *addend2 )

 - Function: void mpz_add_ui (MP_INT *sum , MP_INT *addend1 , unsigned
          long int addend2 )
     Set sum to addend1 + addend2.

 - Function: void mpz_sub (MP_INT *difference , MP_INT *minuend ,
          MP_INT *subtrahend )
```

```
 - Function: void mpz_sub_ui (MP_INT *difference , MP_INT *minuend ,
          unsigned long int subtrahend )
     Set difference to minuend - subtrahend.

 - Function: void mpz_mul (MP_INT *product , MP_INT *multiplicator ,
          MP_INT *multiplicand )

 - Function: void mpz_mul_ui (MP_INT *product , MP_INT *multiplicator ,
          unsigned long int multiplicand )
     Set product to multiplicator times multiplicand.

   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 powm 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 mpz_div (MP_INT *quotient , MP_INT *dividend , MP_INT
          *divisor )

 - Function: void mpz_div_ui (MP_INT *quotient , MP_INT *dividend ,
          unsigned long int divisor )
     Set quotient to dividend / divisor.  The quotient is rounded
     towards 0.

 - Function: void mpz_mod (MP_INT *remainder , MP_INT *divdend , MP_INT
          *divisor )

 - Function: void mpz_mod_ui (MP_INT *remainder , MP_INT *divdend ,
          unsigned long int divisor )
     Divide dividend and divisor and put the remainder in remainder.
     The remainder has the same sign as the dividend, and its absolute
     value is less than the absolute value of the divisor.

 - Function: void mpz_divmod (MP_INT *quotient , MP_INT *remainder ,
          MP_INT *dividend , MP_INT *divisor )

 - Function: void mpz_divmod_ui (MP_INT *quotient , MP_INT *remainder ,
          MP_INT *dividend , unsigned long int divisor )
     Divide dividend and divisor and put the quotient in quotient and
     the remainder in remainder.  The quotient is rounded towards 0.
     The remainder has the same sign as the dividend, and its absolute
     value is less than the absolute value of the divisor.

     If quotient and remainder are the same variable, the results are
     not defined.

 - Function: void mpz_mdiv (MP_INT *quotient , MP_INT *dividend ,
          MP_INT *divisor )

 - Function: void mpz_mdiv_ui (MP_INT *quotient , MP_INT *dividend ,
          unsigned long int divisor )
     Set quotient to dividend / divisor.  The quotient is rounded
     towards -infinity.
```

```
- Function: void mpz_mmod (MP_INT *remainder , MP_INT *divdend ,
        MP_INT *divisor )

- Function: unsigned long int mpz_mmod_ui (MP_INT *remainder , MP_INT
        *divdend , unsigned long int divisor )
    Divide dividend and divisor and put the remainder in remainder.
    The remainder is always positive, and its value is less than the
    value of the divisor.

    For mpz_mmod_ui the remainder is returned, and if remainder is not
    NULL, also stored there.

- Function: void mpz_mdivmod (MP_INT *quotient , MP_INT *remainder ,
        MP_INT *dividend , MP_INT *divisor )

- Function: unsigned long int mpz_mdivmod_ui (MP_INT *quotient ,
        MP_INT *remainder , MP_INT *dividend , unsigned long int
        divisor )
    Divide dividend and divisor and put the quotient in quotient and
    the remainder in remainder.  The quotient is rounded towards
    -infinity.  The remainder is always positive, and its value is
    less than the value of the divisor.

    For mpz_mdivmod_ui the remainder is small enough to fit in an
    unsigned long int, and is therefore returned.  If remainder is not
    NULL, the remainder is also stored there.

    If quotient and remainder are the same variable, the results are
    not defined.

- Function: void mpz_sqrt (MP_INT *root , MP_INT *operand )
    Set root to the square root of operand.  The result is rounded
    towards zero.

- Function: void mpz_sqrtrem (MP_INT *root , MP_INT *remainder ,
        MP_INT *operand )
    Set root to the square root of operand, as with mpz_sqrt.  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 not
    defined.

- Function: int mpz_perfect_square_p (MP_INT *square )
    Return non-zero if square is perfect, i.e. if the square root of
    square is integral.  Return zero otherwise.

- Function: int mpz_probab_prime_p (MP_INT *n , int reps )
    An implementation of the probabilistic primality test found in
    Knuth's Seminumerical Algorithms book.  If the function
    mpz_probab_prime_p(n, reps) returns 0 then n is not prime.  If it
    returns 1, then n is `probably' prime.  The probability of a false
    positive is (1/4)**reps, where reps is the number of internal
    passes of the probabilistic algorithm.  Knuth indicates that 25
    passes are reasonable.

- Function: void mpz_powm (MP_INT *res , MP_INT *base , MP_INT *exp ,
```

MP_INT *mod)

- Function: void mpz_powm_ui (MP_INT *res , MP_INT *base , unsigned long int exp , MP_INT *mod)
  Set res to (base raised to exp) modulo mod. If exp is negative, the result is undefined.

- Function: void mpz_pow_ui (MP_INT *res , MP_INT *base , unsigned long int exp)
  Set res to base raised to exp.

- Function: void mpz_fac_ui (MP_INT *res , unsigned long int n)
  Set res n!, the factorial of n.

- Function: void mpz_gcd (MP_INT *res , MP_INT *operand1 , MP_INT *operand2)
  Set res to the greatest common divisor of operand1 and operand2.

- Function: void mpz_gcdext (MP_INT *g , MP_INT *s , MP_INT *t , MP_INT *a , MP_INT *b)
  Compute g, s, and t, such that a s + b t = g = gcd (a, b). If t is NULL, that argument is not computed.

- Function: void mpz_neg (MP_INT *negated_operand , MP_INT *operand)
  Set negated_operand to -operand.

- Function: void mpz_abs (MP_INT *positive_operand , MP_INT *signed_operand)
  Set positive_operand to the absolute value of signed_operand.

- Function: int mpz_cmp (MP_INT *operand1 , MP_INT *operand2)

- Function: int mpz_cmp_ui (MP_INT *operand1 , unsigned long int operand2)

- Function: int mpz_cmp_si (MP_INT *operand1 , signed long int 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 mpz_mul_2exp (MP_INT *product , MP_INT *multiplicator , unsigned long int exponent_of_2)
  Set product to multiplicator times 2 raised to exponent_of_2. This operation can also be defined as a left shift, exponent_of_2 steps.

- Function: void mpz_div_2exp (MP_INT *quotient , MP_INT *dividend , unsigned long int exponent_of_2)
  Set quotient to dividend divided by 2 raised to exponent_of_2. This operation can also be defined as a right shift, exponent_of_2 steps, but unlike the >> operator in C, the result is rounded towards 0.

- Function: void mpz_mod_2exp (MP_INT *remainder , MP_INT *dividend , unsigned long int exponent_of_2)
  Set remainder to dividend mod (2 raised to exponent_of_2). The

```
     sign of remainder will have the same sign as dividend.

     This operation can also be defined as a masking of the
     exponent_of_2 least significant bits.
```

1.13 gmp.info/Logic on Integers

```
Logical Functions
=================

 - Function: void mpz_and (MP_INT *conjunction , MP_INT *operand1 ,
          MP_INT *operand2 )
     Set conjunction to operand1 logical-and operand2.

 - Function: void mpz_ior (MP_INT *disjunction , MP_INT *operand1 ,
          MP_INT *operand2 )
     Set disjunction to operand1 inclusive-or operand2.

 - Function: void mpz_xor (MP_INT *disjunction , MP_INT *operand1 ,
          MP_INT *operand2 )
     Set disjunction to operand1 exclusive-or operand2.

     This function is missing in the current release.

 - Function: void mpz_com (MP_INT *complemented_operand , MP_INT
          *operand )
     Set complemented_operand to the one's complement of operand.
```

1.14 gmp.info/I-O of Integers

```
Input and Output Functions
==========================

   Functions that perform input from a standard I/O stream, and
functions for output conversion.

 - Function: void mpz_inp_raw (MP_INT *integer , FILE *stream )
     Input from standard I/O stream stream in the format written by
     mpz_out_raw, and put the result in integer.

 - Function: void mpz_inp_str (MP_INT *integer , FILE *stream , int
          base )
     Input a string in base base from standard I/O stream stream, and
     put the read integer in integer.  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.

 - Function: void mpz_out_raw (FILE *stream , MP_INT *integer )
```

```
     Output integer on standard I/O 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.

 - Function: void mpz_out_str (FILE *stream , int base , MP_INT
          *integer )
     Output integer on standard I/O stream stream, as a string of
     digits in base base.  The base may vary from 2 to 36.
```

1.15 gmp.info/Rational Number Functions

```
Rational Number Functions
*************************

   All rational arithmetic functions canonicalize the result, so that
the denominator and the numerator have no common factors.  Zero has the
unique representation 0/1.

   The set of functions is quite small.  Maybe it will be extended in a
future release.

 - Function: void mpq_init (MP_RAT *dest_rational )
     Initialize dest_rational with limb space and set the initial
     numeric value 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 (MP_RAT *rational_number )
     Free the limb space occupied by rational_number.  Make sure to
     call this function for all MP_RAT variables when you are done with
     them.

 - Function: void mpq_set (MP_RAT *dest_rational , MP_RAT *src_rational
          )
     Assign dest_rational from src_rational.

 - Function: void mpq_set_ui (MP_RAT *rational_number , unsigned long
          int numerator , unsigned long int denominator )
     Set the value of rational_number to numerator/denominator.  If
     numerator and denominator have common factors, they are divided
     out before rational_number is assigned.

 - Function: void mpq_set_si (MP_RAT *rational_number , signed long int
          numerator , unsigned long int denominator )
     Like mpq_set_ui, but numerator is signed.

 - Function: void mpq_add (MP_RAT *sum , MP_RAT *addend1 , MP_RAT
          *addend2 )
     Set sum to addend1 + addend2.

 - Function: void mpq_sub (MP_RAT *difference , MP_RAT *minuend ,
          MP_RAT *subtrahend )
     Set difference to minuend - subtrahend.
```

```
 - Function: void mpq_mul (MP_RAT *product , MP_RAT *multiplicator ,
          MP_RAT *multiplicand )
     Set product to multiplicator * multiplicand

 - Function: void mpq_div (MP_RAT *quotient , MP_RAT *dividend , MP_RAT
          *divisor )
     Set quotient to dividend / divisor.

 - Function: void mpq_neg (MP_RAT *negated_operand , MP_RAT *operand )
     Set negated_operand to -operand.

 - Function: int mpq_cmp (MP_RAT *operand1 , MP_RAT *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 mpq_inv (MP_RAT *inverted_number , MP_RAT *number )
     Invert number by swapping the numerator and denominator.  If the
     new denominator becomes zero, this routine will divide by zero.

 - Function: void mpq_set_num (MP_RAT *rational_number , MP_INT
          *numerator )
     Make numerator become the numerator of rational_number by copying.

 - Function: void mpq_set_den (MP_RAT *rational_number , MP_INT
          *denominator )
     Make denominator become the denominator of rational_number by
     copying.  If denominator < 0 the denominator of rational_number is
     set to the absolute value of denominator, and the sign of the
     numerator of rational_number is changed.

 - Function: void mpq_get_num (MP_INT *numerator , MP_RAT
          *rational_number )
     Copy the numerator of rational_number to the integer numerator, to
     prepare for integer operations on the numerator.

 - Function: void mpq_get_den (MP_INT *denominator , MP_RAT
          *rational_number )
     Copy the denominator of rational_number to the integer
     denominator, to prepare for integer operations on the
     denominator.
```

1.16 gmp.info/Low-level Functions

```
Low-level Functions
*******************

   The next release of the GNU MP library (2.0) will include changes to
some mpn functions.  Programs that use these functions according to the
descriptions below will therefore not work with the next release.

   The low-level function layer is designed to be as fast as possible,
not to provide a coherent calling interface.  The different functions
```

have similar interfaces, but there are variations that might be confusing. 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 responsability of the caller to ensure that the destination has enough space for storing the result.

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

All these functions require that the operands are normalized in the sense that the most significant limb must be non-zero. (A future release of might drop this requirement.)

The low-level layer is the base for the implementation of the mpz_ and mpq_ layers.

The code below adds the number beginning at src1_ptr and the number beginning at src2_ptr and writes the sum at dest_ptr. A constraint for mpn_add is that src1_size must not be smaller that src2_size.

```
mpn_add (dest_ptr, src1_ptr, src1_size, src2_ptr, src2_size)
```

In the description below, a source operand is identified by the pointer to the least significant limb, and the limb count in braces.

- Function: mp_size mpn_add (mp_ptr dest_ptr , mp_srcptr src1_ptr , mp_size src1_size , mp_srcptr src2_ptr , mp_size 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. Carry-out, either 0 or 1, is returned.

  This function requires that src1_size is greater than or equal to src2_size.

- Function: mp_size mpn_sub (mp_ptr dest_ptr , mp_srcptr src1_ptr , mp_size src1_size , mp_srcptr src2_ptr , mp_size src2_size)

  Subtarct {src2_ptr, src2_size } from {src1_ptr, src1_size }, and write the result to dest_ptr.

  Return 1 if the minuend < the subtrahend. Otherwise, return the negative difference between the number of words in the result and the minuend. I.e. return 0 if the result has src1_size words, -1 if it has src1_size - 1 words, etc.

  This function requires that src1_size is greater than or equal to src2_size.

- Function: mp_size mpn_mul (mp_ptr dest_ptr , mp_srcptr src1_ptr , mp_size src1_size , mp_srcptr src2_ptr , mp_size src2_size)

  Multiply {src1_ptr, src1_size } and {src2_ptr, src2_size }, and write the result to dest_ptr. The exact size of the result is returned.

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 mpn_div (mp_ptr dest_ptr , mp_ptr src1_ptr , mp_size src1_size , mp_srcptr src2_ptr , mp_size src2_size)
  Divide {src1_ptr, src1_size } by {src2_ptr, src2_size }, and write the quotient to dest_ptr, and the remainder to src1_ptr.

  Return 0 if the quotient size is at most (src1_size - src2_size), and 1 if the quotient size is at most (src1_size - src2_size + 1). The caller has to check the most significant limb to find out the exact size.

  The most significant bit of the most significant limb of the divisor has to be set.

  This function requires that src1_size is greater than or equal to src2_size. The quotient, pointed to by dest_ptr, must be distinct from either input operands.

- Function: mp_limb mpn_lshift (mp_ptr dest_ptr , mp_srcptr src_ptr , mp_size 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 limb shifted out is returned.

  Overlapping of the destination space and the source space is allowed in this function, provdied dest_ptr >= src_ptr.

- Function: mp_size mpn_rshift (mp_ptr dest_ptr , mp_srcptr src_ptr , mp_size src_size , unsigned long int count)
  Shift {src_ptr, src_size } count bits to the right, 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 size of the result is returned.

  Overlaping of the destination space and the source space is allowed in this function, provdied dest_ptr <= src_ptr.

- Function: mp_size mpn_rshiftci (mp_ptr dest_ptr , mp_srcptr src_ptr , mp_size src_size , unsigned long int count , mp_limb inlimb)
  Like mpn_rshift, but use inlimb to feed the least significant end of the destination.

- Function: int mpn_cmp (mp_srcptr src1_ptr , mp_srcptr src2_ptr , mp_size size)
  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.

1.17 gmp.info/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*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.

 - 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.

Some implementations of this function return a remainder whose sign is inverted if the divisor is negative. Such a definition makes little sense from a mathematical point of view. GNU MP might be considered incompatible with the traditional MP in this respect.

- Function: void msqrt (MINT *operand , MINT *root , MINT *remainder)
  Set root to the square root of operand, as with mpz_sqrt. Set remainder to operand-root*root, (i.e. zero if operand is a perfect square).

- 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. (See Initialization , for an explanation of the memory allocation in MP).

- 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.

1.18 gmp.info/Miscellaneous Functions

Miscellaneous Functions
****

```
 - Function: void mpz_random (MP_INT *random_integer , mp_size max_size
          )
     Generate a random integer of at most max_size limbs.  The generated
     random number doesn't satisfy any particular requirements of
     randomness.

 - Function: void mpz_random2 (MP_INT *random_integer , mp_size
          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 bugs.

 - Function: size_t mpz_size (MP_INT *integer )
     Return the size of integer measured in number of limbs.  If
     integer is zero, the returned value will be zero, if integer
     has one limb, the returned value will be one, etc.  (See

                Nomenclature
                , for an explanation of the concept limb.)

 - Function: size_t mpz_sizeinbase (MP_INT *integer , int base )
     Return the size of integer 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 integer 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').
```

1.19 gmp.info/Custom Allocation

```
Custom Allocation
=================

   By default, the initialization functions use malloc, realloc, and
free to do their work.  If malloc or realloc fails, the MP package
terminates execution after a printing fatal error message on standard
error.

   In 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 functions
for allocating and de-allocating memory.  Use mp_set_memory_functions
to do this.

   mp_set_memory_functions has three arguments, allocate_function,
reallocate_function, and deallocate_function, in that order.  If an
argument is NULL, the corresponding default function is retained.
```

The functions you supply should fit the following declarations:

```
void * allocate_function (size_t alloc_size)
     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)
     This function should return a pointer to newly allocated space of
     at least new_size storage units, after copying 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 formely returned from
     allocate_function or reallocate_function, for a request for
     old_size storage units.

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 formely 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.)

NOTE: call mp_set_memory_functions only before calling any other MP functions. Otherwise, the user-defined allocation functions may be asked to re-allocate or de-allocate something previously allocated by the default allocation functions.

1.20 gmp.info/Reporting Bugs

Reporting Bugs
****

If you think you have found a bug in the GNU 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.

Please make sure that the bug is really in the GNU MP library.

You have to send us a test case that makes it possible for us to reproduce the bug.

You also have to explain what is wrong; if you get a crash, or if the results printed are not good and in that case, in what way.

Make sure that the bug report includes all information you would need to fix this kind of bug for someone else. Think twice.

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: tege@gnu.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.

1.21 gmp.info/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 Cummins Publishing Company Inc, 1981.

* Richard M. Stallman, "Using and Porting GCC", Free Software Foundation, 1993.

* Peter L. Montgomery, "Modular Multiplication Without Trial Division", Mathematics of Computation, volume 44, number 170, April 1985.

1.22 gmp.info/Concept Index

Concept Index
****

1.23 gmp.info/Function Index

Function and Type Index
****