# DOCUMENTATION OF LIP

Arjen K. Lenstra

## 1. Introduction

This is a users manual for LIP, a package containing a variety of functions for arithmetic on arbitrary length signed integers that is available from Bellcore. These functions allow easy prototyping and experimentation with new number theory-based cryptographic protocols. LIP is written entirely in ANSI C, has proved to be easily portable, is intended to be easy to use, and achieves an acceptable performance on many different platforms, including 64-bit architectures. We assume that the reader is familiar with ANSI C.

This documentation is organized as follows. The remainder of this section describes how to get started with LIP, and how programs using one of the earlier versions of LIP should be changed to use the present version. Section 2 contains a systematic listing and description of all available functions. With Section 1 this should suffice for most straightforward applications. (An alphabetic listing of all functions is given in Appendix A.) A few machine dependent customizations might have to be carried out before LIP can be compiled and used; they are given in Section 3. Since LIP is written entirely in ANSI C it is often possible to achieve greater efficiency by replacing a few internal macros by assembly language versions; Section 4 describes the relevant macros. Section 5 discusses some of the LIP-functions in detail. A complete overview of all compilation flags is given in Appendix B.

The remainder of this section describes how LIP can be obtained and compiled, it gives two examples of the usage of LIP, it discusses the main differences with earlier versions of LIP, describes the error message mechanism in LIP, how arbitrary length integers are internally represented, how the allocation mechanism works, where bugs in LIP should be reported, and concludes with a remark that might be useful.

(1.1) **Obtaining LIP**. LIP is available for free for research or educational purposes. Some LIP related files can be obtained by anonymous `ftp` from the directory `/usr/spool/ftp/pub/lenstra` on `flash.bellcore.com`. This directory contains the following files: `agreement.ps`, `agreement.plain`, `lipdoc.ps.Z`,

Thanks to Bob Cain, Scott Contini, Roger Golliver, Achim Flammenkamp, Daniel Grove, Paul Leyland, Mark Riordan, Selwyn Russell, and Victor Shoup for their contributions and suggestions, and to many users for reporting bugs.

March 6, 1995, version 0.5.

and `README`. The file `lipdoc.ps.Z` contains a compressed version of this documentation. Upon receipt of a signed and completed paper copy of either `agreement.ps` or `agreement.plain` the author will email the file `lip.tar.Z`, a compressed tarfile containing `lip.c`, `lip.h`, `liptimer.c`, `lipdoc.ps`, and `Makefile`[1]. These files can be extracted as follows[2]:

```
        uncompress lip.tar.Z
        tar xf lip.tar
```

(1.2) **Compiling LIP**. Once the files have been extracted, `lip.c` should be compiled once and for all to obtain `lip.o`:

```
        ( make lip.o ) >& liptimer.out
```

The file `Makefile` assumes that '`gcc`' is available as ANSI C compiler. It uses optimization level '`-O2`'. Edit `Makefile` if another compiler or optimization level should be used.

This initial compilation is rather involved. A default `lip.o` is created and used to run the timer and test program `liptimer.c`. This results in the outputfile `liptimer.out` and an auxiliary file `lippar.h`. The information in `lippar.h` is then used to create another version of `lip.o` with supposedly better choices of various compile-time options. The choices made by `liptimer.c` are based on the average multiplication speeds of numbers of various sizes (which can, for several compile-time options, be found in `liptimer.out`). See Section 3 how `liptimer.c` can be customized. Of course, `lippar.h` can also be changed manually based on the information in `liptimer.out`, after which LIP can be re-compiled.

If the compilation does not work, if `liptimer.c` does not execute properly, or if `liptimer.out` indicates that something does not work, LIP has to be customized. We refer to Section 3 for the settings of some machine dependent constants. Besides the flags that are affected by `lippar.h`, LIP can be compiled with a variety of other flags, which are described at appropriate points in the text below. The flag needed for 64-bit architectures ('`-DALPHA`') has to be set by hand, and is not automatically tried by `liptimer.c`.

From now on we assume that `lip.o` has been created successfully.

(1.3) **First example**. In LIP the type `verylong` is used for arbitrary length signed integers. All variables of type `verylong` have to be initialized as `0` (zero), which is most conveniently done upon declaration; thus, to declare a `verylong` variable `a`, write:

```
        verylong a = 0;
```

The following program, `example.1.c`, reads the decimal representation of two integers from standard input into the `verylong` variables `a` and `b`, puts the product of `a` and `b` in the `verylong` variable `c` using the function `zmul`, and writes the decimal representation of `c` followed by a newline to standard output.

[1] The version of LIP that was used in the RSA-129 project (cf. [1]) can be `ftp`-ed from `ftp.ox.ac.uk:/pub/math/freelip/freelip_1.0.tar.gz`. This version is slightly older than but fully compatible with the version described here that is available from Bellcore.

[2] For `freelip_1.0.tar.gz` use:

```
        gunzip freelip_1.0.tar.gz
        tar xf freelip_1.0.tar
```

```
#include "lip.h"
main()
{
  verylong a = 0;
  verylong b = 0;
  verylong c = 0;
  zread(&a);
  zread(&b);
  zmul(a,b,&c);
  zwriteln(c);
}
```

After compilation using

```
gcc [ -O[level] ] -o example.1 example.1.c lip.o -lm
```

or

```
make example.1
```

execution of `example.1` on input

```
3199044596370769   -312593329000312\
593329
```

produces

```
-1000000000000000000000000000000000001
```

on standard output. Notice the backslash indicating that the second number on input continues on the next line; backslashes are also used by the output functions to split long numbers over two or more lines (but see (3.3)). If instead of the above output another number or the message

```
zstart failure: recompile with smaller NBITS
```

appears, then LIP has to be customized (see Section 3).

Program `example.1.c` shows two basic principles for the use of functions on `verylong` arguments. In the first place, if the value of a `verylong` is only used as input to a function, just give the name of the `verylong` as argument to that function. For instance, `a` and `b` are inputs to the multiplication function `zmul`, and `c` is the input of `zwriteln`. In the second place, if a `verylong` might get a new value as the result of a function, give the address of the `verylong` (i.e., its name preceded by a `&`) as argument to that function. For instance, both `a` and `b` will get new values as result of the calls to `zread`, and `c` will get the product of `a` and `b` as result of the call to `zmul`. The include file `lip.h` contains prototypes of all LIP-functions, so the compiler will print a warning each time a function is called with an incorrect argument.

Besides `zmul`, `zread`, and `zwriteln`, other useful basic functions are:

| | |
|---|---|
| `zadd(a,b,&c)` | `c` gets the value $\mathtt{a} + \mathtt{b}$, |
| `zsub(a,b,&c)` | `c` gets the value $\mathtt{a} - \mathtt{b}$, |
| `zsq(a,&b)` | `b` gets the value $\mathtt{a}^2$, |
| `zdiv(a,b,&c,&d)` | `c` gets the value $[\mathtt{a}/\mathtt{b}]$ and `d` gets `a` mod `b`, |
| `zmod(a,b,&c)` | `c` gets the value `a` mod `b`, |
| `zexpmod(a,b,c,&d)` | `d` gets the value $\mathtt{a}^{\mathtt{b}}$ mod `c`, |
| `zcompare(a,b)` | returns the sign of $\mathtt{a} - \mathtt{b}$, |

zcopy(a,&b) b gets the same value as a, and
zintoz(n,&a) converts a regular long n to a verylong a,

where d is also of type verylong, and where " c gets the value a + b " stands for " the arbitrary length signed integer that is represented by the verylong c gets the value of the sum of the integers represented by the verylongs a and b ". A complete listing of the available functions is given in the next section.

Most basic functions allow identical arguments for inputs and outputs (like ‘zadd(a,b,&a)’ or ‘zadd(a,a,&a)’), but some do not (like zmul, zsq, and zexpmod); the exceptions are documented in the listing below. If in doubt use different arguments, and certainly do not use identical arguments for outputs (i.e., the value of c is undefined after ‘zdiv(a,b,&c,&c)’).

Although it is possible, it is not recommended to use ‘b = a’ to give b the same value as a, because the effect is unpredictable and entirely different from ‘zcopy(a,&b)’: after ‘b = a’, a change of a’s (or b’s) value might or might not affect b (or a)—after ‘zcopy(a,&b)’ a change of a’s (or b’s) value does not affect b (or a).

Please observe the difference between statements like " ‘b = a’ " and " b = a ". The first refers to a statement in C, and is consistently put between ‘ and ’ when used in text. The latter means that the value of the integer represented by b is the same as the value of the integer represented by a; similarly b = 0 means that the value of the integer represented by b is 0.

(1.4) **Second example**. The following program, example.2.c, uses the built-in function zrandomprime to generate five more or less random probable primes of binary lengths 64, 80, 96, 112, 128; the resulting primes are printed in hexadecimal on standard output. An explanation of the second argument of zrandomprime, the 5 in the call below, can be found in (2.9) and (2.10). In its last argument zrandomprime expects a function u that upon call u(b,&c), for verylongs b and c, gives c a random verylong value in the range $[0, b-1]$. In example.2.c the built-in pseudo random generator zrandomb is used. To get different primes for different runs of example.2, zrandomb is initialized with a user selected seed, using a call to the zrandomb-initialization function zrstart (actually, zrstart initializes zrandom, and zrandomb uses zrandom). If the call to zrstart is omitted, zrandomb will start from the default seed 7157891 (which is, in a certain sense, random). Use of zrandom or zrandomb cannot be recommended for generation of ‘cryptographically strong’ primes, like prospective secret factors of composite moduli. In such cases the user should use his/her own random generator in the call to zrandomprime.

```
#include "lip.h"
main ()
{
  verylong seed = 0;
  verylong prime = 0;
  long bitlength;
  zread(&seed);
  zrstart(seed);
```

```
      for (bitlength = 64; bitlength <= 128; bitlength += 16)
      {
        if (zrandomprime(bitlength,5,&prime,zrandomb))
        {
          printf("%3d bits: ",bitlength);
          zhwriteln(prime);
        }
        else
          printf("no %3d bit prime found\n",bitlength);
      }
    }
```

Execution of `example.2` on input

```
3199044596370769
```

might produce[3]

```
 64 bits: E94E89DB AA1D02EB
 80 bits: 9DB0CE8B 34B1481B 737D
 96 bits: DA819735 AFDC454E ECF4A5BD
112 bits: E60EE460 5F668D38 A747F450 E099
128 bits: B90BBA22 A0728D77 07FEBD61 AFD3D9B5
```

on standard output, whereas input

```
-3125933290003125933329
```

produces entirely different random primes on standard output:

```
 64 bits: BE9D28E8 37B57E55
 80 bits: EA98EEC5 5F8CBE9E 4DC9
 96 bits: E64B0DD6 A86D7269 AD82E6FF
112 bits: 86C93C53 0B724B3F 71981F72 AE03
128 bits: F85FC716 32CDE29F 8BAF5B52 5A050463
```

If `zrandomb` is initialized with `seed` equal to `0`, then it will only produce zeros, and the primes found by `zrandomprime` look considerably less random: on input

```
0
```

`example.2` produces the smallest primes of the specified sizes:

```
 64 bits: 80000000 0000001D
 80 bits: 80000000 00000000 0017
 96 bits: 80000000 00000000 00000009
112 bits: 80000000 00000000 00000000 0033
128 bits: 80000000 00000000 00000000 0000001D
```

The above hexadecimal output format (a space every eighth digit) can also be used for input in hexadecimal (using `zhread`): unlike `zread` a space does not indicate the end of the number being read, but a newline does, unless it is preceded by a backslash (as in `zread`).

(1.5) **Early versions of LIP**. The original fixed length unsigned version of LIP, `basic.c`, is part of the code that is distributed over the Internet in the 'Factoring by email' project (cf. [5]). It was also available for some time from the

[3] The values produced depend on the contents of `lippar.h` and might thus vary from machine to machine.

ripem.msu.edu ftp server. Because it uses fixed length integers, the parameter passing mechanism in basic.c is different from LIP, which makes the two packages incompatible unless the following changes are made. Declarations of the form

```
        long a[ZSIZEP], b[ZSIZEP], c[ZZSIZEP];
```

have to be rewritten as

```
        verylong a = 0, b = 0, c = 0;
```

and calls like 'zadd(a,b,c)' have to be changed into 'zadd(a,b,&c)'. After inclusion of lip.h the compiler should be able to indicate which changes of that sort have to be made. Finally, quite a few function names and parameters meanings have changed; a non-exclusive list is given below.

Another early version of LIP, lenstra-3.1.c, used to be available from the same ftp server ripem.msu.edu. That version is compatible with LIP, except that we have tried to assign the function names in LIP in a more consistent manner. With the exception of the timing-functions, the names of all functions operating on verylong arguments are of the form zxxx, where xxx is supposed to give a rough indication of what the function does. For some functions there may be two variants: zsxxx where one of the operands (usually the second) is an ordinary long, and zxxxs where all operands are ordinary longs. In some cases this led to possible ambiguities (like zsq or zsub), but these should not lead to serious problems. The following changes have been made:

| name in lenstra-3.1.c | name in LIP | |
|---|---|---|
| default_m | zdefault_m | |
| makeodd | zmakeodd | |
| pollardrho | zpollardrho | |
| zdivide2 | z2div | |
| zexp2mod | z2expmod | |
| zexps | zsexp | |
| zexpsmod | zsexpmod | |
| zmexp | zmontexp | |
| zmexp_m_ary | zmontexp_m_ary | |
| zmont | zmontmul | |
| zrstart | zrstarts | (LIP has a new zrstart) |
| zsexp | zexpmods | |
| zsinv | zinvs | |
| zsjacobi | zjacobis | |
| zsmulmod | zmulmods | (LIP has a new zsmulmod) |
| zsodinv | zinvodds | |
| zssqrt | zsqrts | |
| ztimes2 | z2mul | |

Furthermore, in LIP we have adopted the rule that a verylong argument should *always* be of type *verylong if its value might change as a result of the function call, even though its allocated space is guaranteed not to change (which makes the '*' in principle superfluous). This led to most of the following changes:

| lenstra-3.1.c | LIP |
|---|---|

```
makeodd(verylong)           zmakeodd(*verylong)
pollardrho(verylong,        zpollardrho(verylong,
    *verylong,*verylong)        *verylong,*verylong,long)
zcomposite(verylong,        zcomposite(*verylong,
    long,long)                  long,long)
zfree(verylong)             zfree(*verylong)
zmulin(*verylong,verylong)  zmulin(verylong,*verylong)
znegate(verylong)           znegate(*verylong)
```

Again, the compiler should be able to indicate where changes have to be made.

(1.6) **Error messages in LIP**. If LIP detects an error a message is printed on standard error and the program exits. LIP can be forced to continue after an error has been detected by compiling it with the '-DNO_HALT' flag. The error messages are supposed to be self-explanatory, but if they are not more information can be found both in Section 2, in lip.h in the description of the function mentioned in the error message, or, as a last resort, in the mostly undocumented source-file lip.c.

To illustrate this, the following program, example.3.c, reads an ordinary long p from standard input, and uses the function zinvodds to compute $\mathtt{i}^{-1}$ modulo p for $1 \leq \mathtt{i} < \mathtt{p}$. Clearly, $\mathtt{i}^{-1}$ modulo p only exists if i and p are coprime, so that an error can be expected if i and p have a factor in common.

```
#include "lip.h"
main ()
{
  register long i;
  long p;
  scanf("%d",&p);
  for (i=1;i<p;i++)
    printf("%d inverse modulo %d is %d\n",
      i,p,zinvodds(i,p));
}
```

Execution of example.3 on input

```
5
```

produces

```
1 inverse modulo 5 is 1
2 inverse modulo 5 is 3
3 inverse modulo 5 is 2
4 inverse modulo 5 is 4
```

on standard output. But execution on input

```
15
```

leads to an error message because 3 divides 15 and therefore does not have an inverse modulo 15:

```
1 inverse modulo 15 is 1
2 inverse modulo 15 is 8
fatal error:
   arguments not coprime in zinvodds
```

```
        exit...
```

The first two lines appear on standard output, the remaining three on standard error. To separate the output into two files, `example.3` can be executed using

```
        ( example.3 > example.3.out ) >& example.3.err
```

Compilation of LIP with `-DNO_HALT` and execution of (the newly compiled) `example.3` on input

```
        9
```

produces:

```
        1 inverse modulo 9 is 1
        2 inverse modulo 9 is 5
        error:
           arguments not coprime in zinvodds
        continue...
        3 inverse modulo 9 is 0
        4 inverse modulo 9 is 7
        5 inverse modulo 9 is 2
        error:
           arguments not coprime in zinvodds
        continue...
        6 inverse modulo 9 is 0
        7 inverse modulo 9 is 4
        8 inverse modulo 9 is 8
```

Some error messages are of the form ‘`...      BUG`’. This either means that one or more machine dependent constants have incorrect values (cf. Section 3), or it means that a bug in LIP has been detected. The latter should be reported to the author at `lenstra@bellcore.com`.

(1.7) **Representation of arbitrary long integers in LIP**. The type `verylong` is declared as

```
        typedef long * verylong
```

which implies that `verylong` variables are internally represented as arrays of `longs`. The user does not have to worry about the allocation of these arrays, as long as each `verylong` variable gets the value 0 before its first use. As an example, the following program fragment should work fine:

```
        long i;
        verylong a;
        verylong row[10];
        a=0;
        for (i=0;i<10;i++)
          row[i]=0;
        for (i=0;i<10;i++)
          zread(&(row[i]));
        for (i=0;i<10;i++)
          zmulin(row[i],&a);
```

But a similar fragment, without the initialization of the `verylongs`, might lead to unexpected results and is incorrect:

```
long i;
verylong a;
verylong row[10];
for (i=0;i<10;i++)
  zread(&(row[i]));/* wrong: using uninitialized row[i] */
for (i=0;i<10;i++)
  zmulin(row[i],&a);/* wrong: using uninitialized a */
```

If `a` is a `verylong` variable, then either the C-statement '`a == 0`' is true and the value of the arbitrary length integer represented by `a` will be interpreted as 0, or space for `a` has actually been allocated, and '`a == 0`' is false. In the latter case the absolute value of the arbitrary length integer represented by `a` equals

$$\sum_{i=1}^{|\mathsf{a}[0]|} \mathtt{a}[i] \cdot \mathtt{RADIX}^{i-1},$$

and its sign equals the sign of $\mathtt{a}[0]$. Here `RADIX` equals $2^{\mathtt{NBITS}}$, where `NBITS` is a (small) integer satisfying the requirements mentioned in Section 3. Furthermore, $0 \le \mathtt{a}[i] < \mathtt{RADIX}$ for $i = 1, 2, \ldots, |\mathtt{a}[0]| - 1$, and $0 < \mathtt{a}[i] < \mathtt{RADIX}$ for $i = |\mathtt{a}[0]|$ except when $\mathtt{a}[0] = 1$ in which case $\mathtt{a}[1]$ may be equal to zero.

The amount of space currently allocated for `a` is kept in $\mathtt{a}[-1]$; the LIP-functions check this location to see if reallocation is needed. The values of $\mathtt{a}[|\mathtt{a}[0]| + 1]$ through $\mathtt{a}[\mathtt{a}[-1]]$ are undefined and cannot be assumed to be zero.

Although the $\mathtt{a}[i]$ are accessible to the user, it is not recommended to access these values directly, or to change them other than by applying any of the LIP-functions. In particular changing the value of $\mathtt{a}[-1]$ may have undesirable side-effects.

(1.8) **Allocation**. LIP uses `calloc` and `realloc` to (re)allocate space for `verylong`s, as described in (2.13). All local `verylong` variables in any of the functions in LIP are declared as `static`. In this way space for those variables will only be allocated during the first call to a function, unless more space is needed than in any of the previous calls and the `verylong` has to be reallocated. So, all local `verylong` variables keep the last (and largest) length that has ever been allocated for them during the present run.

If, however, LIP is compiled with the '`-DFREE`' flag, local `verylong` variables are not made `static`, and the space that has been allocated during a call will be explicitly freed at the end of the function (using the function `zfree`, see (2.13)). Using this compilation flag might be preferable in circumstances where memory is scarce.

Similarly, in frequently used user-defined non-recursive functions that use `verylong` local variables, it may be more efficient to declare these variables as `static verylong`. In recursive functions the `verylong` variables should, in general, not be made `static`, and they should be freed at the end of the function.

(1.9) **Bugs in LIP**. Although LIP has been tested and used extensively for several years and on many different platforms, there are certainly several bugs left. The author would very much appreciate if bugs are reported directly to him at `lenstra@bellcore.com` or at his regular mail address.

(1.10) **Remark**. Be careful with local changes (in a user-defined function) to a `verylong` argument that is passed "by value" (i.e., without a '`&`'): for ordinary `long`s a local change will have no effect outside the function, for `verylong`s this cannot be guaranteed. As an example, consider the following function:

```
no_good(verylong a)
{
  zsadd(a,1,&a);
  zwriteln(a);
}
```

If `a` has value 5 before the call `no_good(a)`, then 6 will be printed, and `a` will have value 6 after the call. If however '`a == 0`' was still true before the call, then 1 will be printed, and after the call '`a == 0`' will still be true. Examples with unpredictable behavior can easily be constructed. Use of this 'feature' is not recommended.

## 2. Available functions

This section contains a systematic listing of all function-prototypes, including a description of each function, subdivided in the following categories:

| | | |
|---|---|---|
| 1 | Auxiliary functions: | comparison, (base)conversion, copying, logarithms, sign manipulations |
| 2 | Basic arithmetic: | $+$, $-$, $*$, $/$, mod, powering, squaring |
| 3 | Input/output: | (hexa)decimal, arbitrary base, machine dependent |
| 4 | Bit manipulation: | and, cat, not, (x)or, parity, shift, weight, get/setbits, high/lowbits, reverse, switch |
| 5 | Modular arithmetic: | $+$, $-$, $*$, $/$, powering, squaring |
| 6 | Montgomery arithmetic: | initialization, $+$, $-$, $*$, $/$, powering, squaring |
| 7 | Euclidean algorithms: | chinese remaindering, (extended) gcd, inverse, jacobi symbol |
| 8 | Random number generation: | initialization, fixed length, bounded |
| 9 | Primality testing and factoring: | probabilistic tests, trial division, squfof, pollard rho, elliptic curve method |
| 10 | Prime generation: | small primes, fixed length, bounded, fixed (length or prime) divisor, with generator |
| 11 | NIST digital signature algorithm: | key generation, signing, verification |
| 12 | Timing functions: | system time, user time |
| 13 | Allocation: | get/free space |

In the descriptions below, please remember the distinction between "'`b = a`'" and "`b` = `a`" and the meaning of "`a` gets the value ...", which are explained in (1.3).

(2.1) **Auxiliary functions**

```
void zstart(void)
```

Initializes some global variables that are used in the division functions. This function will be called automatically, unless LIP is compiled with the '`-DSTART`' flag, in which case LIP expects an explicit user-call to `zstart` at the beginning of all programs that use LIP, before any other LIP-functions is used. The message `zstart failure: recompile with smaller NBITS` is always fatal, and indicates that the machine dependent constant `NBITS` has an incorrect value (cf. Section 3).

`long zscompare(verylong a, long b)`
`long zcompare(verylong a, verylong b)`

Return 1 if $\mathtt{a} > \mathtt{b}$, return 0 if $\mathtt{a} = \mathtt{b}$, return $-1$ otherwise.

`long ziszero (verylong a)`

Returns 1 if $\mathtt{a} = 0$, returns 0 otherwise.

`long zsign(verylong a)`

Returns 1 if $\mathtt{a} > 0$, returns 0 if $\mathtt{a} = 0$, returns $-1$ otherwise.

`void zzero(verylong *a)`

`a` gets the value 0.

`void zone(verylong *a)`

`a` gets the value 1.

`void zintoz(long d, verylong *a)`

`a` gets the value `d`.

`void zuintoz(unsigned long d, verylong *a)`

`a` gets the value `d`.

`void zultoz(unsigned long a[], long b, verylong *c)`

`c` gets the value $\sum_{i=0}^{\mathtt{b}-1} \mathtt{a}[i]r^i$, with $r = 2^{\mathtt{CHARL}\times\mathtt{SIZEOFLONG}}$, to convert '`unsigned long`' base $r$ representation in `a` to `verylong` in `c`.

`long ztoint(verylong a)`

Returns attempted conversion of `a`'s value to `long`, **no** overflow check.

`unsigned long ztouint(verylong a)`

Returns attempted conversion of $|\mathtt{a}|$'s value to `unsigned long`, **no** overflow check.

`long ztoul(verylong a, unsigned long b[], long *c)`

Set `c` to $m$ and $\mathtt{b}[i]$ such that $|\mathtt{a}| = \sum_{i=0}^{m-1} \mathtt{b}[i]r^i$, with $0 \leq \mathtt{b}[i] < r$, with $r = 2^{\mathtt{CHARL}\times\mathtt{SIZEOFLONG}}$, to convert $|\mathtt{a}|$ to an '`unsigned long`' base $r$ representation in `b`. Returns 0 if `c` on input (which should be the length of `b`) is $< m$; returns 1 otherwise.

`long zstrtozbas(char *a, long b, verylong *c)`
`long zstrtoz(char *a, verylong *c)`

`c` gets the value of the string `a` whose characters are interpreted as base $|\mathtt{b}|$ digits (with $\mathtt{b} = 10$ for `zstrtoz`), until a non-digit or a digit $\geq |\mathtt{b}|$ is encountered, for $0 < |\mathtt{b}| \leq 16$. The first character of `a` may be a '-' or a '_' to indicate a negative number. Return the numbers of digits read from `a` (excluding the sign), return zero if $\mathtt{b} = 0$ or if $|\mathtt{b}| > 16$. Notice that, unlike `zsread`, these functions read `a` until the first character that cannot be interpreted as a base $|\mathtt{b}|$ digit.

`double zdoub(verylong a)`

Returns attempted conversion of `a`'s value to `double`, **no** overflow check.

`void zsbastoz(long a, long b[], long c, verylong *d)`
`void zbastoz(verylong a, verylong b[], long c, verylong *d)`

`d` gets the value $\sum_{i=0}^{\mathtt{c}-1} \mathtt{b}[i]\mathtt{a}^i$, to convert base `a` representation in `b` to `verylong` in `d`.

`long zstobas(verylong a, long b, long c[], long *d)`
`long ztobas(verylong a, verylong b, verylong c[], long *d)`

Set `d` to $m$ and $\mathtt{c}[i]$ such that $|\mathtt{a}| = \sum_{i=0}^{m-1} \mathtt{c}[i]|\mathtt{b}|^i$, with $0 \leq \mathtt{c}[i] < |\mathtt{b}|$, to convert $|\mathtt{a}|$ to a base $|\mathtt{b}|$ representation in `c`. Return 0 if `d` on input (which should be the length of `c`) is $< m$, or if $|\mathtt{b}| \leq 1$; return 1 otherwise.

`long zstosymbas(verylong a, long b, long c[], long *d)`
`long ztosymbas(verylong a, verylong b, verylong c[], long *d)`

As `zstobas` and `ztobas`, except that $|\mathtt{c}[i]| \leq |\mathtt{b}|/2$, to convert $|\mathtt{a}|$ to a symmetric base $|\mathtt{b}|$ representation in `c`.

`void zcopy(verylong a, verylong *b)`

`b` gets the value `a`—different from '`b = a`'.

`void zswap(verylong *a, verylong *b)`

Exchanges `a` and `b`.

`long z2logs(long a)`
`long z2log(verylong a)`

Return 0 if $\mathtt{a} = 0$, return $[1 + \log_2 |\mathtt{a}|]$ otherwise (the number of binary bits needed to represent $|\mathtt{a}|$).

`double zslog(verylong a, long b)`
`double zdlog(verylong a, double b)`
`double zlog(verylong a, verylong b)`

Return `double` approximation of the base `b` logarithm of `a`, only for $\mathtt{a} > 0$ and $\mathtt{b} > 1$.

`double zln(verylong a)`

Returns `double` approximation of the natural logarithm of `a` if $\mathtt{a} > 0$, error message '`non-positive argument in zln`' otherwise; result undefined if error occurs.

`void zabs(verylong *a)`

`a` gets the value $|\mathtt{a}|$.

`void znegate(verylong *a)`

`a` gets the value $-\mathtt{a}$.

### (2.2) Basic arithmetic

`void zsadd(verylong a, long b, verylong *c)`
`void zadd(verylong a, verylong b, verylong *c)`

`c` gets the value $\mathtt{a} + \mathtt{b}$.

`void zsub(verylong a, verylong b, verylong *c)`

`c` gets the value $\mathtt{a} - \mathtt{b}$.

`void zsubpos(verylong a, verylong b, verylong *c)`

`c` gets the value $\mathtt{a} - \mathtt{b}$, only for $\mathtt{a} \geq \mathtt{b} \geq 0$.

`void z2mul(verylong a, verylong *b)`

`b` gets the value $2\mathtt{a}$.

`void zsmul(verylong a, long b, verylong *c)`

c gets the value $a * b$.

`void zmul(verylong a, verylong b, verylong *c)`

c gets the value $a * b$, output cannot be input (cf. (5.1)).

`void zmulin(verylong a, verylong *b)`

b gets the value $a * b$, output cannot be input a.

`void zmul_plain(verylong a, verylong b, verylong *c))`

c gets the value $a * b$, output cannot be input, uses ordinary multiplication (cf. (5.1)).

`void zsq(verylong a, verylong *b)`

b gets the value $a^2$, output cannot be input (cf. (5.1)).

`void zsqin(verylong *a)`

a gets the value $a^2$.

`void zsq_plain(verylong a, verylong *b)`

b gets the value $a^2$, output cannot be input, uses ordinary squaring (cf. (5.1)).

`long z2div(verylong a, verylong *b)`
`long z2mod(verylong a)`

b gets the value $\mathrm{sign}(a) * [|a|/2]$, returns $a \bmod 2 \in \{0, 1\}$.

`long zsdiv(verylong a, long b, verylong *c)`
`long zsmod(verylong a, long b)`

c gets the value $[a/b]$, returns $a \bmod b = a - c * b$ of the same sign as b if non-zero. Error message `division by zero in zsdiv` if $b = 0$; result undefined if error occurs.

`void zdiv(verylong a, verylong b, verylong *c, verylong *d)`
`void zmod(verylong a, verylong b, verylong *d)`

c gets the value $[a/b]$, d gets the value $a \bmod b = a - c * b$ of the same sign as b if $d \neq 0$. Error message `division by zero in zdiv` or `division by zero in zmod` if $b = 0$; result undefined if error occurs.

`zsexp(verylong a, long b, verylong *c)`
`zexp(verylong a, verylong b, verylong *c)`

c gets the value $a^b$, arguments cannot be the same. Error message `negative exponent in zexp` if $b < 0$ and $|a| \neq 1$; result undefined if error occurs. Do not use these functions if modular exponentiation can be used instead.

`long zsqrts(long a))`

Returns $|[\sqrt{a}]|$ if $a > 0$, returns 0 otherwise.

`long zsqrt(verylong a, verylong *b, verylong *c)`

b gets the value $|[\sqrt{a}]|$, c gets the value $a - b^2$, only if $a \geq 0$, output cannot be input. Returns 1 if a is a square (i.e., if $c = 0$), returns 0 otherwise. Error message `negative argument in zsqrt` if $a < 0$; result undefined if error occurs.

`long zroot(verylong a, long b, verylong *c)`

c gets the value $[\sqrt[b]{a}]$ (the positive root, if there is a choice), unless one of the error messages occurs: `dth root with d=0 in zroot`, or `dth root` with even d of negative number in zroot, or dth root with d<0 of zero in zroot; result undefined if error occurs. Returns 1 if a is a bth power, returns $-1$ if error, returns 0 otherwise.

`long zispower(verylong n, verylong *f)`

If $\mathtt{n} > 1$ and there is an integer $x$ satisfyinq $x^k = \mathtt{n}$ for some integer $k > 1$, returns the largest such $k$ and f gets $x$. Otherwise returns 0 and leaves $f$ unchanged.

### (2.3) Input/output

In the description of the input functions, an 'empty line' means a line that contains no characters except possibly spaces, and a 'leading space' is a space that is not preceded by a non-space character on the same line. A line can contain at most IN_LINE characters (cf. (3.3)).

`long zfread(FILE *f, verylong *a)`
`long zsread(char *f, verylong *a)`
`long zread(verylong *a)`

Read characters from f (which is supposed to be open for reading for zfread and which is equal to standard input for zread), while skipping leading spaces, empty lines, and everything on a line after a backslash (but see IN_LINE_BREAK in (3.3)), until a non-leading space is read, or until an end of line which is not preceded by a backslash is read on a non-empty line. The characters that are read (except for the possible spaces and backslashes) are interpreted as the decimal representation of an integer which is assigned to a. Note that **all** characters from f are interpreted (until a terminator). This is different from zstrtoz(bas), which stop reading as soon as a character not in the 'proper' range is encountered. The representation on f can be preceded by a '-' or a '_' to indicate a negative number. Return 1 if the representation on f consisted of the digits 0 through 9 only and nothing else went wrong, returns 0 otherwise.

`long zfwrite_c(FILE *f, verylong a, long b, char *c, char *d)`
`long zfwrite(FILE *f, verylong a)`
`long zswrite(char *f, verylong a)`
`long zwrite(verylong a)`

Write the decimal representation of a to f (which is supposed to be open for writing for zfwrite_c and zfwrite, and which is equal to standard output for zwrite)), using at most approximately b (in zfwrite_c) or OUT_LINE = 68 (in zfwrite and zwrite) characters per line (cf. (3.3)), and using a backslash (but see OUT_LINE_BREAK in (3.3)) to indicate continuation on the next line. In zfwrite_c the first line will be preceded by c, each consecutive line by d, where the lengths of c and d are included in the line-length count. Return the decimal length of |a| (where 0 has decimal length 1). Error messages allocation failure in zfwrite_c/zswrite or reallocation failure in zfwrite_c/zswrite indicate that a is too large to be printed; result undefined if error occurs. In zswrite it is assumed that f is large enough.

`long zfwriteln(FILE *f, verylong a)`
`long zwriteln(verylong a)`

Write the decimal representation of a followed by a new line to the file f (which is supposed to be open for writing for zfwriteln, and which is equal to standard output for zwriteln), using at most approximately OUT_LINE = 68 characters per line (cf. Section 3), and using a backslash to indicate continuation on the next line. Return values and error messages as in zfwrite.

`void zhfread(FILE *f, verylong *a)`
`void zhsread(char *f, verylong *a)`
`void zhread(verylong *a)`

Read characters from f (which is supposed to be open for reading for zhfread and which is equal to standard input for zhread), while skipping spaces, empty lines and everything on a line after a backslash (but see IN_LINE_BREAK in (3.3)), until an end of line which is not preceded by a backslash is read on a non-empty line. The characters that are read (except for the possible backslashes and spaces) are interpreted as the hexadecimal representation of an integer which is assigned to a. This representation on f can be preceded by a '-' or a '_' to indicate a negative number. Both lower case 'a' through 'f' and upper case 'A' through 'F' are accepted for the digits 10 through 15. All characters that are not equal to a space or a backslash, or that do not represent one of the digits 0 through 15, are interpreted as 0 (zero).

`void zhfwrite(FILE *f, verylong a)`
`void zhwrite(verylong a)`

Write the hexadecimal representation of a to the file f (which is supposed to be open for writing for zhfwrite, and which is equal to standard output for zhwrite) in blocks of at most 8 characters (but see HEX_BLOCK in (3.3)) separated by spaces (but see HEX_SEP_CHAR in (3.3)), with at most 7 blocks per line (but see HEX_BLOCKS_PER_LINE in (3.3)), and using a backslash (but see OUT_LINE_BREAK in (3.3)) to indicate continuation on the next line. The digits 10 through 15 are represented by 'A' through 'F', unless LIP is compiled with the '-DHEX_LOWER_CASE' flag, in which case 'a' through 'f' will be used.

`void zhfwriteln(FILE *f, verylong a)`
`void zhwriteln(verylong a)`

As zhfwrite and zhwrite, respectively, except that the hexadecimal representation is followed by a newline.

`long zbfread(FILE *f, verylong *a)`

Reads machine dependent representation of a from the binary file f (which is supposed to be open for reading), and which was written using zbfwrite. Returns 1 if successful, returns 0 otherwise. Unexpected results if a file is read that was created (using zbfwrite) on a machine with incompatible internal representation of data (though the most likely result in that case is the error message allocation failed in `zsetlength`).

`long zbfwrite(FILE *f, verylong a)`

Writes machine dependent representation of `a` to the binary file `f` (which is supposed to be open for writing). Returns 1 if successful, returns 0 otherwise. The representation on `f` can be read using `zbfread`.

`long zfread_b(FILE *f, verylong *a, verylong b, long c)`

Reads characters from `f` (which is supposed to be open for reading) while skipping leading spaces, empty lines, and everything on a line after a backslash, until an end of line which is not preceded by a backslash is read on a non-empty line, or if a non-leading space is read and $|\mathtt{b}| < 16$ and $\mathtt{c} = 0$. The characters that are read (except for the possible backslashes and spaces) are interpreted as the base $|\mathtt{b}|$ representation of an integer which is assigned to `a`. For $|\mathtt{b}| \leq 16$ and $\mathtt{c} = 0$ the digits 10 through 15 must be represented by '`a`' through '`f`' or '`A`' through '`F`'. For $|\mathtt{b}| > 16$ or $\mathtt{c} \neq 0$ non-leading spaces must be used to separate base $|\mathtt{b}|$ 'digits', which must be represented as decimal numbers. For $\mathtt{c} = 0$ only the sign of the first digit ('`-`' or '`_`') will be used, of the other digits the absolute value will be taken. For $\mathtt{c} \neq 0$ all digits may have signs. Returns 1 if end of line reached, returns 0 if line of `f` that is currently being read contains more data (which can only happen if $|\mathtt{b}| < 16$ and $\mathtt{c} = 0$). Error message `input base < 2 in zfread_b` if $|\mathtt{b}| \leq 1$; no characters read, no change to `a` if error occurs.

`long zfwrite_b(FILE *f, verylong a, verylong b, long c)`

Writes the base $|\mathtt{b}|$ representation of `a` to the file `f` (which is supposed to be open for writing) for $|\mathtt{b}| > 1$, using digits $\{0, 1, \ldots, |\mathtt{b}| - 1\}$ if $\mathtt{c} = 0$, but using digits centered around zero if $\mathtt{c} \neq 0$. For $\mathtt{c} \neq 0$ the digits will be separated by spaces; otherwise for $|\mathtt{b}| > 16$ the digits will be represented in decimal and separated by spaces, for $|\mathtt{b}| = 16$ in blocks as in `zhfwrite`, for $|\mathtt{b}| < 16$ without spaces, and for $|\mathtt{b}| \leq 16$ as in `zhfwrite` for the digits 10 through 15. Returns the number of digits used for the representation. Error messages `output base < 2 in zfwrite_b`, `allocation failure in zfwrite_b` (indicating that `a` is too large to print), `reallocation failure in zfwrite_b` (indicating the same), and `zfwrite_b   bug` (to be reported); result undefined if error occurs.

`long zfwriteln_b(FILE *f, verylong a, verylong b, long c)`

As `zfwrite_b`, but followed by a newline.

### (2.4) Bit manipulation

For a `verylong` or `long x` we denote by $(\mathtt{x}_i)_{i=0}^{l(x)}$ the *bits* of $|\mathtt{x}|$, i.e., $\mathtt{x}_i \in \{0, 1\}$ such that $|\mathtt{x}| = \sum_{i=0}^{l(x)} \mathtt{x}_i 2^i$, with $\mathtt{x}_{l(\mathtt{x})} = 1$ and where $l(\mathtt{x}) = -1$ if $\mathtt{x} = 0$; for $i < 0$ and $i > l(\mathtt{x})$ we define $\mathtt{x}_i = 0$.

`void zand(verylong a, verylong b, verylong *c)`

The $i$th bit of `c` gets the value $\mathtt{a}_i \cdot \mathtt{b}_i$, for $i = 0, 1, \ldots$.

`void zcat(verylong a, verylong b, verylong *c)`

The $i$th bit of $\mathtt{c}$ gets the value $\mathtt{b}_i$ for $i = 0, 1, \ldots, l(\mathtt{b})$ and the value $\mathtt{a}_{i-l(\mathtt{b})-1}$ for $i = l(\mathtt{b}) + 1, \ldots$.

`void znot(verylong a, verylong *b)`

If $\mathtt{a} = 0$ then $\mathtt{b}$ gets the value 1, otherwise the $i$th bit of $\mathtt{b}$ gets the value $1 - \mathtt{a}_i$, for $i = 0, 1, \ldots l(\mathtt{a})$ and $\mathtt{b}$ gets the same sign as $\mathtt{a}$ (unless $\mathtt{b} = 0$). Notice that $l(\mathtt{b}) < l(\mathtt{a})$ if $|\mathtt{a}| > 0$.

`void zor(verylong a, verylong b, verylong *c)`

The $i$th bit of $\mathtt{c}$ gets the value $\max(\mathtt{a}_i, \mathtt{b}_i)$, for $i = 0, 1, \ldots$.

`void zxor(verylong a, verylong b, verylong *c)`

The $i$th bit of $\mathtt{c}$ gets the value $(\mathtt{a}_i + \mathtt{b}_i) \bmod 2 \in \{0, 1\}$, for $i = 0, 1, \ldots$.

`long zodd(verylong a)`

Returns 1 if $\mathtt{a}_0 = 1$, returns 0 otherwise.

`void zlshift(verylong a, long b, verylong *c)`

$\mathtt{c}$ gets the value $2^{\mathtt{b}} \cdot \mathtt{a}$ if $\mathtt{b} \geq 0$ and $\mathtt{sign(a)} \cdot |\mathtt{a}|/2^{-\mathtt{b}}$ otherwise.

`void zrshift(verylong a, long b, verylong *c)`

$\mathtt{c}$ gets the value $\mathtt{sign(a)} \cdot |\mathtt{a}|/2^{\mathtt{b}}$ if $\mathtt{b} \geq 0$ and $2^{-\mathtt{b}} \cdot \mathtt{a}$ otherwise.

`long zmakeodd(verylong *a)`

Returns $-1$ is $\mathtt{a} = 0$, otherwise returns the largest $k$ such that $\mathtt{a}_i = 0$ for $i = 0, 1, \ldots, k$, and $\mathtt{a}$ gets the value $\mathtt{sign(a)} \cdot |\mathtt{a}|/2^k$.

`long zweights(long a)`

`long zweight(verylong a)`

Return $\sum_{i=0}^{l(\mathtt{a})} \mathtt{a}_i$.

`long zbit(verylong a, long b)`

Returns $\mathtt{a}_{|\mathtt{b}|}$.

`void zgetbits(verylong a, long b, long c, verylong *d)`

The $i$th bit of $\mathtt{d}$ gets the value $\mathtt{a}_{i+|\mathtt{c}|}$ for $i = 0, 1, \ldots, \mathtt{b} - 1$, and $\mathtt{d}$'s other bits are set to zero.

`long zsetbit(verylong *a, long b)`

Returns original value of $\mathtt{a}_{|\mathtt{b}|}$ and replaces $\mathtt{a}_{|\mathtt{b}|}$ by 1 (if is was zero). The sign of $\mathtt{a}$ will not be affected (unless it was zero).

`long zshighbits(verylong a, long b)`

Returns a long with $i$th bit equal to $\mathtt{a}_{l(\mathtt{a})-s+1+i}$, for $i = 0, 1, \ldots, s - 1$, where $s = \min(\mathtt{NBITS}, \mathtt{b})$.

`void zhighbits(verylong a, long b, verylong *c)`

The $i$th bit of $\mathtt{c}$ gets the value $\mathtt{a}_{l(\mathtt{a})-\mathtt{b}+1+i}$, for $i = 0, 1 \ldots, \mathtt{b} - 1$.

`long zslowbits(verylong a, long b)`

Returns a long with $i$th bit equal to $\mathtt{a}_i$, for $i = 0, 1, \ldots, \min(\mathtt{NBITS}, \mathtt{b}) - 1$.

`void zlowbits(verylong a, long b, verylong *c)`

The $i$th bit of $\mathtt{c}$ gets the value $\mathtt{a}_i$, for $i = 0, 1 \ldots, \mathtt{b} - 1$, the other bits become 0.

`long zreverses(long a)`

Returns a long with $i$th bit equal to $\mathtt{a}_{l(\mathtt{a})-i}$, for $i = 0, 1, \ldots, l(\mathtt{a})$.

`void zreverse(verylong a, verylong *b)`

The $i$th bit of $\mathtt{b}$ gets the value $\mathtt{a}_{l(\mathtt{a})-i}$, for $i = 0, 1, \ldots, l(\mathtt{a})$. Notice that $l(\mathtt{b})$ might become smaller than $l(\mathtt{a})$.

`long zswitchbit(verylong *a, long b)`

Returns original value of $a_{|b|}$ and replaces $a_{|b|}$ by $1 - a_{|b|}$. The sign of a will not be affected (unless it was or becomes zero).

(2.5) **Modular arithmetic**

In the modular arithmetic functions below it is assumed that the modulus n is positive and that the inputs (except for the exponents e in the modular exponentiations) are in the range $[0, n-1]$. If these conditions (which are not checked) are satisfied, the outputs will also be in the range $[0, n-1]$. If $n = 0$, most functions will give an error message `modulus zero in` "function-name"; the result is undefined if this error occurs.

`void zaddmod(verylong a, verylong b, verylong n, verylong *c)`

c gets the value $(a + b) \bmod n$.

`void zsubmod(verylong a, verylong b, verylong n, verylong *c)`

c gets the value $(a - b) \bmod n$.

`void zsmulmod(verylong a, long b, verylong n, verylong *c)`
`void zmulmod(verylong a, verylong b, verylong n, verylong *c)`

c gets the value $(a * b) \bmod n$.

`long zmulmods(long a, long b, long n)`

Returns the value $(a * b) \bmod n$.

`long zmulmod26(long a, long b, long n, double c)`

Returns the value $(a * b) \bmod n$ for c = `(double)b/(double)n`. Only for $0 \leq a, b, n < 2^{26}$. Often faster than `zmulmods`.

`void zmulinmod(verylong a, verylong *b, verylong n)`

b gets the value $(a * b) \bmod n$.

`void zsqmod(verylong a, verylong n, verylong *c)`

c gets the value $a^2 \bmod n$.

`void zsqrtmod(verylong a, verylong p, verylong *s)`

computes x for which $x^2 \equiv a \bmod p$ for prime p, and puts x in *s. If no such x exists or if p is not prime, then sets $*s = 0$.

`void zsqinmod(verylong *a, verylong n)`

a gets the value $a^2 \bmod n$.

`void zdivmod(verylong a, verylong b, verylong n, verylong *c)`

c gets the value $(a/b) \bmod n$. Error messages `division by zero in zdivmod` (indicating that $b = 0$, which is not allowed; result undefined if this happens), and `undefined quotient in zdivmod` (indicating that b after removal of common factors with a still has a factor in common with n). In the latter case a factor of n will be returned in c (if LIP is forced to continue).

`void zinvmod(verylong a, verylong n, verylong *b)`

b gets the value $(1/a) \bmod n$. Error messages `division by zero in zinvmod` (indicating that $a = 0$, which is not allowed; result undefined if this happens), and `undefined inverse in zinvmod` (indicating that a has a factor in common with n). In the latter case a factor of n will be returned in b (if LIP is forced to continue).

`void z2expmod(verylong e, verylong n, verylong *b)` b gets the value $2^{\mathtt{e}}$ mod n. Arguments cannot be the same. Error message `undefined quotient in z2expmod` occurs if e negative and $2^{-\mathtt{e}}$ mod n and n are not coprime, in which case a factor of n will be returned in b (if LIP is forced to continue).

```
void zsexpmod(verylong a, long e, verylong n, verylong *b)
void zexpmod(verylong a, verylong e, verylong n, verylong *b)
```

b gets the value $\mathtt{a}^{\mathtt{e}}$ mod n. Arguments (except a and b) cannot be the same. Error message `undefined quotient in zexpmod` occurs if e negative and $\mathtt{a}^{-\mathtt{e}}$ mod n and n are not coprime, in which case a factor of n will be returned in b (if LIP is forced to continue).

```
long zexpmods(long a, long e, long n)
```

Returns $\mathtt{a}^{|\mathtt{e}|}$ mod n.

```
long zdefault_m(long a)
```

Returns default window size for $m$-ary exponentiation (cf. [3: 4.6.3]) with an exponent consisting of a blocks of NBITS bits.

```
void zexpmod_m_ary(
      verylong a, verylong e, verylong n, verylong *b, long m)
```

b gets the value $\mathtt{a}^{\mathtt{e}}$ mod n, computed using the $m$-ary method with window size m (unless $\mathtt{m} \leq 1$, in which case the default window size will be used, or $\mathtt{m} \geq \mathtt{NBITS}$, in which case $\mathtt{NBITS} - 1$ will be used). This is faster than `zexpmod` for large e. Arguments (except a and b) cannot be the same. Error message `undefined quotient in zexpmod_m_ary` occurs if e negative and $\mathtt{a}^{-\mathtt{e}}$ mod n and n are not coprime, in which case a factor of n will be returned in b (if LIP is forced to continue).

```
void zexpmod_doub1(verylong a1, verylong e1,
           verylong a2, verylong e2, verylong n, verylong *b)
void zexpmod_doub2(verylong a1, verylong e1,
           verylong a2, verylong e2, verylong n, verylong *b)
void zexpmod_doub3(verylong a1, verylong e1,
           verylong a2, verylong e2, verylong n, verylong *b)
void zexpmod_doub(verylong a1, verylong e1,
           verylong a2, verylong e2, verylong n, verylong *b)
```

b gets the value $(\mathtt{a1}^{\mathtt{e1}} \cdot \mathtt{a2}^{\mathtt{e2}})$ mod $n$, using, in `zexpmod_doubi`, windows of size i, sliding if $\mathtt{i} > 1$, and an appropriate table of products of powers of a1 and a2 (cf. [8]). Depending on $\max\{\mathtt{e1}, \mathtt{e2}\}$, `zexpmod_doub` uses one of the `zexpmod_doubi`. Error message `negative exponent in zexpmod_doub(i)` if e1 or e2 negative; result undefined if error occurs.

### (2.6) **Montgomery arithmetic**

Arithmetic modulo some fixed odd modulus n can be done somewhat faster than ordinary modular multiplication by using Montgomery arithmetic on the Montgomery representation of the operands involved (cf. [6]). Conversion to and from the Montgomery representation take one Montgomery multiplication each per operand, so conversion should only be done before and after a (lengthy) modular computation.

To use Montgomery arithmetic, first initialize the odd modulus (by calling zmstart(n)), and next convert all operands with the exception of exponents to their Montgomery representation (using ztom). After that, apply the Montgomery addition, subtraction, multiplication, squaring, division, inversion, and exponentiation functions, all of whose names are of the form zmontabc, to the converted operands, just as the ordinary modular functions zabcmod would be applied to the non-converted ones. Finally, after completion of the computation convert the results back from their Montgomery representation to the regular representation (using zmtoz).

Once it is clear how this works, it might be worthwhile to use mixed Montgomery and ordinary arithmetic for multiplications involving small constants (using zsmontmul); see the source text of zmcomposite for an example of this 'mixed' arithmetic.

LIP supports only one Montgomery modulus at a time. There is no mechanism to prevent nonsensical usage of 'old' Montgomery variables with a 'new' Montgomery modulus, or even to apply Montgomery functions to non-Montgomery variables. Some LIP-functions use Montgomery arithmetic and initialize their own Montgomery modulus; unless the user has explicitly given permission to overwrite the user's Montgomery modulus (using zmfree), the user's Montgomery modulus will be restored. The latter might be costly if it is repeated very often, so it is always a good idea to zmfree a Montgomery modulus if it is no longer needed.

If no Montgomery modulus has been initialized, most functions below will give an error message undefined Montgomery modulus in "function-name"; the result is undefined if this error occurs.

void zmstart(verylong n)

Initializes Montgomery arithmetic for the modulus n. Error message even or negative modulus in zmstart if n is even or negative, which is not allowed; result undefined if error occurs.

void zmfree(void)

A call to zmfree allows the internal arithmetic of other functions to replace the Montgomery modulus without restoring a user installed Montgomery modulus.

void ztom(verylong a, verylong *ma)

ma gets the Montgomery representation of the regular integer a.

void zmtoz(verylong ma, verylong *a)

a gets the regular representation of the Montgomery value ma.

void zmontadd(verylong ma, verylong mb, verylong *mc)

mc gets the Montgomery sum of the Montgomery values ma and mb.

void zmontsub(verylong ma, verylong mb, verylong *mc)

mc gets the Montgomery difference of the Montgomery values ma and mb.

void zsmontmul(verylong ma, long d, verylong *mc)

mc gets the Montgomery product of the Montgomery value ma and the regular long d. Usually this is faster than zmontmul(ma,md,&mc), where md is the Montgomery representation of d (and where md can for instance be obtained using `zintoz(d,&md)` followed by `ztom(md,&md)`).

```
void zmontmul(verylong ma, verylong mb, verylong *mc)
```

`mc` gets the Montgomery product of the Montgomery values `ma` and `mb`.

```
void zmontsq(verylong ma, verylong *mb)
```

`mb` gets the Montgomery square of the Montgomery value `ma`.

```
void zmontdiv(verylong ma, verylong mb, verylong *mc)
```

`mc` gets the Montgomery quotient of the Montgomery values `ma` and `mb`. Error messages `division by zero in zmontdiv` (indicating that $\mathtt{mb} = 0$, which is not allowed; result undefined if this happens), and `undefined quotient in zmontdiv` (indicating that `mb` after removal of common factors with `ma` still has a factor in common with the Montgomery modulus `n`). In the latter case a factor of `n` will be returned in `mc` (if LIP is forced to continue).

```
void zmontinv(verylong ma, verylong *mb)
```

`mb` gets the Montgomery inverse of the Montgomery value `ma`. Error messages `division by zero in zmontinv` (indicating that $\mathtt{ma} = 0$, which is not allowed; result undefined if this happens), and `undefined inverse in zmontinv` (indicating that `ma` has a factor in common with the Montgomery modulus `n`). In the latter case a factor of `n` will be returned in `mb` (if LIP is forced to continue).

```
void zmontexp(verylong ma, verylong e, verylong *mb)
```

`mb` gets the value of the Montgomery exponentiation $(\mathtt{ma})^{\mathtt{e}}$, i.e., the Montgomery representation of $\mathtt{a}^{\mathtt{e}} \bmod \mathtt{n}$, where `ma` is the Montgomery representation of `a`. The argument `e` cannot be the same as `ma` or `mb`. Error message `undefined quotient in zmontexp` occurs if `e` is negative and an attempt is made to compute the inverse modulo `n` of some number that is not coprime with `n`. In the latter case a factor of `n` will be returned in `mb` (if LIP is forced to continue).

```
void zmontexp_m_ary(verylong ma, verylong e, verylong *mb, long m)
```

`mb` gets the value of the Montgomery exponentiation $(\mathtt{ma})^{\mathtt{e}}$, i.e., the Montgomery representation of $\mathtt{a}^{\mathtt{e}} \bmod \mathtt{n}$, where `ma` is the Montgomery representation of `a`, computed using the $m$-ary method (cf. [3: 4.6.3]) with window size `m` (unless $\mathtt{m} \leq 1$, in which case the default window size will be used (cf. `zdefault` in (2.5)), or $\mathtt{m} \geq \mathtt{NBITS}$, in which case $\mathtt{NBITS} - 1$ will be used). This is faster than `zmontexp` for large `e`. The argument `e` cannot be the same as `ma` or `mb`. Error message `undefined quotient in zmontexp_m_ary` occurs if `e` is negative and an attempt is made to compute the inverse modulo `n` of some number that is not coprime with `n`, in which case a factor of `n` will be returned in `mb` (if LIP is forced to continue).

```
void zmontexp_doub1(verylong ma1, verylong e1,
                             verylong ma2, verylong e2, verylong *mb)
void zmontexp_doub2(verylong ma1, verylong e1,
                             verylong ma2, verylong e2, verylong *mb)
void zmontexp_doub3(verylong ma1, verylong e1,
                              verylong ma2, verylong e2, verylong *mb)
void zmontexp_doub(verylong ma1, verylong e1,
                              verylong ma2, verylong e2, verylong *mb)
```

mb gets the value of the Montgomery product of the two Montgomery exponentiations $(\mathtt{ma1})^{\mathtt{e1}}$ and $(\mathtt{ma2})^{\mathtt{e2}}$, using, in zmontexp_doubi, windows of size i, sliding if $\mathtt{i} > 1$, and an appropriate table of Montgomery products of Montgomery powers of ma1 and ma2 (cf. [8]). Depending on $\max\{\mathtt{e1}, \mathtt{e2}\}$, zmontexp_doub uses one of the zmontexp_doubi. Error message negative exponent in zmontexp_doub(i) if e1 or e2 negative; result undefined if error occurs.

### (2.7) Euclidean algorithms

```
void zchirem(verylong a, verylong xa,
                                verylong b, verylong xb, verylong *c)
```

c is computed such that $\mathtt{c} \equiv \mathtt{xa} \bmod \mathtt{a}$ and $\mathtt{c} \equiv \mathtt{xb} \bmod \mathtt{b}$, for positive a and b, which must be coprime if $\mathtt{xa} \neq \mathtt{xb}$. Error messages zero or negative argument(s) in zchirem (a or b zero or negative), same moduli with different remainders in zchirem ($\mathtt{a} = \mathtt{b}$ but $\mathtt{xa} \neq \mathtt{xb}$), or moduli not coprime in zchirem (a and b not coprime); result undefined if error occurs.

```
void zgcd(verylong a, verylong b, verylong *c)
```

c gets the greatest common divisor of a and b, computed using the binary gcd algorithm (i.e., no divisions).

```
void zgcdeucl(verylong a, verylong b, verylong *c)
```

c gets the greatest common divisor of a and b, computed using the ordinary Euclidean algorithm (which is usually slower than the binary method, but might be faster in special cases).

```
void zexteucl(verylong a, verylong *xa,
                               verylong b, verylong *xb, verylong *c)
```

c gets the greatest common divisor of a and b, and xa and xb get values such that $\mathtt{a} \cdot \mathtt{xa} + \mathtt{b} \cdot \mathtt{xb} = \mathtt{c}$, computed using Lehmer's method (cf. [3: 4.5.2]). Arguments cannot be the same. Error message non-zero remainder in zexteucl   BUG (to be reported); results undefined if error occurs.

```
long zinvs(long a, long b)
```

Returns an integer $x$ such that $x \cdot |\mathsf{a}| \equiv \gcd(\mathtt{a}, \mathtt{b}) \bmod \mathtt{b}$, using ordinary extended Euclidean algorithm.

```
long zinvodds(long a, long b)
```

Returns an integer $x$ such that $x \cdot |\mathsf{a}| \equiv 1 \bmod \mathtt{b}$, for coprime $\mathtt{a} > 0$ and odd $\mathtt{b} \geq 3$, using binary method. This is usually much faster than the method used by zinvs. Returns 0 is b is even. Error message arguments not coprime in zinvodds; returns zero if error occurs.

```
long zinv(verylong a, verylong b, verylong *c)
```

Returns zero and c gets the value $\mathtt{a}^{-1} \bmod \mathtt{b}$ if a and b are coprime, computed using Lehmer's method; returns 1 and c gets the greatest common divisor of `a` and `b` if `a` and `b` are not coprime. Arguments cannot be the same. Only for `a` $> 0$ and `b` $> 0$. Error message `zero or negative argument(s) in zinv` if `a` $\leq 0$ or `b` $\leq 0$; result undefined if error occurs.

`long zjacobi(verylong a, verylong b)`

Returns, only for `b` $> 0$, the Jacobi symbol of `a` and `b`: zero if `a` and `b` are not coprime, 1 if `a` and `b` are coprime and there exists an integer $x$ such that $x^2 \equiv$ `a` mod `b`, and $-1$ otherwise. Error message `non-positive second argument in zjacobi`; result undefined if error occurs.

`long zjacobis(long a, long b)`

As `zjacobi`, except that `b` is required to be odd, but may be negative. Error message `even second argument in zjacobis`; result undefined if error occurs.

(2.8) **Random number generation**

`void zrstart(verylong s)`

`void zrstarts(long s)`

Initialize or reset the seed of the built-in pseudo-random number generator as `s`.

`void zrseed(verylong *a)`

`a` is set to the current value of the seed of the built-in pseudo-random number generator.

`long zrandom(long bnd)`

If `bnd` $> 0$, the seed $s$ of the built-in pseudo-random number generator will be replaced by $(s \cdot g) \bmod p \in \{0, 1, \ldots, p-1\}$ (where $p = 2^{107} - 1$ is prime and $g = 3^{121}$ generates $(\mathbf{Z}/p\mathbf{Z})^*$) until the resulting $s$ is $< p - (p \bmod$ `bnd`$)$ (to avoid inhomogeneity), and $s \bmod$ `bnd` $\in \{0, 1, \ldots,$ `bnd`$-1\}$ will be returned; otherwise, if `bnd` $\leq 0$, the seed will not be changed and zero will be returned. Notice that `zrandom` will return only zero if the seed is set to zero (using `zrstart` or `zrstarts`). The default initial value of $s$ is 7157891. This function cannot be recommended for generation of 'cryptographically strong' pseudo-random numbers.

`void zrandomb(verylong bnd, verylong *a)`

`a` is set to a more or less randomly (but not homogeneously) selected integer in the interval $[0,$ `bnd` $- 1]$, unless the interval is empty in which case `a` gets the value zero. To generate the random values necessary to construct `a`, `zrandomb` uses the function `zrandom`, and as `zrandom` it will always set `a` to zero if the seed is set to zero (using `zrstart` or `zrstarts`). If cryptographic security is important, `zrandomb` should not be used, and the user should provide his/her own version of `zrandomb`.

```
void zrandoml(long length, verylong *a,
                                   void(*u)(verylong,*verylong))
```

`a` is set to a randomly selected integer of precisely |`length`| bits, with the same sign as `length`. The function `u`, which is used to generate the random value that is needed to construct `a`, should give `c` a more or less random value in the range $[0,$ `b` $- 1]$ upon call `u(b,&c)`, for a `verylong`

b. If cryptographic security is important, zrandomb should not be used for u.

### (2.9) Primality testing and factoring

long zcomposite(verylong *a, long t, long first)

If $a \leq 2^{NBITS/2}$ or if a even, zcomposite returns 0 if a is prime and 1 otherwise. Otherwise, if a is odd and $> 2^{NBITS/2}$, zcomposite returns 1 if it could prove that a is composite (using at most $|t|$ probabilistic compositeness tests) and that, if $t < 0$, it is not a prime power; it returns 0 if $|t|$ probabilistic compositeness tests were unable to prove that a is composite. The latter happens only with probability $(1/4)^{|t|}$ for composite a, so that if $|t|$ is sufficiently large one may assume that a is prime if 0 is returned. Furthermore, if $t < 0$, and still in the case that a is odd and $> 2^{NBITS/2}$, zcomposite returns $-1$ if a factor of a has been detected (in which case the factor will be returned in a).

zcomposite uses the method from [3: 4.5.4, Alg. P], with the 'prime power' extension from [4: 2.5], for at most $|t|$ bases. If $first = 0$ all these bases will be selected (using zrandom) in the range $[3, \min(a, RADIX) - 1]$; otherwise $|first|$ will be used as the first base, and the other $|t| - 1$ bases will be selected (using zrandom) in the range $[3, \min(a, RADIX) - 1]$. This makes it possible to make use of the slightly faster special arithmetic that is used if $|first| = 2$.

For large values of t the function zmcomposite is more efficient than zcomposite (because zmcomposite uses mixed ordinary/Montgomery arithmetic). For numbers that might have small factors zprobprime is on average faster than zcomposite (because zprobprime does some trial divisions before calling zcomposite and zmcomposite).

long zmcomposite(verylong a, long t)

If $a \leq 2^{NBITS/2}$ or if a even, zmcomposite returns 0 if a is prime and 1 otherwise. Otherwise, if a is odd and $> 2^{NBITS/2}$, zmcomposite returns 1 if it could prove that a is composite (using at most $|t|$ probabilistic compositeness tests); it returns 0 if $|t|$ probabilistic compositeness tests were unable to prove that a is composite. The latter happens only with probability $(1/4)^{|t|}$ for composite a, so that if $|t|$ is sufficiently large one may assume that a is prime if 0 is returned.

zmcomposite uses the method from [3: 4.5.4, Alg. P] for at most $|t|$ bases that are selected (using zrandom) in the range $[3, \min(a, RADIX) - 1]$. It uses mixed ordinary/Montgomery arithmetic and is thus slightly faster than zcomposite.

long zprime(verylong a, long t, long first)

If $a \leq 2^{NBITS/2}$ or if a even, zprime returns 1 if a is prime and 0 otherwise. Otherwise, if a is odd and $> 2^{NBITS/2}$, zprime returns 0 if it could prove that a is composite (using at most $|t|$ probabilistic compositeness tests); it returns 1 if $|t|$ probabilistic compositeness tests were unable to prove that a is composite. The latter happens only with probability $(1/4)^{|\mathtt{t}|}$ for composite `a`, so that if $|\mathtt{t}|$ is sufficiently large one may assume that `a` is prime if 1 is returned.

`zprime` uses the method from [3: 4.5.4, Alg. P] for at most $|\mathtt{t}|$ bases. If $\mathtt{first} = 0$ all these bases will be selected (using `zrandom`) in the range $[3, \min(\mathtt{a}, \mathtt{RADIX}) - 1]$; otherwise $|\mathtt{first}|$ will be used as the first base, and the other $|\mathtt{t}| - 1$ bases will be selected (using `zrandom`) in the range $[3, \min(\mathtt{a}, \mathtt{RADIX}) - 1]$. This makes it possible to make use of the slightly faster special arithmetic that is used if $|\mathtt{first}| = 2$.

For large values of `t` it is more efficient to use `zmcomposite` than `zprime`.

`long zprobprime(verylong a, long t)`

Returns 0 if `a` proved to be composite, returns 1 if compositeness could not be proved using $1+|\mathtt{t}|$ probabilistic compositeness tests. `zprobprime` first attempts to find a small factor using trial division (using `ztridiv`). If no small factor could be found, it calls `zcomposite(&a,1,2)`, and if that did not prove compositeness it calls `zmcomposite(a,t)`.

On average `zprobprime` is faster than any of the functions above, if no additional information about the numbers to be tested is known. If a number is suspected to be prime, however, and trial division is unlikely to succeed, then `zcomposite`, `zmcomposite`, or `zprime` should be used.

`long ztridiv(verylong a, verylong *cofac, long low, long high)`

Returns 0 if $\mathtt{low} < 0$ or $\mathtt{low} > \mathtt{high}$ or $\mathtt{high} \geq \mathtt{RADIX}$. Otherwise, `ztridiv` returns the smallest prime factor of `a` in $[\mathtt{low}, \mathtt{high}]$, if any, and sets `cofac` equal to the cofactor; returns smallest prime $> \mathtt{high}$ if `a` has no prime factor in $[\mathtt{low}, \mathtt{high}]$. The function `ztridiv` is only intended for fairly large `a`: it does not avoid trial division with primes $> \sqrt{|\mathtt{a}|}$.

```
long zfecm(verylong n, verylong *fac, long s, long *nb,
       long *bnd, long percent, long t, long info, FILE *fp)
```

Attempts to find a non-trivial factor of `n` using the elliptic curve integer factorization method (ecm), using at most `*nb` curves with initial first phase bound `*bnd` (which grows by `percent` percent per curve) (cf. (5.3)). Returns $-1$ if `n` could not be proved to be composite using $|\mathtt{t}|$ probabilistic compositeness tests, returns 0 if no factor found or if $\mathtt{n} \leq 1$ and returns 1 if a factor has been found (in which case the factor will be put in `fac`). The value of `*nb` is set to the number of curves used and the value of `*bnd` is updated to the last value of `*bnd` used (same as the initial value when $\mathtt{percent} = 0$).

If $\mathtt{s} \neq 0$, the built-in pseudo-random generator `zrandom` will be initialized with a call to `zrstarts` with argument $2\mathtt{s}+2$ for $\mathtt{s} > 0$ and $-2\mathtt{s}+1$ for $\mathtt{s} < 0$; if $\mathtt{s} = 0$ no call to `zrstarts` will be made. The curves will then be randomly selected using `zrandom`. In this way a different non-zero `s` for the same `n` should lead to a different sequence of curves. This makes it possible to run `zfecm` on multiple processors on the same `n`, but using different curves. If $|\mathtt{info}| \geq 2$ various informative messages will be printed on `*fp`, per phase and per curve; for $|\texttt{info}| = 1$ they will only be given per curve, and for $\texttt{info} = 0$ or $\texttt{fp} = \texttt{NULL}$, `zfecm` runs silently. Error messages `impossible return value of zcomposite in zfecm BUG` or two numbers followed by `this is a wrong factor, found in zfecm BUG`; both should be reported.

We refer to (5.3) for some hints how `nb`, `bnd`, and `percent` might be chosen, and how `zfecm` can be customized. The author would appreciate to hear about any factor of 38 or more decimal digits found by `zfecm`.

For applications that do not use `zfecm` and for which the small amount of memory needed for some global arrays used by `zfecm` might be a problem, LIP can be compiled with the '`-DNO_ECM`' flag. If both `zfecm` and the '`-DNO_ECM`' flag are used, the first call to `zfecm` will result in the message `Compile without the -DNO_ECM flag to get ecm code` on standard error.

```
long zecm(verylong n, verylong *fac, long s, long nb,
                      long bnd, long percent, long t, long info)
```

As `zfecm`, but `nb` and `bnd` will not be changed, and `fp` is standard output.

```
long zpollardrho(verylong n, verylong *res, verylong *cof, long t)
```

Attempts to factor `n` with Brent's version of Pollard's rho using at most `t` iterations of the main loop. If `t` is zero, then runs till factor is found. The two cofactors are put in `res` and `cof`. Returns positive integer if factor is found, 0 otherwise.

```
long zsquf(verylong n, verylong *f1, verylong *f2)
```

Attempts to factor $\texttt{n} < \texttt{RADIX}^2$ using Shanks's 'squfof.' Returns positive integer if successful, and puts factors in `f1` and `f2`. Otherwise returns 0. If $\texttt{n} \geq \texttt{RADIX}^2$ then returns 0 without attempting to factor `n`.

### (2.10) **Prime generation**

```
long zpnext(void)
```

Returns the next small prime, starting at 2, unless `zpstart` has been called, in which case the first subsequent call to `zpnext` returns 3. After returning the last prime it can generate (which is approximately $(2 \cdot \texttt{PRIM\_BND} + 1)^2$) the next call to `zpnext` wraps around and returns 2. For the default setting $\texttt{PRIM\_BND} = 2^{\texttt{NBITS}/2-1}$ and `NBITS` equal to 30 (but see (3.3)) the largest prime that can be generated by `zpnext` is 1073840111.

```
long zpnextb(long bnd)
```

Returns 0 if $\texttt{bnd} \geq (2 \cdot \texttt{PRIM\_BND} + 1)^2 - 4 \cdot \texttt{NBITS}$. Otherwise, the smallest prime $\geq$ `bnd` will be returned, and the small prime generator will be repositioned in such a way that `zpnext` returns the small primes from there on.

```
void zpstart(void)
```

Repositions the small prime generator so that the next call to `zpnext` will return 3.

```
void zpstart2(void)
```

Repositions the small prime generator so that the next call to `zpnext` will return 2.

```
long zp(void)
```

Returns zero if no call to `zpnext` or `zpnextb` has been made yet, or if `zpnext` or `zpnextb` has not yet been called after the most recent call to `zpstart` or `zpstart2`. Otherwise, `zp` returns the prime most recently returned by `zpnext` or `zpnextb`.

```
long zrandomprime(long length, long t, verylong *p,
                                        void(*u)(verylong,verylong*))
```

`p` will be set to zero if $|\mathtt{length}| < 2$. Otherwise, `p` will be set to a random probable prime of precisely $|\mathtt{length}|$ bits, with `p` equal to 3 modulo 4 if $\mathtt{length} < 0$. The function `u` is as in `zrandoml`, and will be used to generate the random values. At most $1 + |\mathtt{t}|$ probabilistic compositeness tests will be carried out per candidate `p`-value (cf. (2.9)). Returns 1 if successful, returns 0 otherwise.

Do not use `zrandomb` for `u` if cryptographically strong primes are needed. `zrandomprime` works by picking an odd number of the right size (and residue class mod 4, if $\mathtt{length} < 0$), and keeps adding 2 (or 4) to it until it passes $1 + |\mathtt{t}|$ probabilistic compositeness tests, or until it is too large in which case `zrandomprime` starts all over again, thus following the suggestion from [7: Appendix].

```
long zrandomqprime(long lp, long lq, long t, verylong *p,
     verylong *q, verylong *quot, void(*u)(verylong,verylong*))
```

If $\mathtt{lq} < 0$, a positive number is expected in `q`; if $\mathtt{lq} \geq 0$, `zrandomqprime` attempts to set `q` to a randomly selected probable prime of precisely `lq` bits. Furthermore, `zrandomqprime` attempts to set `p` to a randomly selected probable prime of precisely `lp` bits, such that `q` divides $\mathtt{p}-1$, and sets `quot` to the resulting $(\mathtt{p}-1)/\mathtt{q}$. The function `u` is as in `zrandoml`, and will be used to generate the random values. At most $1+|\mathtt{t}|$ probabilistic compositeness tests will be carried out per candidate `p` or `q`-value (cf. (2.9)). Returns 1 if successful, returns 0 otherwise.

Do not use `zrandomb` for `u` if cryptographically strong primes are needed. `zrandomqprime` uses `zrandomprime` to find a `q`, if necessary, of the right size that passes $1 + |\mathtt{t}|$ probabilistic compositeness tests. Once `q` has been determined, at most $2 \cdot \mathtt{lp}$ random `p`'s with `q` dividing $\mathtt{p} - 1$ are selected, until `p` passes $1 + |\mathtt{t}|$ probabilistic compositeness tests, thus following the suggestion from [7: Appendix]. This implies that `zrandomqprime` can only be expected to work if `lp` is substantially larger than `lq` if $\mathtt{lq} \geq 0$, or than $\log_2(\mathtt{q})$ if $\mathtt{lq} < 0$. Error message `wrong q in zrandomqprime` if $\mathtt{lq} < 0$ and $\mathtt{q} \leq 0$; result undefined if error occurs.

```
long zrandomfprime(long lq, long t, verylong fac,
      verylong *p, verylong *q, void(*u)(verylong,verylong*))
```

Repeatedly uses `zrandoml` (and the function `u` to generate the random values) to set `q` to a probable prime of `lq` bits that passes $1 + |\mathtt{t}|$ probabilistic compositeness tests (cf (2.9)), until $\mathtt{p} = \mathtt{fac}\cdot\mathtt{q}+1$ is also probably prime (passing $1+|\mathtt{t}|$ probabilistic compositeness tests). Returns 1 if successful, returns 0 otherwise (for instance, if $\mathtt{fac} < 2$ or if `fac` is odd). Do not use `zrandomb` for `u` if cryptographically strong primes are needed.

```
long zrandomgprime(long lq, long t, long small, verylong *p,
       verylong *q, verylong *g, void(*u)(verylong,verylong*))
```

Calls `zrandomfprime(lq,t,factor,p,q,u)` with $\mathtt{factor} = 2$ to set `q` and $\mathtt{p} = 2\cdot\mathtt{q}+1$ to two probable primes that pass $1+|\mathtt{t}|$ probabilistic compositeness tests each (cf (2.9)), such that `q` has binary length equal to `lq`, and sets `g` to a generator of the multiplicative group modulo `p`. If $\mathtt{small} = 0$, the generator `g` will be randomly selected (using `u`), otherwise the smallest positive `g` will be selected. Returns 1 if successful, returns 0 otherwise. Do not use `zrandomb` for `u` if cryptographically strong primes are needed.

(2.11) **NIST digital signature algorithm**

These functions are not included in the publicly available version of LIP.

```
long zdsa_make_key(long lp, long lq, long t, verylong p,
                verylong q, verylong g, verylong x, verylong y,
                                 void(*u)(verylong,verylong*))
```

Attempts to find (using `zrandomqprime` with `u` as random generator and $|\mathtt{t}|$ as number of compositeness tests) primes `p` and `q` of `lp` and `lq` bits, respectively, such that `q` divides $\mathtt{p}-1$, and an element `g` of order `q` modulo `p` such that $0 < \mathtt{g} < \mathtt{p}$ (using `u`). Furthermore, uses `u` to find an `x` in $(0,\mathtt{q})$ and sets `y` equal to $\mathtt{g}^{\mathtt{x}} \bmod \mathtt{p}$. Returns 1 is successful, returns 0 otherwise. Notice that `p`, `q`, `g`, `x`, and `y` (upon successful return) satisfy the requirements of the public and private keys for the NIST digital signature algorithm as described in [7], if $|\mathtt{t}|$ is set to at least 50.

Prints a warning message `WARNING --- using cryptographically weak random generator, private key x might be compromised` on standard error if `zrandomb` is used as the generator `u` of supposedly random integers, because `x` must be generated by a cryptographically secure random generator.

```
long zdsa_check_public_key(verylong p, verylong q, verylong g,
                                                       long t)
```

Returns 0 if either `p` or `q` was proved to be composite (using at most $|\mathtt{t}|$ probabilistic compositeness tests), or if `q` does not divide $\mathtt{p}-1$, or if `g` is not an element of order `q` modulo `p`. Returns 1 otherwise. If 0 is returned the requirements for the public key for the NIST digital signature algorithm are not satisfied.

```
long zdsa_check_private_key(verylong p, verylong q, verylong g,
                               verylong x, verylong y, long t)
```

Returns 0 if either `p` or `q` was proved to be composite (using at most $|\mathtt{t}|$ probabilistic compositeness tests), or if `q` does not divide $\mathtt{p}-1$, or if `g` is not an element of order `q` modulo `p`, or if `y` does not equal $\mathtt{g}^{\mathtt{x}} \bmod \mathtt{p}$. Returns 1 otherwise. If 0 is returned the requirements for the public/private key pair for the NIST digital signature algorithm are not satisfied.

```
long zdsa_sign(verylong p, verylong q, verylong g, verylong m,
                verylong x, verylong *k, verylong *r, verylong *s,
                                    void(*u)(verylong,verylong*))
```

Computes the signature `r`, `s` of the message digest `m` modulo `q`, using the public/private key pair `p`, `q`, `g`, `x`, and the session key `k`, following the specifications of the NIST digital signature algorithm from [7]. If `k` is in the interval $[1, q-1]$ upon call, that value will be used; otherwise, if the value of `k` is not in $[1, q-1]$, `zdsa_sign` uses `u` as random generator to find a session key `k` in $[1, q-1]$; the value used will be returned in `k`. Returns 1 upon successful computation of the signature.

Prints a warning message `WARNING --- using cryptographically weak random generator, random k might be compromised` on standard error if `k` was generated by `zdsa_sign` using `zrandomb` as the generator `u` of supposedly random integers, because `k` must be generated by a cryptographically secure random generator.

Error message `common factor of k and q in zdsa_sign` indicates that the public key is incorrect; returns 0 if error occurs (and if LIP is forced to continue), and the common factor will be printed on standard error.

On a sparc 10 model 50 a signature for a 512-bit `p` and a 160-bit `q` takes on average about 0.15 seconds, which can be reduced to 0.045 seconds by using the '`-DSINGLE_MUL`' flag.

```
long zdsa_verify(verylong p, verylong q, verylong g, verylong m,
                              verylong y, verylong r, verylong s)
```

Returns 1 if `r`, `s` is a valid signature (according to the specifications of the NIST digital signature algorithm from [7]) for the message digest `m` modulo `q` with public key `p`, `q`, `g`, `y`. Returns 0 if the signature is not valid.

Error message `common factor of s and q in zdsa_verify` indicates that the public key is incorrect; returns 0 if error occurs (and if LIP is forced to continue), and the common factor will be printed on standard error.

On a sparc 10 model 50 a verification for a 512-bit `p` and a 160-bit `q` takes on average about 0.2 seconds, which can be reduced to 0.06 seconds by using the '`-DSINGLE_MUL`' flag.

### (2.12) **Timing functions**

```
double gettime(void)
```

Returns the user time plus the system time (in seconds) spent on the current process till the moment of call. Subtract the value returned at the first call from the value returned at the second call to find the time spent on the computation between the two calls.

```
double getutime(void)
```

As `gettime`, but returns user time only.

```
double getstime(void)
```

As gettime, but returns system time only.

void starttime(void)

Starts, or restarts, a timer; to be used with printtime.

void printtime(FILE *f)

Prints the time spent since the last call to starttime to file f (using format "%8.5lf sec."), followed by a newline, and flushes f.

(2.13) **Allocation**

void zsetlength(verylong *a, long length, char *s)

Allocates (or reallocates) enough space for a so that it can store an integer of absolute value $<$ RADIX$^{\texttt{length}}$, unless length $<$ SIZE in which case SIZE will be used instead of length (cf. (3.3)). If LIP is compiled with the '-DPRT_REALLOC' flag each call to zsetlength, including the calls made by the LIP-functions themselves, will print the message x (re)allocating to y on standard error, where x is the contents of the string s, and y the value used for length. In this way the user can get some idea how often variables are (re)allocated, and to what size, to find the right initial value of SIZE for his/her application.

In normal use of LIP, the user does not have to make his/her own calls to zsetlength. If LIP is compiled with the '-DNO_ALLOCATE' flag, however, the LIP-functions assume that the user has used zsetlength to allocate space for all verylong operands: except for zsetlength all LIP-functions will assume that 'a == 0' is not true for all their verylong arguments, if the '-DNO_ALLOCATE' flag is used. But even if this flag is used, automatic reallocation will take place if it turns out that not enough space has been allocated for a variable. Error messages (re)allocation failed in zsetlength, indicating that LIP is out of space (the name of the function and the attempted allocation size will be printed as well), and negative size allocation in zsetlength which indicates a wrong call to zsetlength (and a bug in LIP if the user did not call zsetlength him/herself).

void zfree(verylong *a)

Frees the space allocated for a. Only useful for variables local to recursive functions or to functions that are used exclusively in the initial stages of a computation. For variables in non-recursive functions that are used throughout a computation it is more efficient to keep the space allocated for verylong variables allocated for later calls (by declaring them as static verylong).

## 3. Machine or user dependent constants and include files

The default settings of all machine or user dependent constants described in this section can be changed by changing lip.h or lip.c, or, for most of them, by compiling LIP with the appropriate flag(s): for instance, to give SIZE the value 2 instead of the default value 20, compile LIP with the '-DSIZE=2' flag.

(3.1) **Machine dependent constants**

The default settings of the machine dependent constants in LIP is intended for use on machines where chars have 8 bits, sizeof(long) equals 4, and sizeof(double) equals twice sizeof(long). For machines with sizeof(long) and sizeof(double) equal to 8 (such as alpha architectures), the -DALPHA or -DALPHA50 flags can be used for faster arithmetic and more efficient memory usage (see (5.2)).

The default settings are as follows:

```
#define CHARL      8
#define SIZEOFLONG 4
#define NBITS      30, or 26 if the '-DSINGLE_MUL' flag is used
```

CHARL should be set to the number of bits of a char, and SIZEOFLONG should be set to sizeof(long). These two values depend on the machine and/or the implementation of C on that machine. Given CHARL and SIZEOFLONG, the value for NBITS should be chosen in such a way that NBITS is a positive even integer strictly less than CHARL · SIZEOFLONG. By default NBITS is set to 30, the largest possible value for CHARL = 8 and SIZEOFLONG = 4. The choice NBITS = 26, however, makes it possible to use a different set of macros and to get, on many machines, somewhat faster multiplication of verylongs; usage of the '-DSINGLE_MUL' flag sets NBITS to 26 and activates these alternative macros, but simply using the '-DNBITS=26' flag keeps the default macros. Although the '-DSINGLE_MUL' flag often leads to faster multiplication, it also leads to slightly slower addition and slightly higher memory demands. Furthermore, NBITS = 26 is inconvenient for some (factoring) applications. Nevertheless, for many applications (like elliptic curve factoring, cf. (5.3), or DSA, cf. (2.11)) the '-DSINGLE_MUL' flag can be recommended.

If NBITS is chosen exceedingly small (i.e., if NBITS $\leq$ (CHARL · SIZEOFLONG)/2) then the macros should be changed to take advantage of that choice. Such a small choice for NBITS can however not be recommended, because it is more efficient to use either the largest possible value or the '-DSINGLE_MUL' flag.

(3.2) **Remark**. The code that is used with the '-DSINGLE_MUL' flag assumes that the order of the words in doubles is 'high-low' (sparcs). On architectures where that is incorrect and the order is 'low-high', LIP should be compiled with the '-DDOUBLES_LOW_HIGH' flag (decs). If LIP is compiled following the instructions in (1.2) the right order will be used, if the timer-program decides that using the '-DSINGLE_MUL' flag is optimal.

(3.3) **User dependent constants**

```
#define SIZE                 20
#define IN_LINE              2048
#define IN_LINE_BREAK        '\\'
#define OUT_LINE             68
#define OUT_LINE_BREAK       '\\'
#define HEX_BLOCK            8
#define HEX_BLOCKS_PER_LINE  7
#define HEX_SEP_CHAR         ' '
#define PRIM_BND             (1<<((NBITS>>1)-1))
```

`SIZE` is the default and minimum number of `long`s that will be allocated for a `verylong` variable (cf. (1.7)); it can be set to any integer such that $\mathtt{SIZE}\cdot\mathtt{NBITS} \geq \mathtt{CHARL}\cdot\mathtt{SIZEOFLONG}$. The best value depends on the application: a large value might require fewer reallocations, and a small value might use less memory. In applications where, for instance, large matrices with `verylong` entries modulo some `verylong m` are used, it is probably best to use the smallest possible `SIZE` with $\mathtt{SIZE}\cdot\mathtt{NBITS} \geq \log_2(\mathtt{m})$. To find out which value of `SIZE` might be good for a certain application, see the documentation of `zsetlength` in (2.13).

The input functions for `verylong`s accept at most `IN_LINE` characters per line. The default choice of 2048 should not be too restrictive, because longer lines can easily be split into smaller ones, as explained in (1.3) and (2.3). `OUT_LINE` is an approximate bound for the maximal number of characters per line in the various output functions for `verylong`s (cf. (2.3)). On unusually narrow screens 40 might look better than the default choice of 68, and on old lineprinters 130 might be a good choice.

`IN_LINE_BREAK` and `OUT_LINE_BREAK` are the backslashes that indicate continuation on the next line of `verylong`s in input or output, respectively. To change them the source code `lip.c` has to be changed; they cannot be changed using a flag. They can assume any character value, but it is obviously not a good idea to give them confusing values like `'='`, `'*'`, or `'2'`.

`HEX_BLOCK`, `HEX_BLOCKS_PER_LINE`, and `HEX_SEP_CHAR` determine the format of hexadecimal output: at most $\mathtt{HEX\_BLOCKS\_PER\_LINE} = 7$ blocks of at most $\mathtt{HEX\_BLOCK} = 8$ characters per line, separated by a space (`HEX_SEP_CHAR`). The values of `HEX_BLOCK` and `HEX_BLOCKS_PER_LINE` can be set during compilation; the source code has to be changed to change `HEX_SEP_CHAR` into any other character value.

As explained in (2.10), `PRIM_BND` determines the bound on the small primes that can be generated using the small prime generator `zpnext`. By default `PRIM_BND` is set to the value above, unless one of the '`-DSINGLE_MUL`' (cf. (3.1)), '`-DALPHA`', or '`-DALPHA50`' flags is used, in which case $\mathtt{PRIM\_BND} = 2^{14}$. The small prime generator allocates two arrays consisting of `PRIM_BND` `short ints`, so if `NBITS` is substantially larger than the default value 30 (as might be the case on machines where `sizeof(long)` equals 8), the default setting of `PRIM_BND` might require too much memory, and `PRIM_BND` should be given a smaller value.

### (3.4) **Include files**

It might depend on the operating system which include files are needed. For me the following include files suffice:

```
<stddef.h>
<stdio.h>
<math.h>
<malloc.h>
<sys/time.h>
<sys/resource.h>
```

It seems that on HP workstations one also needs

```
<syscall.h>
```

and furthermore that on more recent operating systems `<sys/time.h>` and `<sys/resource.h>` have to be replaced by `<sys/times.h>` and `<limits.h>`. It is also possible that the definitions of the timings functions (`gettime` etc.) have to be changed on some of those systems.

## 4. Macros

The performance of LIP may be improved by replacing a few macros by inline assembly functions. In this section we list these macros, describe their intended effect, and indicate which implementations of them are available in LIP. Depending on the application it is always a good idea to try compilation with the '`-DSINGLE_MUL`' flag. This flag will be tried automatically if the instructions in (1.2) are followed to compile LIP. See also Remark (3.2).

```
static void zaddmulp(long *a, long b, long d, long *t)
```

Let $x = \mathtt{a} + \mathtt{t} + \mathtt{b} \cdot \mathtt{d}$. As a result of `zaddmulp` the value $x \bmod \mathtt{RADIX} \in \{0, 1, \ldots, \mathtt{RADIX} - 1\}$ is assigned to `a`, and `t` gets the value $[x/\mathtt{RADIX}]$.

Only for non-negative integers `a`, `b`, `d`, and `t` that are all $<$ `RADIX`, so that `a` and `t` are again non-negative and $<$ `RADIX` after the call.

```
static void zaddmulpsq(long *a, long b, long *t)
```

Let $x = \mathtt{a} + \mathtt{b}^2$. As a result of `zaddmulpsq` the value $x \bmod \mathtt{RADIX} \in \{0, 1, \ldots, \mathtt{RADIX} - 1\}$ is assigned to `a`, and `t` gets the value $[x/\mathtt{RADIX}]$.

Only for non-negative integers `a`, `b`, and `t` that are all $<$ `RADIX`, so that `a` and `t` are again non-negative and $<$ `RADIX` after the call.

There are four versions of `zaddmulp` and `zaddmulpsq` available in LIP. The default version presumes a two's complement machine in which integer overflow is ignored and where arithmetic on `doubles` is fast; it uses one `long*long` multiplication, two `double*double` multiplications, and a few additions. If LIP is compiled with the '`-DSINGLE_MUL`' flag (cf. (3.1), (3.2)), another version is used that makes the same assumptions, but that uses only one `double*double` multiplication, and some additions, shifts, and logical operations. The latter version is often faster than the default version, but uses a smaller value of `NBITS`, and is therefore less convenient for certain applications. If neither of these two versions work, LIP can be compiled with the '`-DPLAIN`' or the '`-DKARAT`' flags, which do not assume anything on top of C. The '`-DKARAT`' flag uses less multiplications but more additions than the '`-DPLAIN`' flag, and it is usually faster. Compilation with more than 1 of these flags is not allowed. Obviously, if a particular machine has a 32 × 32 → 64 bit integer multiplier, then it should be possible to write a substantially faster inline assembly version of `zaddmulp` and `zaddmulpsq`.

```
static void zaddmulone(long a[], long b[])
```

Adds $\mathtt{b}[i]$ to $\mathtt{a}[i-1]$, for $i = 1, 2, \ldots, \mathtt{b}[0]$, where the $\mathtt{a}[i]$ and $\mathtt{b}[i]$ are non-negative integers $<$ `RADIX`, and normalizes the results (except for $\mathtt{a}[\mathtt{b}[0]]$):

```
long i;
long carry = 0;
for (i=1; i<=b[0]; i++)
{
        carry += a[i-1] + b[i];
        a[i-1] = carry % RADIX;
        carry /= RADIX;
    }
    a[b[0]] += carry;
```

Notice that the divisions can easily be replaced by mask or shift operations and that a[b[0]] might be $\geq$ `RADIX` after `zaddmulone`. This is equivalent to (but faster than) `zaddmul(1,a,b)` (see below). The arrays `a` and `b` may be assumed to be non-overlapping.

```
static void zaddmul(long d, long a[], long b[])
```

Adds `d` times b[$i$] to a[$i-1$], for $i = 1, 2, \ldots,$ b[0], where `d` and the a[$i$] and b[$i$] are non-negative integers < `RADIX`, and normalizes the results (except for a[b[0]]):

```
    long i;
    long carry = 0;
    for (i=1; i<=b[0]; i++)
    {
        zaddmulp(&(a[i-1]),b[i],d,&carry);
    }
    a[b[0]] += carry;
```

```
static void zaddmulsq(long d, long a[], long b[])
```

Adds b[0] times b[$i$] to a[$i-1$], for $i = 1, 2, \ldots,$ `d`, where the a[$i$] and b[$i$] are non-negative integers < `RADIX`, and normalizes the results (except for a[d]):

```
    long i;
    long carry = 0;
    for (i=1; i<=d; i++)
    {
        zaddmulp(&(a[i-1]),b[i],b[0],&carry);
    }
    a[d] += carry;
```

The above two descriptions assume that `zaddmulp` is a regular C-function; the `&`'s in the calls to `zaddmulp` should of course be removed if `zaddmulp` is a macro. Notice that a[b[0]] in `zaddmul` and a[d] in `zaddmulsq` might be $\geq$ `RADIX` after the call. The arrays `a` and `b` may be assumed to be non-overlapping, except that in `zaddmul` the array `&(a[0])` might be identical to the array `&(b[1])`. If LIP is compiled with the '`-DSINGLE_MUL`' flag (cf. (3.1), (3.2)) but without the '`-DPLAIN`' or '`-DKARAT`' flags, the macros for `zaddmul` and `zaddmulsq` are replaced by ordinary (but often faster) functions. These functions do not call the `-DSINGLE_MUL`-version of `zaddmulp`, but use an explicit pipeline that exploits the fact that on some machines floating point and integer operations can run in parallel.

```
static void zmmulp(long a[])
```

Adds `d` $=$ (a$/n$) mod `RADIX` times zn[$i$] to a[$i-1$], for $i = 1, 2, \ldots,$ zn[0], where $n$ is the current Montgomery modulus (cf. (2.6)), `zn` its internal LIP-representation, the a[$i$] and zn[$i$] are non-negative integers $<$ RADIX, and normalizes the results (including a[zn[0]], but not beyond that):

```
long i;
long carry = 0;
for (i=1; i<=zn[0]; i++)
{
  zaddmulp(&(a[i-1]),zn[i],d,&carry);
}
a[zn[0]] += carry;
if (a[zn[0]] >= RADIX)
{
  a[zn[0]] -= RADIX;
  a[zn[0]+1] ++;
}
```

The multiplier d can easily be computed once the inverse zninv of zn[1] modulo RADIX is known (cf. (1.7)). This inverse is computed in zmstart during the initialization of the Montgomery modulus, and is kept in the global long zninv (unless LIP is compiled with the '-DPLAIN' flag or the '-DKARAT' flag, in which case it is kept in the global longs zninv1 and zninv2 in such a way that $0 \leq$ zninv1, zninv2 $< \sqrt{\texttt{RADIX}}$ and zninv $=$ zninv2 $\cdot \sqrt{\texttt{RADIX}} +$ zninv1. If LIP is compiled with the '-DSINGLE_MUL' flag (cf. (3.1), (3.2)) but without the '-DPLAIN' and '-DKARAT' flags, the macro zmmulp is replaced by an ordinary (but often faster) function that makes an attempt to use pipelining.

If any of the above macros is replaced by assembly language versions it is probably a good idea to do the same for the functions zdiv21, zsubmul, and zmulmods. We refer to lip.c for the source code of these functions.

## 5. Detailed description of some LIP-functions

### (5.1) Multiplication and squaring

Depending on the size of the operands, LIP uses ordinary ('pencil-and-paper', cf. [3: 4.3.1]) multiplication or squaring if at least one of the operands is sufficiently small, and Karatsuba's method (cf. [3: 4.3.3]) otherwise. Karatsuba's method is applied recursively until the ordinary method applies, or until the maximum recursion depth KAR_DEPTH is reached, in which case LIP also resorts to the ordinary method. The default value of KAR_DEPTH is 20, which is more than enough for most applications; it can be set to any value x using the '-DKAR_DEPTH=x' flag.

To decide when an operand is small, LIP uses the crossover constants KAR_MUL_CROV for multiplication and KAR_SQU_CROV for squaring: if an operand has less than KAR_MUL_CROV significant blocks of NBITS bits (i.e., if $|$a[0]$| <$ KAR_MUL_CROV for a verylong operand a, cf. (1.7)), the multiplication functions consider it to be small (and similarly with KAR_SQU_CROV for the squaring functions). Both constants have default value 30, which can be changed in the usual way using the appropriate flags. The optimal values, which depend on the machine, are best determined experimentally. If LIP is compiled following the instructions in (1.2), near optimal values will be selected (and can be found in the file `lippar.h`).

The multiplication and squaring functions in LIP are not intended for extensive use on excessively large integers (say, in excess of $10^5$ bits). For such applications truly 'fast multiplication' functions are more suitable than Karatsuba's method.

### (5.2) Arithmetic on 64-bit architectures

On alpha architectures, longs are 64 bits and doubles have a 52-bit mantissa. If one wants to use $\texttt{NBITS} > 50$, then the normal division algorithm may take a long time, or it may give an incorrect answer. LIP has special code to efficiently perform division on the alpha so that one can use $\texttt{NBITS} = 62$. To use this code, compile LIP with the '`-DALPHA50`' or '`-DALPHA`' flag. Both flags will work, but either one may give more efficient code depending on the application.

The `ALPHA50` code will convert the input variables `a` and `b` to a 50-bit radix representation and then perform the division. The result is converted back to a 62-bit radix representation.

The `ALPHA` code performs the division with a 62-bit radix. In order to do this, it must be able to quickly compute the quotient $q = \frac{a}{b}$, where $a < \texttt{RADIX}^3$ and $\texttt{RADIX} \leq b < \texttt{RADIX}^2$. Let $R$ be the `RADIX`. The normal division code will convert $a$ and $b$ to a double representation, which we will denote by $\tilde{a}$ and $\tilde{b}$. Then it computes $\tilde{q} = \tilde{a}/\tilde{b}$. However, because the double has less bits in the mantissa than the long on the alpha, $|q - \tilde{q}|$ may be as large as $2^{12}$. To deal with this problem, we first compute $\tilde{q_1} = (\frac{R-3072}{R}) \cdot \tilde{a}/\tilde{b}$. Then we set $r = a - \tilde{q_1} \cdot b$ and compute $\tilde{q_2} = (\frac{R+3072}{R}) \cdot r/\tilde{b}$. Then the value of $\tilde{q_1} + \tilde{q_2}$ is no more than 1 larger than $q$, which is close enough. This is briefly explained below.

If $x$ is a long, and $y =$ (double) $x$ is a double, then $x(1 - \frac{1}{2^{54}}) \leq y \leq x(1 + \frac{1}{2^{54}})$ due to rounding error in the conversion. So

$$\tilde{q_1} \leq (\frac{R-3072}{R}) \cdot \frac{\tilde{a} \cdot (1 + \frac{1}{2^{54}})}{\tilde{b} \cdot (1 - \frac{1}{2^{54}})}$$

$$= (\frac{R-3072}{R}) \cdot q \cdot (1 + \frac{1}{2^{54}})/(1 - \frac{1}{2^{54}}) < q.$$

Also,

$$\tilde{q_1} \geq (\frac{R-3072}{R}) \cdot \frac{\tilde{a} \cdot (1 - \frac{1}{2^{54}})}{\tilde{b} \cdot (1 + \frac{1}{2^{54}})}$$

$$= (\frac{R-3072}{R}) \cdot q \cdot (1 - \frac{1}{2^{54}})/(1 + \frac{1}{2^{54}})$$

$$> (1 - \frac{1}{2^{50}}) \cdot q > q - 2^{12}$$

because $q < 2^{62}$. Thus, $0 \leq r \leq (2^{12} + 1) \cdot b$ and $0 \leq \frac{r}{b} \leq 2^{12} + 1$. If $q_2 = \frac{r}{b}$ then $q_2 < \tilde{q_2} < q_2 + 1$ which is obtained in the same way that $\tilde{q_1}$ is bounded. Hence $q < \tilde{q_1} + \tilde{q_2} < q + 1$.

### (5.3) Elliptic curve method

The implementation of zecm is described in detail in [2]. Here we give a superficial description of the method, and we explain some of the constants that can be changed by the user. Let n be the number to be factored, i.e., the first argument of zecm. First of all, the elliptic curve method (ecm) is a probabilistic method: ecm cannot be guaranteed to work, but only has a certain probability of success that depends on the size of the factors of n and the choice of the other parameters. The probability of success is higher for smaller factors. The method consists of a number of independent trials; in zecm the maximum number of trials is given in nb, its fourth argument. Each trial consists of two phases. In the first phase a random elliptic curve $C$ modulo n and a point $x$ on $C$ are selected, and $x^k$ is computed on $C$. The value for $k$ depends on the first phase bound (bnd, the fifth argument of zecm), and is roughly equal to the product of all prime powers less than bnd. If $C$ is very lucky the computation of $x^k$ fails and a factor of n is detected, in which case zecm returns 1. Otherwise, if $x^k$ has been computed successfully, the computation moves to the second phase.

In the second phase powers of $x^k$ are computed and compared, as described in [2]. If $C$ is lucky a factor of n will be detected at the end of the second phase, and zecm returns 1. Otherwise, a new random curve might be selected for a new factoring attempt, until n has been factored, or until nb trials have been carried out.

As explained in (2.9), the sequence of curves depends on s, the third argument of zecm. Although it cannot be guaranteed it is quite likely that two different choices for s (and reasonably small choices for nb) lead to two disjoint sequences of curves. After each curve the first phase bound is increased by a certain percentage (given by percent, the sixth argument of zecm). A good choice would for instance be 5 or 10 percent. If the initial or thus computed first phase bound is less than ECM_MINBOUND it will be replaced by ECM_MINBOUND, and similarly if it is larger than ECM_MAXBOUND it will be replaced by ECM_MAXBOUND. The default settings for ECM_MINBOUND and ECM_MAXBOUND are 10 and 5000000, respectively, which can be changed by using the appropriate flags.

There are four ecm-related constants in LIP that require more background, for which we refer to [2]. In the first phase ECM_BATCH prime powers will be combined and one inversion modulo n will be carried out per batch. Using the '-DECM_BATCH=x' flag the batch size can be changed to any (integer) value x, cf. the last paragraph of [2: Section 5]. On a Sparc 10 '-DECM_BATCH=10' is somewhat faster than the default value 1, but on a DEC 5000 the default setting is better.

The other three constants are related to [2: (6.3)]. The bound used for the second phase ($B_2$ in [2: (6.3)]) equals ECM_MULT $= 10$ times the first phase bound, which is, for our implementation, close to optimal. The largest value for $e$ and $t$ as in [2: (6.3)] are given by ECM_MAXE and ECM_MAXT, respectively; the default settings are ECM_MAXE $= 60$ and ECM_MAXT $= 12$. All these constants can be changed in the usual way with the appropriate flags.

The choice of the number of curves, the first phase bound, and the growth percentage per curve, all strongly depend on the application and the information that might be known about the number n to be factored. In the table below the optimal choices for nb and bnd are given as a function of the number of decimal digits of the smallest factor $p$ of n, and for a probability of success of 60%. So, if $p$ is known to have 15 digits, then nb $= 62$, bnd $= 830$, and percent $= 0$ are good choices. In general the number of digits is $p$ is not known. In that case it is better to take a low bnd for the initial curves, and to let bnd grow with the number of curves. For instance, nb $= 100$, bnd $= 125$, and percent $= 5$ gives a good chance to find factors in the 10 to 20 digit range.

| $\log_{10} p$ | nb | bnd |
|---|---|---|
| 12 | 50 | 125 |
| 13 | 53 | 250 |
| 14 | 57 | 500 |
| 15 | 62 | 830 |
| 16 | 68 | 1500 |
| 17 | 75 | 2500 |
| 18 | 85 | 4200 |
| 19 | 100 | 6500 |
| 20 | 120 | 10000 |
| 21 | 145 | 15000 |
| 22 | 175 | 22000 |
| 23 | 210 | 32000 |
| 24 | 250 | 45000 |
| 25 | 300 | 65000 |
| 26 | 400 | 85000 |
| 27 | 500 | 115000 |
| 28 | 650 | 155000 |
| 29 | 750 | 205000 |
| 30 | 950 | 275000 |

## Appendix A: alphabetical listing of all LIP-functions

double getstime(void): (2.12)
double gettime(void): (2.12)
double getutime(void): (2.12)
void printtime(FILE *f): (2.12)
void starttime(void): (2.12)
long z2div(verylong a, verylong *b): (2.2)
void z2expmod(verylong e, verylong n, verylong *b): (2.5)
long z2log(verylong a): (2.1)
long z2logs(long a): (2.1)
long z2mod(verylong a): (2.2)
void z2mul(verylong a, verylong *b): (2.2)
void zabs(verylong *a): (2.1)
void zadd(verylong a, verylong b, verylong *c): (2.2)

void zaddmod(verylong a, verylong b, verylong n, verylong *c): (2.5)

void zand(verylong a, verylong b, verylong *c): (2.4)

void zbastoz(verylong a, verylong b[], long c, verylong *d): (2.1)

long zbfread(FILE *f, verylong *a): (2.3)

long zbfwrite(FILE *f, verylong a): (2.3)

long zbit(verylong a, long b): (2.4)

void zcat(verylong a, verylong b, verylong *c): (2.4)

void zchirem(verylong a, verylong xa,
verylong b, verylong xb, verylong *c): (2.7)

long zcompare(verylong a, verylong b): (2.1)

long zcomposite(verylong *a, long t, long first): (2.9)

void zcopy(verylong a, verylong *b): (2.1)

long zdefault_m(long a): (2.5)

void zdiv(verylong a, verylong b, verylong *c, verylong *d): (2.2)

void zdivmod(verylong a, verylong b, verylong n, verylong *c): (2.5)

double zdlog(verylong a, double b): (2.1)

double zdoub(verylong a): (2.1)

long zdsa_check_private_key(verylong p, verylong q, verylong g,
verylong x, verylong y, long t): (2.11)

long zdsa_check_public_key(verylong p, verylong q, verylong g,
long t): (2.11)

long zdsa_make_key(long lp, long lq, long t, verylong p,
verylong q, verylong g, verylong x, verylong y,
void(*u)(verylong,verylong*)): (2.11)

long zdsa_sign(verylong p, verylong q, verylong g, verylong m,
verylong x, verylong *k, verylong *r, verylong *s,
void(*u)(verylong,verylong*)): (2.11)

long zdsa_verify(verylong p, verylong q, verylong g, verylong m,
verylong y, verylong r, verylong s): (2.11)

long zecm(verylong n, verylong *fac, long s, long nb,
long bnd, long percent, long t, long info): (2.9)

void zexpmod(verylong a, verylong e, verylong n, verylong *b): (2.5)

void zexpmod_doub(verylong a1, verylong e1,
verylong a2, verylong e2, verylong n, verylong *b): (2.5)

void zexpmod_doub1(verylong a1, verylong e1,
verylong a2, verylong e2, verylong n, verylong *b): (2.5)

void zexpmod_doub2(verylong a1, verylong e1,
verylong a2, verylong e2, verylong n, verylong *b): (2.5)

void zexpmod_doub3(verylong a1, verylong e1,
verylong a2, verylong e2, verylong n, verylong *b): (2.5)

void zexpmod_m_ary(
verylong a, verylong e, verylong n, verylong *b, long m): (2.5)

long zexpmods(long a, long e, long n): (2.5)

void zexteucl(verylong a, verylong *xa,
verylong b, verylong *xb, verylong *c): (2.7)

```
long zfecm(verylong n, verylong *fac, long s, long *nb,
        long *bnd, long percent, long t, long info, FILE *fp): (2.9)
long zfread(FILE *f, verylong *a): (2.3)
long zfread_b(FILE *f, verylong *a, verylong b, long c): (2.3)
void zfree(verylong *a): (2.13)
long zfwrite(FILE *f, verylong a): (2.3)
long zfwrite_c(FILE *f, verylong a, long b, char *c, char *d): (2.3)
long zfwrite_b(FILE *f, verylong a, verylong b, long c): (2.3)
long zfwriteln(FILE *f, verylong a): (2.3)
long zfwriteln_b(FILE *f, verylong a, verylong b, long c): (2.3)
void zgcd(verylong a, verylong b, verylong *c): (2.7)
void zgcdeucl(verylong a, verylong b, verylong *c): (2.7)
void zgetbits(verylong a, long b, long c, verylong *d): (2.4)
void zhfread(FILE *f, verylong *a): (2.3)
void zhfwrite(FILE *f, verylong a): (2.3)
void zhfwriteln(FILE *f, verylong a): (2.3)
void zhighbits(verylong a, long b, verylong *c): (2.4)
void zhread(verylong *a): (2.3)
void zhsread(char *f, verylong *a): (2.3)
void zhwrite(verylong a): (2.3)
void zhwriteln(verylong a): (2.3)
void zintoz(long d, verylong *a): (2.1)
long zinv(verylong a, verylong b, verylong *c): (2.7)
void zinvmod(verylong a, verylong n, verylong *b): (2.5)
long zinvodds(long a, long b): (2.7)
long zinvs(long a, long b): (2.7)
long zispower(verylong n, verylong *f): (2.2)
long ziszero (verylong a): (2.1)
long zjacobi(verylong a, verylong b): (2.7)
long zjacobis(long a, long b): (2.7)
double zln(verylong a): (2.1)
double zlog(verylong a, verylong b): (2.1)
void zlowbits(verylong a, long b, verylong *c): (2.4)
void zlshift(verylong a, long b, verylong *c): (2.4)
long zmakeodd(verylong *a): (2.4)
long zmcomposite(verylong a, long t): (2.9)
void zmfree(void): (2.6)
void zmod(verylong a, verylong b, verylong *d): (2.2)
void zmontadd(verylong ma, verylong mb, verylong *mc): (2.6)
void zmontdiv(verylong ma, verylong mb, verylong *mc): (2.6)
void zmontexp(verylong ma, verylong e, verylong *mb): (2.6)
void zmontexp_doub(verylong ma1, verylong e1,
                     verylong ma2, verylong e2, verylong *mb): (2.6)
void zmontexp_doub1(verylong ma1, verylong e1,
                     verylong ma2, verylong e2, verylong *mb): (2.6)
```

```
void zmontexp_doub2(verylong ma1, verylong e1,
                    verylong ma2, verylong e2, verylong *mb): (2.6)
void zmontexp_doub3(verylong ma1, verylong e1,
                    verylong ma2, verylong e2, verylong *mb): (2.6)
void zmontexp_m_ary(verylong ma, verylong e, verylong *mb,
                                                long m): (2.6)
void zmontinv(verylong ma, verylong *mb): (2.6)
void zmontmul(verylong ma, verylong mb, verylong *mc): (2.6)
void zmontsq(verylong ma, verylong *mb): (2.6)
void zmontsub(verylong ma, verylong mb, verylong *mc): (2.6)
void zmstart(verylong n): (2.6)
void zmtoz(verylong ma, verylong *a): (2.6)
void zmul(verylong a, verylong b, verylong *c): (2.2)
void zmul_plain(verylong a, verylong b, verylong *c)): (2.2)
void zmulin(verylong a, verylong *b): (2.2)
void zmulinmod(verylong a, verylong *b, verylong n): (2.5)
void zmulmod(verylong a, verylong b, verylong n, verylong *c): (2.5)
long zmulmods(long a, long b, long n): (2.5)
long zmulmod26(long a, long b, long n, double c): (2.5)
void znegate(verylong *a): (2.1)
void znot(verylong a, verylong *b): (2.4)
long zodd(verylong a): (2.4)
void zone(verylong *a): (2.1)
void zor(verylong a, verylong b, verylong *c): (2.4)
long zp(void): (2.10)
long zpnext(void): (2.10)
long zpnextb(long bnd): (2.10)
long zpollardrho(verylong n, verylong *res, verylong *cof,
                                                long t): (2.9)
long zprime(verylong a, long t, long first): (2.9)
long zprobprime(verylong a, long t): (2.9)
void zpstart(void): (2.10)
void zpstart2(void): (2.10)
long zrandom(long bnd): (2.8)
void zrandomb(verylong bnd, verylong *a): (2.8)
long zrandomfprime(long lq, long t, verylong fac,
     verylong *p, verylong *q, void(*u)(verylong,verylong*)): (2.10)
long zrandomgprime(long lq, long t, long small, verylong *p,
     verylong *q, verylong *g, void(*u)(verylong,verylong*)): (2.10)
void zrandoml(long length, verylong *a,
                                void(*u)(verylong,verylong*)): (2.8)
long zrandomprime(long length, long t, verylong *p,
                               void(*u)(verylong,verylong*)): (2.10)
long zrandomqprime(long lp, long lq, long t, verylong *p,
  verylong *q, verylong *quot, void(*u)(verylong,verylong*)): (2.10)
```

long zread(verylong *a): (2.3)
void zreverse(verylong a, verylong *b): (2.4)
long zreverses(long a): (2.4)
long zroot(verylong a, long b, verylong *c): (2.2)
void zrshift(verylong a, long b, verylong *c): (2.4)
void zrstart(verylong s): (2.8)
void zrstarts(long s) : (2.8)
void zsadd(verylong a, long b, verylong *c): (2.2)
void zsbastoz(long a, long b[], long c, verylong *d): (2.1)
long zscompare(verylong a, long b): (2.1)
long zsdiv(verylong a, long b, verylong *c): (2.2)
long zsetbit(verylong *a, long b): (2.4)
void zsetlength(verylong *a, long length, char *s): (2.13)
void zsexpmod(verylong a, long e, verylong n, verylong *b): (2.5)
long zshighbits(verylong a, long b): (2.4)
long zsign(verylong a): (2.1)
double zslog(verylong a, long b): (2.1)
long zslowbits(verylong a, long b): (2.4)
long zsmod(verylong a, long b): (2.2)
void zsmontmul(verylong ma, long d, verylong *mc): (2.6)
void zsmul(verylong a, long b, verylong *c): (2.2)
void zsmulmod(verylong a, long b, verylong n, verylong *c): (2.5)
void zsq(verylong a, verylong *b): (2.2)
void zsq_plain(verylong a, verylong *b): (2.2)
void zsqin(verylong *a): (2.2)
void zsqinmod(verylong *a, verylong n): (2.5)
void zsqmod(verylong a, verylong n, verylong *c): (2.5)
long zsqrt(verylong a, verylong *b, verylong *c): (2.2)
void zsqrtmod(verylong a, verylong p, verylong *s): (2.5)
long zsqrts(long a)): (2.2)
long zsquf(verylong n, verylong *f1, verylong *f2): (2.9)
long zsread(char *f, verylong *a): (2.3)
void zstart(void): (2.1)
long zstobas(verylong a, long b, long c[], long *d): (2.1)
long zstosymbas(verylong a, long b, long c[], long *d): (2.1)
long zstrtoz(char *a, verylong *c): (2.1)
long zstrtozbas(char *a, long b, verylong *c): (2.1)
void zsub(verylong a, verylong b, verylong *c): (2.2)
void zsubmod(verylong a, verylong b, verylong n, verylong *c): (2.5)
void zsubpos(verylong a, verylong b, verylong *c): (2.2)
void zswap(verylong *a, verylong *b): (2.1)
long zswitchbit(verylong *a, long b): (2.4)
long zswrite(char *f, verylong a): (2.3)
long ztobas(verylong a, verylong b, verylong c[], long *d): (2.1)
long ztoint(verylong a): (2.1)

`void ztom(verylong a, verylong *ma)`: (2.6)
`long ztosymbas(verylong a, verylong b, verylong c[], long *d)`: (2.1)
`unsigned long ztouint(verylong a)`: (2.1)
`long ztoul(verylong a, unsigned long b[], long *c)`: (2.1)
`long ztridiv(verylong a, verylong *cofac, long low, long high)`: (2.9)
`void zuintoz(unsigned long d, verylong *a)`: (2.1)
`void zultoz(unsigned long a[], long b, verylong *c)`: (2.1)
`long zweight(verylong a)`: (2.4)
`long zweights(long a)`: (2.4)
`long zwrite(verylong a)`: (2.3)
`long zwriteln(verylong a)`: (2.3)
`void zxor(verylong a, verylong b, verylong *c)`: (2.4)
`void zzero(verylong *a)`: (2.1)

## Appendix B: alphabetical listing of all compilation flags

`-DALPHA`

LIP will use 62 bit `RADIX`. See (5.2)

`-DALPHA50`

LIP will use 62-bit `RADIX` with 50-bit division. See (5.2)

`-DCHARL=x`

LIP will assume that there are `x` bits per byte. Default value 8. See (3.1).

`-DDOUBLES_LOW_HIGH`

LIP will assume that the order of the words in `doubles` is ‘low-high’. Default assumption is ‘high-low’. See (3.1).

`-DECM_BATCH=x`

Default value 1. See (5.3).

`-DECM_MAXE=x`

Default value 60. See (5.3).

`-DECM_MAXT=x`

Default value 12. See (5.3).

`-DECM_MULT=x`

Default value 10. See (5.3).

`-DFREE`

Local `verylong` variables will not be declared `static`. Therefore, space for `verylong` variables will be allocated (and freed) for each call to the function where they appear. See (1.8).

`-DHEX_BLOCK=x`

Hexadecimal output will be printed in blocks of at most `x` characters. Default value 8. See (3.3) and (2.3).

`-DHEX_BLOCKS_PER_LINE=x`

Hexadecimal output will be given in at most `x` blocks per line. Default value 7. See (3.3) and (2.3).

`-DIN_LINE=x`

The maximum number of characters per line upon input of a verylong variable will be set to x. Default value 2048. See (3.3) and (2.3).

-DKARAT

LIP will use macros that do not assume anything on top of C, and that use three long * long multiplies to get the product of two NBITS-bits integers. See (4.1).

-DKAR_DEPTH=x

The maximum recursion depth of Karatsuba multiplication and squaring will be set to x. Default value 20. See (5.1).

-DKAR_MUL_CROV=x

The crossover point between regular multiplication and Karatsuba multiplication will be set at x blocks of NBITS nits. Default value 30. See (5.1).

-DKAR_SQU_CROV=x

The crossover point between regular squaring and Karatsuba squaring will be set at x blocks of NBITS nits. Default value 30. See (5.1).

-DHEX_LOWER_CASE

Hexadecimal digits 10 though 15 will be printed as 'a' through 'f'. See (2.3): zhfwrite, zhwrite.

-DNBITS=x

LIP will split verylongs in pieces of x bits each. Default value 30, unless (2.1) is used to compile LIP and the timer-program decides to used the '-DSINGLE_MUL' flag, in which case the default vale will be 26. See (3.1) and (1.7).

-DNO_ALLOCATE

LIP will assume that the user calls zsetlength for the initial allocation of all verylong variables. See (2.13).

-DNO_ECM

The LIP-function zecm as described in (2.9) will not be compiled. See (2.9).

-DNO_HALT

Forces LIP to continue after an error has been detected. See (1.6).

-DOUT_LINE=x

The approximate maximum number of characters per line upon output of a verylong variable will be set to x. Default value 68. See (2.3): zfwrite(ln), zswrite, zwrite(ln).

-DPLAIN

LIP will use macros that do not assume anything on top of C, and that use four long * long multiplies to get the product of two NBITS-bits integers. See (4.1).

-DPRIM_BND=x

The upper bound on the primes generated by the small prime generator will be set to approximately $(2 \cdot \mathtt{PRIM\_BND} + 1)^2$. Default value $2^{\mathtt{NBITS}/2-1}$. See (3.3) and (2.10): zpnext and zpnextb.

-DPRT_REALLOC

LIP will print messages on standard error about (re)allocation of `verylong` variables. See (2.13).

`-DSINGLE_MUL`

LIP will use `NBITS` = 26 and a different set of macros. See (4.1).

`-DSIZE=x`

The minimal allocation length for `verylong`s will be set to `x`. Default value 20. See (3.3).

`-DSIZEOFLONG=x`

LIP will assume that the values of `sizeof(long)` is `x`. Default value 4. See (3.1).

`-DSTART`

LIP will assume that the user calls `zstart` once, before using any other LIP-function. See (2.1): `zstart`.

## References

1. D. Atkins, M. Graff, A.K. Lenstra, P.C. Leyland, *THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE*, Proceedings Asiacrypt'94, to appear.
2. W. Bosma, A. K. Lenstra, *An implementation of the elliptic curve integer factorization method*, Proceedings Computational Algebra and Number Theory Conference, Sydney 1992, to appear.
3. D.E. Knuth, *The art of computer programming*, volume 2, *Seminumerical algorithms*, second edition, Addison-Wesley, Reading, Massachusetts, 1981.
4. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, J. M. Pollard, *The factorization of the ninth Fermat number*, Math. Comp. **61** (1993), 319–349.
5. A. K. Lenstra, M. S. Manasse, *Factoring by electronic mail*, Advances in cryptology, Eurocrypt '89, Lecture Notes in Comput. Sci. **434** (1990), 355–371.
6. P. L. Montgomery,, *Modular multiplication without trial division*, Math. Comp. **44** (1985), 519–521.
7. NIST, *A proposed federal information processing standard for digital signature standard (DSS)*, Federal Register **56** (1991), 42980–42982.
8. S.-M. Yen, C.-S. Laih, A. K. Lenstra, *Multi-exponentiation*, IEE Proc.-Comput. Digit. Tech. **14** (1994), 325–326.

Room MRE-2Q334, Bellcore, 445 South Street, Morristown, NJ 07960, U. S. A.
*E-mail address*: lenstra@bellcore.com