ZEN

A new toolbox for computing in finite extensions of finite rings

User's manual

F. Chabaud
fchabaud@free.fr
R. Lercier
lercier@celar.fr

DCSSI
18 rue du Dr. Zamenhoff
92131 Issy-les-Moulinaux
France

Centre d'Électronique de l'Armement
CASSI/SCY/EC
35998 Rennes Armées
France

December 17, 2000

Some parts of this work was performed during Ph.D thesis of the authors, respectively at "Laboratoire d'Informatique de l'École Normale Supérieure - Paris 75230 Cedex 05" and "Laboratoire d'Informatique de l'École Polytechnique - Palaiseau 91128 Cedex". These two institutions are affiliated to the "Centre National pour la Recherche Scientifique".

Contents

Chapter 1

Introduction

Many computational problems need arithmetic operations in polynomial finite rings of $\mathbb{Z}/n\mathbb{Z}$ where n is an integer ($n > 1$). Integer factorizations, primality testing are for instance such applications. To solve them, programmers use general symbolic mathematical softwares or write specific programs (most of the time in C[]).

On the first hand, symbolic mathematical softwares (Maple[], Mathematica[],...) handle with difficulty computations in finite fields. In the worst cases, such programs perform computations with rationals before finally reducing the objects modulo the characteristic n, in the best cases, such reductions are performs but extensions of a finite ring cannot be implemented. In any cases, applications written with such softwares are ten to hundred times slower than an "ad hoc" implementation in C. On the other hand, optimized C libraries (CESAR, Lidia) deal only with one side of finite fields, mainly $\mathbb{Z}/n\mathbb{Z}$.

We hardly believe we can keep the efficiency of these C libraries while working in any polynomial extension of $\mathbb{Z}/n\mathbb{Z}$. We designed the ZEN library to perform efficient arithmetic operations in these sets.

Via oriented object concepts programmed in C, you can work in the same way not only in any polynomial extension of $\mathbb{Z}/n\mathbb{Z}$, but also in any polynomial extension of another finite ring even if n is not a prime or even if the polynomial which defines an extension is not irreducible. The current finite ring is an argument of any procedure of this library. So, once a program is written for a given finite field, for instance $\mathbb{Z}/2\mathbb{Z}$, only few minor changes will be necessary to make it work in other finite fields, for instance $\mathbb{Z}/5\mathbb{Z}$, $\mathbb{Z}/18446744073709551629\mathbb{Z}$, $(\mathbb{Z}/2\mathbb{Z})[t]/(t^{10}+t^3+1)$, $((\mathbb{Z}/1753\mathbb{Z})[t](t^2+7))[u]/(u^2+t)$,...or even in rings, for instance in $\mathbb{Z}/1024\mathbb{Z}$, $(\mathbb{Z}/2\mathbb{Z})[t]/(t^{10}+1)$,...In this later case, the functions of ZEN still works but exceptions are raised if inverses cannot be computed.

To combine simplicity and efficiency, the procedures needed to handle elements, polynomials, matrices,...are set at the running time while initializing the current ring because the computer structure of these objects depends on this ring. So, even if the syntax of our functions is always the same, the procedures dynamically called are functions of the ring.

Two other original features of ZEN are the "precomputation" and "clone" concepts. In practice, it well known we can improve some algorithms at the expense of precomputations. Nevertheless, precomputations can take time! In ZEN, no precomputation is done by default but the user can perform some to

speed up multiplications, exponentiations,.... To that end, ZEN provides a procedure whose arguments are a ring and a flag to control the way precomputations are added to the ring.

It is well known too you can increase performances by changing the way you represent elements of a finite fields. A ring is defined by default by its polynomial basis in ZEN. But, some operations can be significantly speeded up if you change its representation. For instance, if you are working in a finite field, you can look for a generator and then working in the set of indexes. In ZEN, these representations are called "clones" and you can compute clones of any rings. To that end, ZEN provides a procedure whose arguments are a ring and a flag to control which clone you want to compute. Obviously, you can then add precomputations to a clone.

This library can be used at two levels.

1. For a current usage, the functions of the library can be used to perform operations on elements, polynomials, matrices, series and elliptic curves over every polynomial extension over $\mathbb{Z}/n\mathbb{Z}$ as described previously. Here, you only have to include zen.h and use the types and the functions of ZEN in your C sources and link your object files with the library libzen.a. These functionalities are described in this manual.

2. For advanced users who really need to gain more efficiency, it is possible to replace procedure calls by macros in their own functions. But, such users have to know the internal data structures of the library and to write specific functions for each structure if they still want to handle any finite ring. These users are then sure that their applications are not penalized by inopportune procedure calls.

Parts of this library was developped during the thesis of the authors [1, 4].

Chapter 2

Beginning with ZEN

We here suppose the reader familiar with the simple notions of algebra such as set, group, ring, field, equivalence relations, equivalence class, etc... The purpose of this section is to recall the construction of a finite field, and to make the parallel with the library functions. The detailed definitions of the notions used in this section can be found in any first degree algebra course.

2.1 Modular rings

The basis of every construction of a finite field is a prime field $\mathbb{Z}/_{p\mathbb{Z}}$, with p a prime integer. This field is the set of integers modulo p. More generally, for all integer n, the set of integers modulo n is a finite ring.

In C language, mathematical objects are represented by types. The ZEN library therefore defines some types corresponding to the mathematical objects. THe first of these types is the ZENRing one which stands for a ring. In order to build a C representation R of the finite ring $R = \mathbb{Z}/_{n\mathbb{Z}}$, we need first to declare the variable ZENRing R; and then to build it using a C representation of n. As ZEN is based upon BigNum, we use this representation for integers. Hence, we will need a couple BigNum n; BigNumLength nl; to represent n. We will describe more precisely what a BigNum is later, but for the moment, we can use the following function to set n to the value n: ZBNReadFromString(&n,&nl,"1234567",10). This makes the couple (n,nl) represent $n = 1234567$ in base 10. Now we can build a representation of $R = \mathbb{Z}/_{1234567\mathbb{Z}}$ using the intended function of ZEN: ZENBaseRingAlloc(R,n,nl).

Once the ring initialized, one can use the mathematical objects defined on it, using the same type of programmation. For instance an element will be declared as a ZENElt E;, and allocated using ZENEltAlloc(E,R). You can use this element in ZEN functions, for instance set it to one: ZENEltSetToOne(E,R). For efficiency reasons, there is no garbage collector in ZEN. Hence, it is necessary to free the objects after use. For an element, the function to use is ZENEltFree(E,R), and for a ring ZENRingClose(R).

2.2 Ring extensions

Let's recall that if a set of cardinality n is a finite field, then n is a power of a prime $n = p^m$, and there exists isomorphic representations of this set using an irreducible polynomial P over $\mathbb{Z}/_{p\mathbb{Z}}$. More generally, the quotient structure $\mathbb{Z}/_{n\mathbb{Z}}[X]/_{P(X)}$ is a finite field if and only if n is prime and P is irreducible.

One of the advantage of ZEN, is to allow easy use of polynomial extensions. For instance, if we want to define a polynomial extension over the above ZENRing R, we will need to declare a polynomial ZENPoly P, define it over R, for instance with the function ZENPolyReadFromString, and use it to build the extension ZENRing R2 with ZENRingExtAlloc(R2,P,R).

Now the main advantage of ZEN is that there is no change in the syntax of functions whether you work in a modular ring, an extension or a tower of extensions. It is therefore possible to write generic programs, and choose the ring or field of definition dynamically when they are used, even if the mathematical structure differs.

2.3 A programming example

As this is certainly the best way to understand how to make programs that use ZEN, here is a small example. A more complex and more interesting example is given in the ZENFACT documentation.

The purpose of this program is to perform inversion of some elements in some finite rings. This is a very simple example but it should be of some help for understanding the philosophy of ZEN.

2.3.1 Preliminaries

The first thing to do is to include some standard libraries header files.

```
#include <stdlib.h>
#include <stdio.h>
#include "zen.h"
```

We do not need any function in this program, due to its simplicity, but we begin our main function by the declarations we need.

```
main()
{
  BigNum q,r;
  BigNumLength ql,rl;
  ZENElt A,I,E0,E1,EE0;
  ZENRing K,R,E,EE;
  ZENPoly P,Q,PR;
```

We do not need other objects in this example. That's why we don't have here any ZENMat, ZENSr, nor ZENEc. Nevertheless, the operations that use these types are similar in their syntax to the folowwing ones.

2.3.2 Modular rings

Our example will first consist in building the finite field $\mathbb{Z}/_{234776683\mathbb{Z}}$. For this purpose we need a BigNum representing the modulo. We can for instance use the following function to this purpose.

```
ZBNReadFromString(&q,&ql,"234776683",10);
```

In order to check the correctness of the implementation, we now print the obtained BigNum.

```
printf("q="); ZBNPrintToFile(stdout,q,ql,10); printf("\n");
```

Let's see now how to build our finite field. The syntax of the function is rather simple, and affects to the K variable the structure we want.

```
ZENBaseRingAlloc(K,q,ql);
```

If we want to invert 234675 in this field, we will need two elements, one for the data, the other for the result. We first need to allocate them and as they are elements of K, the syntax we use is as follows.

```
ZENEltAlloc(A,K); ZENEltAlloc(I,K);
```

Now, we can set the data and check the value.

```
ZENEltReadFromString(A,"234675",10,K);
printf("A="); ZENEltPrintToFile(stdout,A,10,K); printf("\n");
```

In order to obtain the result, we need a call to the suited function.

```
ZENEltInverse(I,A,K);
printf("1/A mod q = ");
ZENEltPrintToFile(stdout,I,10,K);
printf("\n");
```

As we no more need these values, we mustn't forget to free the memory they use. This point is important for larger programs that could grow indefinitely in memory without such precautions.

```
ZENEltFree(A,K); ZENEltFree(I,K);
```

We now take the invert of the same value, but considering it in another structure $\mathbb{Z}/_{8745287453\mathbb{Z}}$. The fact is that this is not a field. Hence, in such a ring, we can no more ignore the return values of the inversion function, as this can be the signal of a mathematical incoherence. We continue to ignore the returned values of the first functions for simplicity purpose.

```
ZBNReadFromString(&r,&rl,"8745287453",10);
printf("r="); ZBNPrintToFile(stdout,r,rl,10); printf("\n");
```

```
ZENBaseRingAlloc(R,r,rl);

ZENEltAlloc(A,R); ZENEltReadFromString(A,"234675",10,R);
printf("A="); ZENEltPrintToFile(stdout,A,10,R); printf("\n");
ZENEltAlloc(I,R);
```

Now, we take care of the possible exception. Of course, as the example was chosen, we will find here a factor of the modulus.

```
if(ZENEltInverse(I,A,R)==ZEN_NO_INVERSE) {
  printf("Non invertible element : modulus factor = ");
  ZENEltPrintToFile(stdout,ZENRingFact(R),10,R); printf("\n"); }
else {
  printf("1/A mod r = ");
  ZENEltPrintToFile(stdout,I,10,R);
  printf("\n"); }
```

In fact, the set of possibly returned values is slightly larger, but it is unuseful to here detail all the other possibilities. We don't forget to free our variables before proceeding, including this ring.

```
ZENEltFree(A,R); ZENEltFree(I,R); ZENRingClose(R);
```

2.3.3 Extension rings

We will now work in a slightly more complex structure, namely

$$\mathbb{Z}/234776683\mathbb{Z}[X]/X^3 + 3234234X^2 + 234234X + 124123.$$

This example is such that we have again a finite field. The definition of this new ZENRing has the same structure as before. We first define a modulus and use it to define the field.

```
ZENPolyReadFromString(P,
                      "(1)*X^3+(3234234)*X^2+(234234)*X+(124123)",
                      10,K);
printf("P="); ZENPolyPrintToFile(stdout,P,10,K);

ZENExtRingAlloc(E,P,K); ZENPolyFree(P,K);
```

Now, if we want to invert $234234t + 3234234$ in this field, we just have to write the following.

```
ZENEltAlloc(E0,E);
ZENEltReadFromString(E0,"(0)*t^2+(234234)*t+(3234234)",10,E);
printf("E0="); ZENEltPrintToFile(stdout,E0,10,E); printf("\n");
```

The syntax of the strings used for polynomials could appear complex at this point, but the apparently redundant parenthesis will soon show their usefulness in towers of extensions. We now perform the inversion.

```
ZENEltAlloc(E1,E); ZENEltInverse(E1,E0,E);
printf("1/E0 mod q = ");
ZENEltPrintToFile(stdout,E1,10,E);
printf("\n");
```

Of course, all this could appear heavy for such a simple operation, but the important point here is that all the strings used to define the parameters of this example could be defined dynamically in the program. In this case, the program is compiled once for all and can work for every structure.

2.3.4 Double extension

We continue to increase complexity. We denote as follows the two structures we have just defined:

$$\begin{array}{rcl} K & = & \mathbb{Z}/_{234776683\mathbb{Z}} \\ E & = & K[t]/_{t^3+3234234t^2+234234t+124123} \end{array}$$

and we now build the following structure:

$$E[X]/_{X^2+(124123t^2+234234)X+(234234t+3234234)}$$

We will use here a construction of the polynomial that uses coefficients. It should be here more readable. We first define the element $124123t^2 + 234234$ over E, and as the element E0 was kept, we can define our new modulus.

```
ZENEltReadFromString(E1,"(124123)*t^2+(234234)",10,E);
printf("E1=");
ZENEltPrintToFile(stdout,E1,10,E);
printf("\n");
```

We now allocate a polynomial, of maximal degree 2, set it to X^2, and then set its coefficients. This procedure is the only guaranteed one to obtain the desired result. In particular, one should not omit to first set the polynomial to its monomial of highest degree, because this is the way to fix the degree of the polynomial. Setting the coefficients, on the contrary, is a fast procedure which does nothing but... setting a coefficient.

```
ZENPolyAlloc(Q,2,E);
ZENPolySetToXi(Q,2,E);
ZENPolySetCoeff(Q,1,E1,E); ZENPolySetCoeff(Q,0,E0,E);
printf("Q=");
ZENPolyPrintToFile(stdout,Q,10,E);
printf("\n");
```

We can now build our double extension using the same function as before.

```
ZENExtRingAlloc(EE,Q,E); ZENPolyFree(Q,E);
```

The following is the setting and inversion of an element in this double extension.

```
 ZENEltAlloc(EE0,EE); ZENPolySetToXi(ZENElt2Pol(EE0,EE),1,E);
 ZENPolySetCoeff(ZENElt2Pol(EE0,EE),1,E1,E);
 ZENPolySetCoeff(ZENElt2Pol(EE0,EE),0,E0,E);
 printf("EE0=");
 ZENEltPrintToFile(stdout,EE0,10,EE);
 printf("\n");

 ZENEltFree(E0,E); ZENEltFree(E1,E);

 ZENEltAlloc(E1,EE); ZENEltInverse(E1,EE0,EE);
 printf("1/EE0=");
 ZENEltPrintToFile(stdout,E1,10,EE);
 printf("\n");

 ZENEltFree(E1,EE);
 ZENRingClose(EE); ZENRingClose(E); ZENRingClose(K);
 exit(0);
}
```

2.3.5 Compilation

The compilation of this program will be done by a command like the following one gcc -I../include example.c ../lib/linux/libzen.a -lm, as the file example.c is located in the zen/inputs directory. The obtained output of the a.out executable should then be the following.

```
q=234776683
A=234675
1/A mod q = 11532995
r=8745287453
A=234675
Non invertible element : modulus factor = 7
P=(1)*X^3+(3234234)*X^2+(234234)*X+(124123)E0=(0)*t^2+(234234)*t+(3234234)
1/E0 mod q = (226257462)*t^2+(25554697)*t+(70525059)
E1=(124123)*t^2+(234234)
Q=((0)*t^2+(1))*X^2+((124123)*t^2+(234234))*X+((0)*t^2+(234234)*t+(3234234))
EE0=((124123)*t^2+(234234))*u+((0)*t^2+(234234)*t+(3234234))
1/EE0=((203916487)*t^2+(83557476)*t+(113630396))*u+ \
       ((138383902)*t^2+(180392990)*t+(206406567))
```

Chapter 3

Understanding ZEN

3.1 The main principles

In this section, we describe the main concepts of ZEN. Mainly the objects it is supposed to handle, the functions working on these objects, the way how precomputations can be performed or what is a clone of a ring to increase performances.

3.1.1 The types of ZEN

The few types of ZEN are as follows. First of all, let's recall that ZEN was originally based on the BigNum library, developed jointly by INRIA and Digital PRL.

BigNum

Large integers. Basically a big integer is an array of BigNumDigit where BigNumDigit is simply an integer, generally of type unsigned long.
Hence, an integer n will be represented by a couple of two C variables (n,nl) of respective types (BigNum, BigNumLength), where the BigNum type is a pointer BigNumDigit $\star$. Value of n will then be

$$n = \sum_{i=0}^{\mathsf{nl}-1} \mathsf{n[i]} \left(2^{\mathsf{SIZE\_BLOC}}\right)^i .$$

If you know the size sl of your integer in bits, the number of BigNumDigits used will be given by the ZEN macro divSizeBloc(sl)+1. The number of bits used in the most significant BigNumDigit is modSizeBloc(sl).

Note 1 *It is important to note that due to an evolution in the* BigNum *library, the length of a* BigNum *is now unsigned. This may cause some bugs in previous implementations on top of* BigNum[1].

[1] For instance, the following loop on a BigNumLength nl will now infinitely run because nl is always larger than zero:

The description of the ZBN layer functions is done in appendix 4.8.

3.1.1.1 The main objects of ZEN

Afterwards, the following types (ZENElt, ZENPoly, ZENSr and ZENMat) are allocated in a "current" ring of type ZENRing.

ZENElt	Any element of a finite ring. This type is in fact an union of elementary types, the size of which are at most the size of a unsigned long or the size of a pointer. In a first approximation, you can see a ZENElt like a polynomial or an integer.
ZENPoly	Any polynomial of a finite ring. This structure can be seen as containing the degree of a polynomial, its allocated length and an array of ZENElts for its coefficients.
ZENSr	Any truncated series of a ZENRing. As algorithms for polynomials can be speeded up for series computations, we introduced the concept of truncated series in ZEN. This structure can be seen as containing the valuation of a series, its truncated degree, its allocated length and an array of ZENElts for its coefficients.

```
for(nl=ZBNNumDigits(n,nl)-1;nl>=0;nl--)
```

Use the following instead:

```
for(nl=ZBNNumDigits(n,nl);nl--;)
```

ZENMat

Any matrix of a ZENRing. It is classically a two-dimensional array of elements. This means that C-implementation involves a differentiation between matrices that does not exist in the mathematical definition, because the access of the (r, c) element of a matrix can be done either by looking at the c-th element of the r-th row, or the r-th element of the c-th column. A ZENMat will therefore be of ZENMatTypeRow if the continuity of rows is kept in the data structure, and of ZENMatTypeCol otherwise. Of course, this has no influence on the result of the operations (except if indicated). However, the performances may greatly change in certain cases. For instance, the permutation of two rows of a matrix will be faster if the matrix is of type ZENMatTypeRow.
Furthermore, a special class of matrices is the class of permutation matrix. A permutation matrix is a square matrix with a single one in every row or column. It can be stored by keeping for every row OR every column the column OR row index of the one. We have therefore two additional types for permutation matrices ZENMatTypeColPermutation and ZENMatTypeRowPermutation. Permutation matrices should be used only by advanced users.
The dimensions of a matrix are of Dim type which is defined in zentypes.h to be int.

As explained previously, we allocate and then handle variables of these types in rings. Rings must have the following type.

ZENRing

This is the main structure of ZEN. It represents one finite ring, that is to say $\mathbb{Z}/n\mathbb{Z}$ where n is any integer (even if it is not prime) or any polynomial finite ring over another (even if the definition polynomial is not irreducible). It contains some general data about finite rings, mainly its characteristic, its degree, its number of elements, its definition polynomial and `pointers to functions` able to act on elements of this finite ring. So each time the user will call a ZEN procedure, it will be probably automatically replaced thanks to macros defined in zen.h by a pointer stored in this structure.

Built over a ring, an elliptic curve has type ZENEc.

ZENEc — An elliptic curve of a finite ring. In ZEN any elliptic curve is a set of couples (X, Y) such that

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6,$$

plus a point at infinity. So such a structure contains the coefficients a_1, a_2, a_3, a_4 and a_6, its discriminant D (it must be different from 0) and invariant J. Moreover, as in a ZENRing pointers on functions are stored here too to perform operations on elliptic curve points. So some other macros defined in zen.h replaced calls of procedures described in this documentation by calls to fields of this structure.

And finally, a point of an elliptic curve has type ZENEcPt.

ZENEcPt — Any point of an elliptic curve defined over any finite ring. It can be the infinity point or a couple (X, Y).

3.1.1.2 Precomputations and clones

We implemented two ways to improve performances in ZEN, "precomputations" and "clones".

Precomputations consist in computing data in advance in order to speed up some operations, typically multiplications or exponentiations. By default we chose to do no precomputation while initializing a ring because it can take a lot of time for large rings. But for some specific applications, this time is quite small in front of the remaining computations. This is why we provide the function ZENAddPrc(Rg, prc) which performs and stores the precomputations specified by the variable prc of type ZENPrc in a ring Rg.

ZENPrc — Structure for handling precomputations. When working in a finite rings, some procedures can be speeded up at the expense (in time and space) of specific precomputations. So, once initialized a finite ring, you can in ZEG add or possibly suppress some precomputations in it. This structure enables the user to specify which precomputations he needs. Then, he only has to give it to the procedures which actually does the precomputations.

Another way to improve performances is to work with another representation of a ring. The default representation is a polynomial representation. Elements in these rings are integers modulo another integer or polynomials modulo another polynomial. There exists numerous other representations. For instance, for small rings, you can tabulate everything and then handle indexes in tabulars, or compute a normal basis and then work in this basis. These representations are called clones in ZEN. You can compute a clone of any ring Rg with the function ZENRingClone(Rg, cln). The type of the clone depends on the parameter cln which is of type ZENCln.

ZENCln	Structure for handling clones. Finite rings can have several representation. The default is the polynomial representation but ZEN can handle other representations, for instance by tabulating the operations.

3.1.2 The functions of ZEN

3.1.2.1 Modular rings

As we work in finite rings, we must begin by building a modular ring. This is done by using the BigNum representation of an integer and a call to the ZENBaseRingAlloc(R,n,nl) procedure. It allocates a ZENRing R; and initializes its parameters.

On success it returns 0 and a non zero exception flag otherwise. We will see later (section 3.1.3) how to use the informations of this flag. This is the normal behavior of all ZEN functions with the exception of the void ones. In fact, if you know that your program should never raise an exception, you can simply not use the return values because in debug mode, the library will not remain silent in case of exceptions.

Now, if we want to use objects on this ring we just have for instance to allocate a ZENElt E: ZENEltAlloc(E,R). Some simple functions with self-contained names allow to initialize the element:

- ZENEltSetToZero(E,R).
- ZENEltSetToOne(E,R).
- ZENZToElt(E,n,nl,R) gives the value of the BigNum (n,nl) to E.
- ZENEltReadFromString(E,s,base,R) reads the string s in base base and affects E accordingly.
- ZENEltReadFromFile(E,fd,base,R) reads the FILE *fd in base base and affects E accordingly.

At the end of computation, one should liberate the memory allocated for this element with the function ZENEltFree(E,R). The figure 3.1 resume the sequence of operations in a modular ring.

Note 2 *It is possible to build the rational field* $\mathbb{Q}$ *by using a* BigNum (n,nl) = 0. *This feature is still experimental, and the efficiency is not guarenteed.*

3.1.2.2 Polynomial extensions

The main advantage of ZEN is that polynomial extensions are very easy to build. As soon as a ZENPoly P is defined with the correct value, an extension ZENRing R2 can be created by the command ZENExtRingAlloc(R2,P,R). This can be repeated as long as wanted with the same syntax. Two elements in a ring Rx are always added by the function ZENEltAdd(A,B,Rx), whatever the ring is. Of course, for efficiency reasons, it will be faster to define a field like $\mathbb{F}_{16}$ as an extension over $\mathbb{F}_2$ of degree 4 instead of a double extension of degree 2, even if these two constructions are isomorphic.

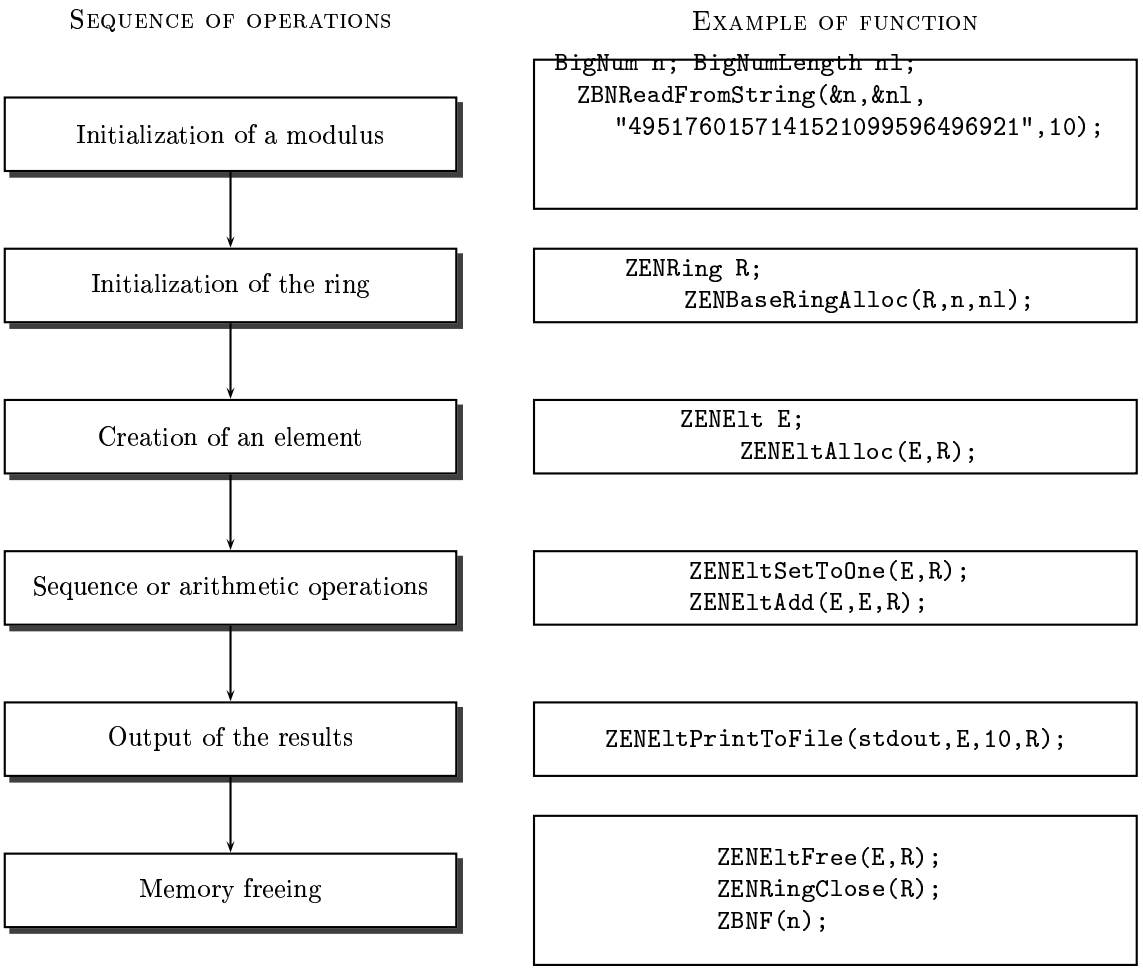

Figure 3.1: Sequence of operations in a modular ring.

Types				Modified	Parameters	Ring	
of arithmetic	of object	of operation		object		used	
ZEN	Elt	Multiply	(	X,	A, B,	R	)

Figure 3.2: ZEN syntax

3.1.2.3 The syntax of ZEN

These small examples show the syntax principles of ZEN that are resumed in figure 3.2

3.1.3 Error handling

Procedures of the ZEN library raise errors by their outputs. There are 2 types of errors, system and mathematical errors.

System errors occur when a function returns ZENERR or ZENNULL. In that case, the internal global variable zen_error the type of which is zen_err was set by this function with ZENSetError() (see the file sys.h). Here, the user only has to call the function ZENError() in its code to handle this error.

Most functions of the library can be used in any rings (only few of them can only apply in finite fields) even if they are not defined everywhere. Mathematical errors generally occures when a function returns ZEN_NO_INVERSE. That means that somewhere in the code, a function tried to inverse an element e modulo a non prime integer n or a non irreducible polynomial $P(X)$ and that the gcd of e with n or $P(X)$ is not 1. In that case, a factor of the corresponding modulo is put in ZENRingFact(Rg) where Rg is the current finite ring.

The situation can be sligtly more complicated when building several not prime or not irreducible finite rings. For instance, when you are first working in $R_0 = \mathbb{Z}/15\mathbb{Z}$ and then in $R_1 = R_0[T]/(T^2 + 4T + 3)$. On the first hand, a call to ZENEltInverse() with argument $2T + 6$ will return ZEN_NO_INVERSE and a call to ZENRingFact(R_1) will return $T + 3$. On the other hand, a call to ZENEltInverse() with argument $3T + 6$ will return ZEN_NO_INVERSE and a call to ZENRingFact(R_1) will return 3. Here, the degree of ZENRingFact(R_1) is 0 and so, we can call ZENRingFact(R_0) to finally get 3, a factor of 15, the modulo of the first finite ring.

The values of ZENERR and ZENNULL are defined as follows:

```
#define ZENERR -1
#define ZENNULL NULL
```

In fact, in case of error, the functions of ZEN return one of the following

```
# define ZERR _ZENERR()
# define ZNULL _ZENNULL()
```

This is intended to allow interactive debugging by giving the possibility of setting breakpoints in the functions _ZENERR and _ZENNULL. Therefore this feature is disabled in optimized compilation. In that case, most of the tests are skipped, except those that check if an inverse was impossible to compute.

In debugging mode, the flag debugflag can be set to non zero. This activate verbose error outputs. Each time one of the two functions _ZENERR() or _ZENNULL is called, an error message as printed by ZENError() is printed on

the standard error output, according to the value of zen_error The real structure of zen_error is as follows.

```
typedef struct zen_err{
  arith lib;
  int fct;
  int err;
} zen_err;
extern zen_err zen_error;
extern int debugflag;
```

The internal functions of ZEN raised errors with the function ZENSetError().

Procedure 1 *Setting the error condition flags*

```
void ZENSetError(arit,function,error)
   arith arit;
   int function,error;
```

Input: *The three flags describing the error exception:*

- *the arithmetic in which it occured,*
- *the function of* ZEN *in which it occured,*
- *the type of error.*

Side effect: *The* zen_error *global structure is set.*
Note: *This is not a user purpose function. It is described here to inform users about the way the errors are handled by the library.*

When OptimizingCode is set to NO in file specif.def, the following flag is set by default :

```
int debugflag=1;
```

This flag activates automatic printing of error messages whenever a system error occurs. This is useful for debugging small programs in which it is somehow pedantic to test all returned values.

Procedure 2 *Writing a message describing an error*

```
void ZENError ()
```

Side effect: *An error message is printed on the standard error output.*
Note: *The message depends on* zen_error.

Procedure 3 *Debugging function*

```
int _ZENERR()
```

Output: ZENERR
Side effect: *If* debugflag *is non zero, prints an error message.*
Note: *This function can be used to set breakpoint in a debugger. It is never called when the library is compiled in optimized mode.*

Procedure 4 *Debugging function*

```
void *_ZENNULL()
```

Output: ZENNULL *pointer.*
Side effect: *If* debugflag *is non zero, prints an error message.*
Note: *This function can be used to set breakpoint in a debugger. It is never called when the library is compiled in optimized mode.*

3.2 Optimization

In most cases, the above functions will be sufficient to obtain good performances. Nevertheless, some applications will need much more efficiency, and the following is a way to achieve quite good improvements.

3.2.1 Precomputations

Another principle of ZEN is to perform only what is asked for. Therefore, on a ring creation, no precomputation is done. But some precomputations can be asked for by using a precomputation structure ZENPrc Prc. For instance, ZENPrcSetAll(Prc) activates all the precomputations. A call to ZENRingAddPrc(R,Prc) will then perform the precomputations that are compatible with the type of ring. The available flags are described in section 4.7.1.

3.2.2 Clones

They are several way to represent a ring and this fact yields several computer representations. To deal with these numerous representations, the user can compute a "clone" of a ring with the function ZENRingClone(). Unfortunataly, only a few clones are now available.

3.2.2.1 Tabulating

A small finite ring can be cloned using the index representation. All the elements Z of a ZENRing R are represented in the clone C obtained from R by the result (n,nl) of ZENEltToZ(n,p_nl,Z,R). That is to say, each element of a ring is ordered by the integer value it takes once evaluated in the characteristic.

Addition, multiplication, negation and inversion are tabulated at the initialization. Therefore, all the subsequent operations will take constant time. The limit size is that of an unsigned char, that is to say 256 elements. Polynomials and matrices use also the same representation which saves memory .

3.2.2.2 Use of logarithms

A small finite field $\mathbb{F}$ can be cloned using the logarithm representation. The first operation performed is to find a generator α of the finite field. Then, a table of all the logarithms is computed. The adopted representation in ZEN is the following:

Element of $\mathbb{F}$	$\mapsto$	ZEN representation
0	$\mapsto$	0
1	$\mapsto$	1
α	$\mapsto$	2
α^i	$\mapsto$	$i+1$

Hence, multiplication and inversion are easily performed by a modular addition on the exponent, assuming that a first test of equality to zero is performed on each operand:

$$\begin{array}{rcl} \alpha^i \times \alpha^j & \mapsto & (i+1)+(j+1)-1 \\ (\alpha^i)^{-1} & \mapsto & -(i+1)+2 \end{array}$$

For negation, the table of this operation is computed at the initialization of the clone. For addition, another table is computed that stores all the exponent of each element incremented by one. Addition of two elements can then be performed by a multiplication using the formula

$$\alpha^i + \alpha^j = \alpha^i(1+\alpha^{j-i}).$$

The limit size is that of an unsigned short, that is to say at most 65536 elements. Polynomials and matrices use also the same representation which saves memory.

3.2.2.3 Use of chinese remainder theorem

A ZENRing can be built upon two ZENRings using the Chinese remainder theorem.

Theorem *Let m and n be two natural integers, m prime with n. The two rings $\mathbb{Z}/_{(mn)\mathbb{Z}}$ and $\mathbb{Z}/_{m\mathbb{Z}} \times \mathbb{Z}/_{n\mathbb{Z}}$ are isomorphic. More precisely, the application*

$$\begin{array}{rcccl} \theta & : & \mathbb{Z}/_{mn\mathbb{Z}} & \to & \mathbb{Z}/_{m\mathbb{Z}} \times \mathbb{Z}/_{n\mathbb{Z}} \\ & & x & \mapsto & (x \bmod m, x \bmod n) \end{array}$$

is isomorphic and its reciprocal is

$$\theta^{-1}(x_m, x_n) = x_m n(n^{-1} \bmod m) + x_n m(m^{-1} \bmod n) \bmod mn.$$

The same kind of result can be stated for polynomials.

The implementation of these results in ZEN is more general: one can use N ZENRings to build the two isomorphics ZENRings. The N ZENRings must be of same level (N modular rings, or N extensions over same ring). The representation of an element in such a ring, is the N-array of the N projections of this element in the N subrings.

The function ZENChineseRingCreate() performs such a construction.

3.2.2.4 Use of Montgomery's representation

Montgomery's idea is implemented in ZEN. The following description is largely inspired from [5, pages 133–135].

We now assume that we want to work in a modular ring $\mathbb{Z}/_{N\mathbb{Z}}$, with N an odd integer. Let R be an integer greater than N and prime with N. Then, the mapping

$$\begin{array}{rccc} \hat{\phi} & : & \mathbb{Z}/_{N\mathbb{Z}} & \rightarrow & \mathbb{Z}/_{N\mathbb{Z}} \\ & & x & \mapsto & \hat{x} = Rx \bmod N \end{array}$$

is invertible and we call $\hat{x}$ the N-rsidu of x.

It is always possible to find u and v two integers such that

$$Ru - vN = 1,$$

with

$$0 < u < N \text{ and } 0 < v < R.$$

Theorem Let z be an integer between 0 and RN, the following algorithm computes $zR^{-1} mod N$.

1. Compute $m = ((z \bmod R)v) \bmod R$.
2. Let $t = (z + mN)/R$.
3. If t is greater than N, subtract N.
4. Return t.

Proof We have

$$\begin{array}{rcl} m = zv \bmod R & \Leftrightarrow & mN = z(vN) \bmod R, \\ & \Leftrightarrow & mN = -z \bmod R, \\ & \Leftrightarrow & mN + z = \alpha R, \end{array}$$

with α an integer. Hence, t in step 2 is an integer. Furthermore, $tR = z \bmod N$. As $0 \leq z < RN$ and $0 \leq m < R$, we have, after step 2, $0 \leq t < 2N$. Hence, the returned value after step 3 is $zR^{-1} \bmod N$.

We denote $\hat{\phi}^{-1}$ the above procedure.

Size (bits)	128	256	512	768	1024
ZEN(standard ring)	0.027	0.13	0.76	2.27	5.0
ZEN(clone + precomputations)	0.013	0.08	0.53	1.65	3.8

Figure 3.3: modular exponentiation $a^b \bmod c$ on sparc II (times given in seconds)

Montgomery's idea is to replace modular oerations on integers by operations on N-rsidus. In this case we have

$$\hat{x}\hat{\times}\hat{y} = \hat{\phi}^{-1}(\hat{x} \times \hat{y}),$$

because

$$(xy)R = (xR)(yR)R^{-1}.$$

On the other hand, we have

$$\hat{(x^{-1})} = (\hat{x})^{-1}\hat{R},$$

because

$$(x^{-1})R = (xR)^{-1}R^2.$$

Implementation in ZEN In order to use Montgomery's representation, one has to use a clone. The implementation follows what precedes, except that the inversion is faster using the following procedure:

1. Convert the N-rsidu in the original modular ring.
2. Inversion in the original ring.
3. Re-conversion to obtain the N-rsidu.

Performances The table 3.3 shows the improvement obtained by Montgomery's reduction on the classical example of modular exponentiation. The observed gain is in the range 25 to 40 %.

Note Montgomery's idea applies only for odd modulus. For even ones, one has to use chinese remainder theorem to get rid of even factors.

Chapter 4

Enumerating the functions of ZEN

All the procedures described here form the libzen.a library. These procedures are all macros defined in zen.h. There are four types of such macros:

- Macros which are an interface for a field of a ZENRing Rg. The last argument of these macros is Rg.
- Macros which are an interface for an elliptic curve ZENEc E. The last argument of these macros is E.
- Macros which are an interface for a function of the library. Such functions are prefixed by _ZEN.
- Macros for basic operations, for instance getting a field of a structure described in section 3.1.1.

Note 3 The ZEN format is entirely made of macros. It is therefore strictly forbidden to use side effects in parameters of ZEN calls.

4.1 Procedures to handle finite rings.

A first set of procedures enables the user to allocate or free a finite ring.

4.1.1 Extension allocation.

The principle of the library is that efficient procedures are chosen on the creation of a ZENRing by setting a lot of pointers on functions inside the ZENRing structure. Therefore, the use of generic functions is possible together with efficiency as the function call only needs one more dereferenciation of function pointer.

Procedure 5 *Initialization of a ring* $\mathbb{Z}/n\mathbb{Z}$.

```
int ZENBaseRingAlloc(R, n, nl)
    ZENRing R;
    BigNum n;
    BigNumLength nl;
```

Input: *An unallocated* ZENRing, *and a* BigNum n *of size* nl.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: R *is allocated and set to* $\mathbb{Z}/n\mathbb{Z}$. *If* $n = 0$, R *is allocated and set to* $\mathbb{Q}$ *(experimental feature).*
Note: *No precomputations are done in order to speed up some arithmetic operations. If you want some, see* ZENRingAddPrc() *and* ZENRingRmPrc().

Procedure 6 *Initialization of a finite extension over another finite ring.*

```
int ZENExtRingAlloc(Ex, P, Rg)
    ZENPoly P;
    ZENRing Ex,Rg;
```

Input: *An unallocated* ZENRing, *and a polynomial* P *defined over a finite ring* Rg.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *The extension* Ex *is allocated and set to* $\mathsf{Rg}[X]/(\mathsf{P}(X))$.

Procedure 7 *Closing a finite ring.*

```
int ZENRingClose(Rg)
    ZENRing Rg;
```

Input: *A finite ring* Rg.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: Rg *is completely freed.*
Note: *A call to* ZENRingRmPrc(Rg, ZENRingPrcp(Rg)) *is performed at the beginning of this procedure.*

Procedure 8 *Closing a clone ring.*

```
int ZENRingCloneClose(R)
    ZENRing R;
```

Input: *A ring* R.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: R *is freed as well as the underlying* ZENRingOrigin(R), *if it exists. If* R *is a chinese ring, the list of rings is clone freed, but the under-underlying* ZENRingOrigin(R) *is not freed.*

Procedure 9 *Closing a finite ring.*

```
int ZENRingFullClose(Rg)
   ZENRing Rg;
```

Input: *A finite ring* Rg.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: Rg *is completely freed, as well as the underlying rings.*
Note: *This procedure can be used for freeing a* ZENRing *previously read from a file by* ZENRingReadFromFile.

Procedure 10 *Copying a ring.*

```
ZENRing ZENRingCopy(R)
   ZENRing R;
```

Input: *A finite field* R.
Output: ZENNULL *if an error occurred, a copy of the finite Ring* R *without ist precomputation otherwise.*

Procedure 11 *Writing to file to an internal representation.*

```
int ZENRingPrintToFile(file, Rg)
   FILE *file;
   ZENRing Rg;
```

Input: *A stream* file, *an allocated* ZENRing Rg.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Printing a representation of* Rg *to* file.
Note: *In an extension tower, one only needs to save the last ring as the structure is recursively saved on file.*

Procedure 12 *Reading from file.*

```
int ZENRingReadFromFile(Rg, file)
   FILE *file;
   ZENRing Rg;
```

Input: *A stream* file *and a pointer on a ring* Rg.
Output: *0 if no error occurred,* ZENERR *otherwise.*
Side effect: Rg *is created according to the datas in* file.

4.1.2 Ring parameters

Procedure 13 *The "characteristic".*

```
BigNum ZENRingP(Rg)
   ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: *The "characteristic" of* Rg.
Note: *This procedure is a macro which returns one field of* Rg. *You must not disallocate it.*

Procedure 14 *The size of the "characteristic".*

```
int ZENRingPl(Rg)
   ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: *The size of the "characteristic" of* Rg *in base* $2^{\mathsf{SIZE\_BLOC}}$.
Note: *This procedure is a macro which returns one field of* Rg.

Procedure 15 *The size of the elements.*

```
int ZENRingSizeElt(Rg)
   ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: *The size of the elements of* Rg *in base* $2^{\mathsf{SIZE\_BLOC}}$.
Note: *This procedure is a macro which returns one field of* Rg. *Usually it has the same value as* ZENRingPl(Rg) *except if the modulus is a power of* $2^{\mathsf{SIZE\_BLOC}}$. *In this case it is equal to* 1 *or* ZENRingPl(Rg) $-$ 1.

Procedure 16 *The number of elements of* Rg.

```
BigNum ZENRingQ(Rg)
   ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: *The number of elements of* Rg.
Note: *This procedure is a macro which returns one field of* Rg. *You must not disallocate it.*

Procedure 17 *The size of the number of elements.*

```
int ZENRingQl(Rg)
    ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: *The size of the number of elements of* Rg *in base* $2^{\text{SIZE\_BLOC}}$.
Note: *This procedure is a macro which returns one field of* Rg.

Procedure 18 *The finite subring.*

```
ZENRing ZENRingDef(Rg)
    ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: NULL *if* Rg *is* $\mathbb{Z}/n\mathbb{Z}$, *the finite ring* R *if* Rg *is a polynomial finite ring defined over* R.
Note: *This procedure is a macro which returns one field of* Rg. *You must not disallocate it.*

Procedure 19 *The definition polynomial.*

```
ZENPoly ZENRingPol(Rg)
    ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: NULL *if* Rg *is* $\mathbb{Z}/n\mathbb{Z}$, *the polynomial which defines* Rg *otherwise.*
Note: *This procedure is a macro which returns one field of* Rg. *You must not disallocate it.*

Procedure 20 *The degree of finite ring.*

```
int ZENRingDeg(Rg)
    ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: *0 if* Rg *is* $\mathbb{Z}/n\mathbb{Z}$, *the degree of the polynomial which defines* Rg *otherwise.*
Note: *This procedure is a macro which returns one field of* Rg.

Procedure 21 *The number of finite rings defined over a subfinite ring.*

```
int ZENRingExt(Rg)
   ZENRing Rg;
```

Input: *A* ZENRing Rg.
Output: *The number of finite rings defined over* Rg.
Note: *This procedure is a macro which returns one field of* Rg.

Procedure 22 *A factor of a modulo.*

```
ZENElt ZENRingFact(Rg)
   ZENRing Rg;
```

Input: *A finite ring* Rg.
Output: *A factor of a modulo which defines* Rg *or a subfinite ring of* Rg *if a function of the library returned* ZEN_NO_INVERSE.
Note: *This procedure is a macro which returns one field of* Rg. *You must not disallocate it. This field is set by operations such as* ZENEltInverse *when an inverse was impossible to compute. Therefore, a call to this function is pertinent only after a* ZEN_NO_INVERSE *return of such a function.*

4.2 Procedures to handle elements of finite rings

These procedures are current operations on elements of finite rings.

4.2.1 Allocation

Procedure 23 *Creation.*

```
int ZENEltAlloc(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *A finite ring* Rg *and a* ZENElt a.
Output: *0 if no error occurred,* ZENERR *otherwise.*
Side effect: a *is allocated.*

Procedure 24 *Freeing.*

```
void ZENEltFree(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *A* ZENElt a *of a finite ring* Rg *already allocated with* ZENEltAlloc.
Side effect: a *is freed.*

4.2.2 Assignment

Procedure 25 *Assigning.*

```
void ZENEltAssign(a, b, Rg)
   ZENElt a, b;
   ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Side effect: a *is filled with* b.

Procedure 26 *Setting to zero.*

```
void ZENEltSetToZero(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *An allocated element* a *of a finite ring* Rg.
Side effect: a *is set to zero.*

Procedure 27 *Setting to one.*

```
void ZENEltSetToOne(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *An allocated element* a *of a finite ring* Rg.
Side effect: a *is set to one.*

Procedure 28 *Setting to a generator.*

```
void ZENEltSetToGenerator(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *An allocated element* a *of a finite ring* Rg.
Side effect: a *is set to one if* Rg *is a prime field or set to* t *if* Rg $= R[t]/((P(t))$ *where* $P(t)$ *is a polynomial which defines an extension over a sub-ring* R.

Procedure 29 *Setting to random.*

```
void ZENEltSetRandom(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *An allocated element* a *of a finite ring* Rg.
Side effect: a *is set to random.*

Procedure 30 *Enumerating.*

```
void ZENEltSetNext(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *An allocated* ZENElt a *of a finite ring* Rg.
Side effect: a *is set to another element of* Rg.
Note: *This procedure can be seen more or less as* ZENEltToZ(n, nl, a, R)*;* ZBNAddCarry(n, nl, 1)*;* ZENZToElt(a, n, nl, R). *After a number of call of this function equal to the number of elements* ZENRingQ(Rg)*, one obtain the same element.*

Procedure 31 *Converting a* BigNum *to a* ZENElt

```
int ZENEltFromBigNum(e,n,nl,R)
   ZENElt e;
   BigNum n;
   BigNumLength nl;
   ZENRing R;
```

Input: *A* ZENElt*, a* BigNum *and its length and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *The big integer is first reduced modulo the characteristic of the field. The result is then assigned to* e *using* ZENZToElt.

Procedure 32 *Converting a* ZENElt *to a* ZENElt

```
int ZENEltConvert(e1,R1,e2,R2)
   ZENElt e1,e2;
   ZENRing R1,R2;
```

Input: *Two* ZENElt*s, and two* ZENRing*s.*
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *The element* e1 *of* R1 *is affected with* e2 *after possible modular reduction.* R1 *and* R2 *must be compatible rings.*

4.2.3 Test

Procedure 33 *Equality.*

```
int ZENEltAreEqual(a, b, Rg)
   ZENElt a, b;
   ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Output: *The predicate* a = b.

Procedure 34 *Is Zero.*

```
int ZENEltIsZero(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *An allocated element* a *of a finite ring* Rg.
Output: *The predicate* a = 0.

Procedure 35 *Is one.*

```
int ZENEltIsOne(a, Rg)
   ZENElt a;
   ZENRing Rg;
```

Input: *An allocated element* a *of a finite ring* Rg.
Output: *The predicate* a = 1.

Procedure 36 *Is a square in finite fields.*

```
int ZENEltIsASquare(a, Rg)
    ZENElt a;
    ZENRing Rg;
```

Input: *An element* a *of a finite ring* Rg.
Output: ZENERR *if an error occurred, 1 if* a *is a square, 0 otherwise.*
Note: *This procedure is valid only in finite fields. The algorithm used is an exponantiation when the characteristic is odd. In characteristic 2 the answer is always 1!*

4.2.4 Arithmetic

Procedure 37 *Addition.*

```
void ZENEltAdd(b, a, Rg)
    ZENElt b, a;
    ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Side effect: b += a.

Procedure 38 *Negation.*

```
void ZENEltNegate(b, a, Rg)
    ZENElt b, a;
    ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Side effect: b = −a.

Procedure 39 *Subtract.*

```
void ZENEltSubtract(b, a, Rg)
    ZENElt b, a;
    ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Side effect: b -= a.

Procedure 40 *Squaring.*

```
void ZENEltSquare(b, a, Rg)
   ZENElt b, a;
   ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Side effect: $b = a^2$.

Procedure 41 *Multiplication.*

```
void ZENEltMultiply(c, a, b, Rg)
   ZENElt c, a, b;
   ZENRing Rg;
```

Input: *Three allocated elements* a, b *and* c *of a finite ring* Rg.
Side effect: $c = a \times b$.
Note: *ACHTUNG !!! One must have* $c \neq a$. *A call to this function with the same element as parameter* c *and* a *can cause segmentation faults.*

Procedure 42 *Inverse.*

```
int ZENEltInverse(b, a, Rg)
   ZENElt b, a;
   ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: $b = 1/a$ *if* a *has an inverse,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*
Note: *One can have* b = a

Procedure 43 *Exponentiation.*

```
int ZENEltExp (R, k, kl, P, Rg)
   ZENElt R, P;
   BigNum k;
   BigNumLength kl;
   ZENRing Rg;
```

Input: *Two allocated elements* R *and* P *of a finite ring* Rg, *a* BigNum k *of size* kl.
Output: *0 if no error occured,* ZENERR *otherwise.*
Side effect: $R = P^k$.
Note: *The algorithm used is by default the binary method, but after precomputation, it is the m-ary method.*

Procedure 44 *"Trace".*

```
void ZENEltTrace(b, a, Rg)
   ZENElt b, a;
   ZENRing Rg;
```

Input: *Two allocated elements* a *and* b *of a finite ring* Rg.
Side effect: b = $Tr_{\mathsf{Rg}/R}(\mathsf{a})$ *where* R *is the base ring of* Rg. *If* Rg *is* $\mathbb{Z}/n\mathbb{Z}$ *where* n *is an integer,* b = a.

Procedure 45 *"Absolute Trace".*

```
int ZENEltAbsoluteTrace(e, f, Rg)
   ZENElt e, f;
   ZENRing Rg;
```

Input: *Two allocated elements* e *and* f *of a finite ring* Rg.
Output: ZENERR *if an error occurred,* ZEN_HAS_INVERSE *otherwise.*
Side effect: f = $Tr_{\mathsf{Rg}/(Z/pZ)}(\mathsf{e})$ *where* p *is the characteristic of* Rg. *If* Rg=$\mathbb{Z}/p\mathbb{Z}$, f = e.

.

Procedure 46 *Square roots in finite fields*

```
int ZENEltSquareRoot(R, P, Rg)
   ZENElt R, P;
   ZENRing Rg;
```

Input: *Two elements* R *and* P *of a finite field* Rg.
Output: ZENERR *if an error occurred, 0 if* P *is a square, 1 otherwise.*
Side effect: R *is filled with the square root of* P *if 0 is returned.*
Note: *This procedure is valid only in finite fields. The algorithm of Tonelli and Shanks is used.*

4.2.5 Input/Output

Procedure 47 *Converting from string.*

```
int ZENEltReadFromString(G, s, base, Rg)
   ZENElt G;
   char *s;
   int base;
   ZENRing Rg;
```

Input: *A finite ring* Rg*, an allocated element* G *and a string* s *representing a* ZENElt *in base* base $\in \{2, \ldots, 16\}$.
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: G *is filled with* s *if the output is not* ZENERR.

Procedure 48 *Converting to string.*

```
char *ZENEltPrintToString(G, base, Rg)
   ZENElt G;
   int base;
   ZENRing Rg;
```

Input: *An allocated element* G *of a finite ring* Rg *and a base* base $\in \{2, \ldots, 16\}$.
Output: *An allocated string representing* G *in base* base *or* ZENNULL *if an error occurred.*

Procedure 49 *Reading from file.*

```
int ZENEltReadFromFile(G, file, base, Rg)
   ZENElt G;
   FILE *file;
   int base;
   ZENRing Rg;
```

Input: *A stream* file*, an allocated element* G *of a finite ring* Rg *and a base* base $\in \{2, \ldots, 16\}$.
Output: *0 if no error occurred,* ZENERR *otherwise.*
Side effect: G *is filled with the* ZENElt *read in* file *if no error occurred.*

Procedure 50 *Printing to file.*

```
int ZENEltPrintToFile(file, G, base, Rg)
    FILE *file;
    ZENElt G;
    int base;
    ZENRing Rg;
```

Input: *A stream* file*, an allocated element* G *of a finite ring* Rg *and a base* base $\in \{2, \ldots, 16\}$.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* G *in base* base *to* file.

Procedure 51 *Converting from a string to an internal representation.*

```
int ZENEltGetFromString(G, s, Rg)
    char *s;
    ZENElt G;
    ZENRing Rg;
```

Input: *An allocated element* G *of a finite ring* Rg *and a string* s *representing a* ZENElt *to an internal representation.*
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: G *is filled with* s

Procedure 52 *Converting to string to an internal representation.*

```
char *ZENEltPutToString(G, Rg)
    ZENElt G;
    ZENRing Rg;
```

Input: *An allocated element* G *of a finite ring* Rg.
Output: *An allocated string representing* G *to an internal representation or* ZENNULL *if an error occurred.*

Procedure 53 *Getting from file to an internal representation.*

```
int ZENEltGetFromFile(G, file, Rg)
    ZENElt G;
    FILE *file;
    ZENRing Rg;
```

Input: *A stream* file*, and an element* G *of a finite ring* Rg.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: G *is filled with the element read in* file

Procedure 54 *Writing to file to an internal representation.*

```
int ZENEltPutToFile(file, G, Rg)
   FILE *file;
   ZENElt G;
   ZENRing Rg;
```

Input: *A stream* file*, an allocated element* G *of a finite ring* Rg.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* G *to* file.

Procedure 55 *Evaluating an element as a multivariate polynomial over Z.*

```
void ZENEltToZ(p, p_pl, G, Rg)
   BigNum p;
   BigNumLength *p_pl;
   ZENElt G;
   ZENRing Rg;
```

Input: *A Bignum* p *of allocated size greater or equal than* ZENRingQl(Rg)*, a pointer* p_pl *on the real size of* p*, an allocated element* G *of a finite ring* Rg.
Side effect: p *is filled with the value of* G *considered as a multivariate polynomial in which we substitute all the variables with the "characteristic",* ⋆p_pl *contains the real size of* p.

Procedure 56 *Getting an element from its evaluation as a multivariate polynomial over Z.*

```
void ZENZToElt(G, p, pl, Rg)
   BigNum p;
   BigNumLength pl;
   ZENElt G;
   ZENRing Rg;
```

Input: *A Bignum* p *of size* pl*, an allocated element* G *of a finite ring* Rg.
Side effect: G *is filled with the element whose evaluation considered as a multivariate polynomial in which we substitute all the variables with the "characteristic" is equal to (*p*,* pl*).*

Procedure 57 *Clone conversion.*

```
void ZENEltToClone(C, B, Rg)
   ZENElt C,B;
   ZENRing Rg;
```

Input: *A* ZENElt B *of the original ring of* Rg *and a* ZENElt C *of the clone* Rg.
Side effect: C *is filled with* B.

Procedure 58 *Clone conversion.*

```
void ZENCloneToElt(B, C, Rg)
   ZENElt B,C;
   ZENRing Rg;
```

Input: *A* ZENElt B *of the original ring of* Rg *and a* ZENElt C *of the clone* Rg.
Side effect: B *is filled with* C.

4.3 Procedures to handle polynomials over finite rings

These procedures are current operations on polynomials over finite rings.

4.3.1 Allocation

Procedure 59 *Creation.*

```
int ZENPolyAlloc(PX, deg, Rg)
   ZENPoly PX;
   int deg;
   ZENRing Rg;
```

Input: *An unallocated* ZENPoly, *a finite ring* Rg *and a degree* deg.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The polynomial is allocated together with its* deg*+1 coefficients.*
Note: *All the coefficients of the polynomial are NOT set to zero. Hence, to create and set a polynomial to* $a_{\mathsf{n}}X^{\mathsf{n}} + a_{\mathsf{n}-1}X^{\mathsf{n}-1} + \cdots + a_0$, *one will have to:*

1. *Create a polynomial with* ZENPolyAlloc(PX, n, Rg).
2. *Set it to* X^{n} *in order to initialize it with* ZENPolySetToXi(PX, n, Rg).
3. *Set all the non null coefficients* a_i, $0 \leq i \leq \mathsf{n}$ *using* ZENPolySetCoeff().

Procedure 60 *Degree.*

```
int ZENPolyDeg(PX, Rg)
   ZENPoly PX;
   ZENRing Rg;
```

Input: *An allocated polynomial* PX *of a finite ring* Rg.
Output: *The degree of* PX.
Note: *This procedure is a macro which returns one field of* PX. *You can assign* ZENPolyDeg(PX, Rg), *but beware that all functions assume this field correct.*

Procedure 61 *Length.*

```
int ZENPolyLgt(PX, Rg)
   ZENPoly PX;
   ZENRing Rg;
```

Input: *An allocated polynomial* PX *of a finite ring* Rg.
Output: *The maximal degree of* PX.
Note: *This procedure is a macro which returns one field of* PX. *Assigning* ZENPolyLgt(PX, Rg) *can lead to bugs.*

Procedure 62 *Copying an allocated polynomial.*

```
ZENPoly ZENPolyCopy (PX, Rg)
   ZENPoly PX;
   ZENRing Rg;
```

Input: *An allocated polynomial* PX *of a finite ring* Rg.
Output: ZENNULL *if an error occurred, a copy of* PX *otherwise.*

Procedure 63 *Freeing.*

```
void ZENPolyFree(PX,Rg)
   ZENPoly PX;
   ZENRing Rg;
```

Input: *An allocated polynomial* PX *of a finite ring* Rg.
Side effect: PX *is disallocated.*

4.3.2 Assigning

Procedure 64 *Assigning.*

```
void ZENPolyAssign(RX, PX, Rg)
   ZENPoly RX, PX;
   ZENRing Rg;
```

Input: *Two allocated polynomials* PX *and* RX *of a finite ring* Rg.
Side effect: RX *is filled with* PX.

Procedure 65 *Setting to zero.*

```
void ZENPolySetToZero(RX, Rg)
   ZENPoly RX;
   ZENRing Rg;
```

Input: *An allocated polynomial* RX *of a finite ring* Rg.
Side effect: RX *is set to zero.*

Procedure 66 *Setting to* X^i.

```
void ZENPolySetToXi(RX, deg, Rg)
   ZENPoly RX;
   int deg;
   ZENRing Rg;
```

Input: *An allocated polynomial* RX *of a finite ring* Rg *allocated at least for degree* deg.
Side effect: RX *is set to* X^{deg}.

Procedure 67 *Setting randomly.*

```
void ZENPolySetRandom(RX, deg, Rg)
   ZENPoly RX;
   int deg;
   ZENRing Rg;
```

Input: *An allocated polynomial* RX *of length at least* deg *of a finite ring* Rg.
Side effect: RX *is set randomly to a polynomial of degree* deg.

Procedure 68 *Setting a coefficient.*

```
void ZENPolySetCoeff(RX, d, b, Rg)
   ZENPoly RX;
   int d;
   ZENElt b;
   ZENRing Rg;
```

Input: *An initialized polynomial* RX *of degree greater or equal to* d *and an element* b *of a finite ring* Rg, *the degree* d *of the coefficient to set.*
Side effect: *The coefficient of* X^d *in* RX *is set to* b.
Note: *The degree of* RX *is NOT updated by this operation.*

Procedure 69 *Getting a coefficient.*

```
void ZENPolyGetCoeff(b, RX, d, Rg)
   ZENElt b;
   ZENPoly RX;
   int d;
   ZENRing Rg;
```

Input: *An allocated polynomial* RX *of a finite ring* Rg, *the degree* d *of the coefficient to get and an element* b *to assign.*
Side effect: b *is filled with the coefficient of* X^d *in* RX.

Procedure 70 *Extracting a coefficient.*

```
ZENElt ZENPolyGetCoeffPtr(RX, d, Rg)
    ZENPoly RX;
    int d;
    ZENRing Rg;
```

Input: *An allocated polynomial* RX *of a finite ring* Rg*, the degree* d *of the coefficient to get.*
Output: *The pointer to the coefficient of* X^d *in* RX*.*
Side effect: *You must not disallocate the output.*

Procedure 71 *Updating the degree of a polynomial.*

```
void ZENPolyUpdateDegree(RX, Rg)
    ZENPoly RX;
    ZENRing Rg;
```

Input: *An allocated polynomial* RX *of a finite ring* Rg*.*
Side effect: *Resetting the degree of* RX*, assuming it has decreased.*

Procedure 72 *Convert.*

```
int ZENPolyConvert(P1, R1, P2, R2)
    ZENPoly P1, P2;
    ZENRing R1, R2;
```

Input: *Two* ZENPoly*s and two* ZENRing*s.*
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *The coefficients of* P1 *of* R1 *are affected with coefficients of* P2 *of* R2 *after possible modular reduction of those coefficients.* R1 *and* R2 *must be compatible rings.*

4.3.3 Test

Procedure 73 *Equality.*

```
int ZENPolyAreEqual(RX, PX, Rg)
    ZENPoly RX, PX;
    ZENRing Rg;
```

Input: *Two allocated polynomials* RX *and* PX *of a finite ring* Rg*.*
Output: *The predicate* RX = PX*.*

Procedure 74 *Is zero.*

```
int ZENPolyIsZero(RX, Rg)
   ZENPoly RX;
   ZENRing Rg;
```

Input: *An allocated polynomial* RX *of a finite ring* Rg.
Output: *The predicate* RX $= 0$.

Procedure 75 *Is a polynomial equal to* X^i.

```
int ZENPolyIsXi (PX, deg, Rg)
   ZENPoly PX;
   int deg;
   ZENRing Rg;
```

Input: *An allocated polynomial* PX *of a finite ring* Rg *and a degree* deg.
Output: *The predicate* PX $= X^{\mathsf{deg}}$.

4.3.4 Arithmetic

Procedure 76 *Addition.*

```
void ZENPolyAdd(RX, PX, Rg)
   ZENPoly RX, PX;
   ZENRing Rg;
```

Input: *Two allocated polynomials* RX *and* PX *of a finite ring* Rg.
Side effect: RX+ = PX.

Procedure 77 *Negation.*

```
void ZENPolyNegate(RX, PX, Rg)
   ZENPoly RX, PX;
   ZENRing Rg;
```

Input: *Two allocated polynomials* RX *and* PX *of a finite ring* Rg.
Side effect: RX $= -$PX.

Procedure 78 *Subtraction.*

```
void ZENPolySubtract(RX, PX, Rg)
   ZENPoly RX, PX;
   ZENRing Rg;
```

Input: *Two allocated polynomials* RX *and* PX *of a finite ring* Rg.
Side effect: RX$-=$PX.

Procedure 79 *Squaring.*

```
int ZENPolySquare(RX, PX, Rg)
   ZENPoly RX, PX;
   ZENRing Rg;
```

Input: *Two allocated polynomial* RX *and* PX *of a finite ring* Rg.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: RX $=$ PX^2.

Procedure 80 *Multiplication.*

```
int ZENPolyMultiply(RX, PX, QX, Rg)
   ZENPoly RX, PX, QX;
   ZENRing Rg;
```

Input: *Three allocated polynomial* RX, PX *and* QX *of a finite ring* Rg.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: RX $=$ PX $\times$ QX.
Note: *One must have* RX $\neq$ PX. *The Karatsuba's algorithm is always used.*

Procedure 81 *Multiplication by a scalar.*

```
void ZENPolyMultiplyScalar(RX, PX, e, Rg)
   ZENPoly RX, PX;
   ZENElt e;
   ZENRing Rg;
```

Input: *Two polynomials* PX, RX *and an element* e *of a finite ring* Rg.
Note: RX $=$ ePX.

Procedure 82 *Scalar product.*

```
void ZENPolyDot(e, PX, QX, Rg)
   ZENElt e;
   ZENPoly PX, QX;
   ZENRing Rg;
```

Input: *Two polynomials* PX, RX *and an allocated element* e *of a finite ring* Rg.
Side effect: e $= \sum p_i q_i$ *where* p_i *and* q_i *are the coefficients of* PX *and* QX.

Procedure 83 *Evaluation.*

```
void ZENPolyEval(f, PX, e, Rg)
   ZENElt f, e;
   ZENPoly PX;
   ZENRing Rg;
```

Input: *A polynomial* PX *and two elements* f *and* e *of a finite ring* Rg.
Side effect: f = PX(e).

Procedure 84 *Make a polynomial monic.*

```
int ZENPolyMakeMonic (RX, PX, Rg)
   ZENPoly RX, PX;
   ZENRing Rg;
```

Input: *Two polynomials* PX *and* RX *of a finite ring* Rg.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: R(X) *is filled with* P(X) *divided by its highest coefficient.*

Procedure 85 *Divide.*

```
int ZENPolyDivide(RX, MX, PX, QX, Rg)
   ZENPoly RX, MX, PX, QX;
   ZENRing Rg;
```

Input: *Two polynomials* PX *and* QX *to divide in a finite ring* Rg. RX *will be the quotient and* MX *the remainder.*
Output: ZEN_HAS_INVERSE *if no error occurred,* ZEN_NO_INVERSE *if a factor of a modulo was discovered,* ZENERR *for an error*
Side effect: RX *is filled with the euclidian quotient of* PX *by* QX, MX *with the remainder of* PX *by* QX.

Procedure 86 *Gcd of 2 polynomials.*

```
int ZENPolyGcd (RX, PX, QX, Rg)
   ZENPoly RX, PX, QX;
   ZENRing Rg;
```

Input: *Three polynomials* PX, QX *and* RX *of a finite ring* Rg.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: $\mathsf{R}(X) = gcd(\mathsf{P}(X),\mathsf{Q}(X))$ *if* ZEN_HAS_INVERSE *is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*

Procedure 87 *Extended gcd of 2 polynomials.*

```
int ZENPolyExtGcd (IX0, BX0, AX0, Rg)
   ZENPoly IX0, AX0, BX0;
   ZENRing Rg;
```

Input: *Three polynomials* IX0, AX0 *and* BX0 *of a finite ring* Rg.
Output: ZENERR *if an error occurred, -2 if the gcd of* AX0 *and* BX0 *is not 1,* ZEN_NO_INVERSE *if a factor of a modulo was found,* ZEN_HAS_INVERSE *otherwise*
Side effect: IX0$= 1/$BX0 $\bmod$ AX0 *if* ZEN_HAS_INVERSE *is returned,* IX0 *is gcd(*AX0, BX0*) if -2 is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*

Procedure 88 *Resultant of 2 polynomials.*

```
int ZENPolyResultant(Res, A, B, Rg)
   ZENElt Res;
   ZENPoly A, B;
   ZENRing Rg;
```

Input: *Three polynomials* $\star$Res, A *and* B *of a finite ring* Rg.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: $\star\mathsf{Res}(X) = resultant(\mathsf{A}(X),\mathsf{B}(X))$ *if* ZEN_HAS_INVERSE *is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*
Note: *Computation of the resultant of two polynomials by the subresultant algorithm. From P. Gaudry.*

4.3.5 Input/Output

Procedure 89 *Converting from string.*

```
int ZENPolyReadFromString(PX, s, base, Rg)
   ZENPoly PX;
   char *s;
   int base;
   ZENRing Rg;
```

Input: *A finite ring* Rg*, an unallocated polynomial* PX *and a string* s *representing a polynomial in base* base$\in \{2, \ldots, 16\}$.
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: PX *is allocated and filled with* s *if the output is not* ZENERR
Note: *Maple's format is used. The biggest monomial must be at the beginning of the string*

Procedure 90 *Converting to string.*

```
char *ZENPolyPrintToString(PX, base, Rg)
   ZENPoly PX;
   int base;
   ZENRing Rg;
```

Input: *An allocated polynomial* PX *of a finite ring* Rg *and a base* base$\in \{2, \ldots, 16\}$.
Output: *An allocated string representing* PX *in base* base *or* ZENNULL *if an error occurred.*
Note: *Maple's format is used.*

Procedure 91 *Reading from file.*

```
int ZENPolyReadFromFile(PX, file, base, Rg)
   ZENPoly PX;
   FILE *file;
   int base;
   ZENRing Rg;
```

Input: *A stream* file *and an unallocated polynomial* PX *of a finite ring* Rg.
Output: *0 if no error occurred,* ZENERR *otherwise.*
Side effect: PX *is allocated and filled with the polynomial read in* file *if no error occurred.*
Note: *Maple's format is used. The biggest monomial must be at the beginning of the stream*

Procedure 92 *Printing to file.*

```
int ZENPolyPrintToFile(file, PX, base, Rg)
   FILE *file;
   ZENPoly PX;
   int base;
   ZENRing Rg;
```

Input: *A stream* file, *an allocated polynomial* PX *of a finite ring* Rg *and a base* base$\in \{2, \ldots, 16\}$.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* PX *in base* base *to* file.
Note: *Maple's format is used.*

Procedure 93 *Converting from a string to an internal representation.*

```
int ZENPolyGetFromString(PX, s, Rg)
   char *s;
   ZENPoly PX;
   ZENRing Rg;
```

Input: *An unallocated polynomial* PX *of a finite ring* Rg *and a string* s *representing a polynomial to an internal representation.*
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: PX *is allocated and filled with* s *if the output is different from* ZENERR

Procedure 94 *Converting to string to an internal representation.*

```
char *ZENPolyPutToString(PX, Rg)
   ZENPoly PX;
   ZENRing Rg;
```

Input: *An allocated polynomials* PX *of a finite ring* Rg.
Output: *An allocated string representing* PX *to an internal representation or* ZENNULL *if an error occurred.*

Procedure 95 *Getting from file to an internal representation.*

```
int ZENPolyGetFromFile(PX, file, Rg)
   ZENPoly PX;
   FILE *file;
   ZENRing Rg;
```

Input: *A stream* file*, and an unallocated polynomial* PX *of a finite ring* Rg*.*
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: PX *is filled with the polynomial read in* file

Procedure 96 *Writing to file to an internal representation.*

```
int ZENPolyPutToFile(file, PX, Rg)
   FILE *file;
   ZENPoly PX;
   ZENRing Rg;
```

Input: *A stream* file*, an allocated polynomial* PX *of a finite ring* Rg*.*
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* PX *to* file*.*

Procedure 97 *Compute the derivative of a polynomial*

```
int ZENPolyDerive(RX,PX,Rg)
   ZENPoly RX,PX;
   ZENRing Rg;
```

Input: *Two polynomial* RX *and* PX *defined over a finite ring* Rg*.*
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: $\mathsf{RX} = \mathsf{PX}'$*.*

Procedure 98 *Roots of a polynomial of degree 2 in finite fields*

```
int ZENPolyRootsDegree2(roots, P, Rg)
   ZENPoly roots, P;
   ZENRing Rg;
```

Input: *Two* ZENPoly P *and* roots *over a* ZENRing Rg.
Output:

ZENERR *if an error occured,*

ZEN_NO_INVERSE *if an inverse was impossible to compute,*

ZEN_HAS_INVERSE *if the polynomial has one root.*

Side effect: *The polynomial* roots *is set to a polynomial of degree -1, 0 or 1, the coefficients of which are the roots of* P.
Note: roots *must be allocated for at least degree 1. We must have* P $\neq$ roots. *Valid only in finite fields.*

Procedure 99 *Roots of* $X - X^p = \gamma$.

```
int ZENPolyRootsCanonical(roots, gamma, Rg)
   ZENPoly roots;
   ZENElt gamma;
   ZENRing Rg;
```

Input: *A* ZENPoly roots *over a* ZENRing Rg *and an element* a.
Output:

ZENERR *if an error occured,*

ZEN_NO_INVERSE *if an inverse was impossible to compute,*

ZEN_HAS_INVERSE *if the polynomial has one root.*

Side effect: *The polynomial* roots *is set to a polynomial of degree -1 or* $p =$*charateristic(*Rg*), the coefficients of which are the roots of* P.
Note: P *must be allocated for at least degree* $p - 1$. *We must have* P $\neq$ roots. *Valid only in finite fields.*

Procedure 100 *Getting a BigNum from a Polynomial.*

```
BigNum ZENPolyToZ(p_pl, P, Rg)
   BigNumLength *p_pl;
   ZENPoly P;
   ZENRing Rg;
```

Input: *A pointer on an integer* p_pl*, and a polynomial* P *of a* ZENRing Rg.
Output: *A* BigNum *of size* ⋆p_pl *if no error occured,* ZENNULL *otherwise.*

Procedure 101 *Getting a polynomial from a BigNum.*

```
ZENPoly ZENZToPoly(p, pl, Rg)
   BigNum p;
   BigNumLength pl;
   ZENRing Rg;
```

Input: *A* BigNum *(*p, pl*) and a finite Ring* Rg.
Output: ZENNULL *if an error occured, otherwise a polynomial which once, evaluated in* ZENRingQ(Rg)*, returns (*n, nl*).*

4.4 Procedures to handle matrices over finite rings.

4.4.1 Parameters of a matrix

Procedure 102 *Testing type of a matrix*

```
int ZENMatIsRowType(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: *1 if the matrix is of* Row *type, 0 if it is of* Col *type*
Note: *Permutation matrices can be used.*

Procedure 103 *Testing type of a matrix*

```
int ZENMatIsPermutation(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: *1 if the matrix is a permutation, 0 if it is a plain matrix*

Procedure 104 *Testing type of a matrix*

```
int ZENMatAreSameType(A,B,R)
   ZENMat A,B;
   ZENRing R;
```

Input: *Two* ZENMats *and a* ZENRing
Output: *1 if the matrix are of same type, 0 otherwise.*
Note: *This function checks also if the matrix is permutation or plain.*

Procedure 105 *Number of rows of a matrix*

```
Dim ZENMatNbRow(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: *The number of rows of the matrix*

Procedure 106 *Number of columns of a matrix*

```
Dim ZENMatNbCol(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: *The number of columns of the matrix*

4.4.2 Allocation

Procedure 107 *Allocation of a matrix*

```
int ZENMatAlloc(M,r,c,R)
   ZENMat M;
   Dim r,c;
   ZENRing R;
```

Input: *An unallocated matrix* M*, a number of rows* r*, a number of columns* c *greater than 0 and a* ZENRing*.*
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *Matrix* M *is allocatedd together with* r *rows and* c *columns.*
Note: *The type of the matrix is chosen to minimize the memory needed.*

Procedure 108 *Allocation of a matrix*

```
int ZENMatRowAlloc(M,r,c,R)
   ZENMat M;
   Dim r,c;
   ZENRing R;
```

Input: *An unallocated matrix* M*, a number of rows* r*, a number of columns* c *greater than 0 and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The matrix is allocated* ZENMat *with type* ZENMatTypeRow *with* r *rows and* c *columns.*

Procedure 109 *Allocation of a matrix*

```
int ZENMatColAlloc(M,r,c,R)
   ZENMat M;
   Dim r,c;
   ZENRing R;
```

Input: *An unallocated matrix* M*, a number of rows* r*, a number of columns* c *greater than 0 and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The matrix is allocated* ZENMat *with type* ZENMatTypeCol *with* r *rows and* c *columns.*

Procedure 110 *Allocation of a permutation matrix*

```
int ZENPermutationRowAlloc(PI,n,R)
   ZENMat PI;
   Dim n;
   ZENRing R;
```

Input: *An unallocate* ZENMat*, a size* r *greater than 0 and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *Matrix* PI *is allocated with type* ZENPermutationTypeRow *with* r *rows and* r *columns.*
Note: *The matrix is set to identity*

Procedure 111 *Allocation of a permutation matrix*

```
int ZENPermutationColAlloc(PI,n,R)
   ZENMat PI;
   Dim n;
   ZENRing R;
```

Input: *An unallocate* ZENMat*, a size* r *greater than 0 and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *Matrix* PI *is allocated with type* ZENPermutationTypeCol *with* r *rows and* r *columns.*
Note: *The matrix is set to identity*

Procedure 112 *Freeing a matrix*

```
void ZENMatFree(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Side effect: *The matrix is freed*
Note: *Permutation matrices can be used*

Procedure 113 *Copying a matrix*

```
ZENMat ZENMatCopy(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: *A plain* ZENMat *copied from* M *of same type, or* ZENNULL *if an error occured.*
Note: *Permutation matrix can be input, but the output is a plain matrix.*

4.4.3 Assigning

Procedure 114 *Assigning a matrix to another matrix*

```
int ZENMatAssign(A,B,R)
    ZENMat A,B;
    ZENRing R;
```

Input: *Two* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: A=B.
Note: B *can be a permutation matrix. In this case,* A *can be either a permutation matrix or a plain matrix. Otherwise, it must be a plain matrix.*

Procedure 115 *Assigning to zero*

```
int ZENMatSetToZero(M,R)
    ZENMat M;
    ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: M=*0.*
Note: M *must be a plain matrix.*

Procedure 116 *Assigning to identity*

```
int ZENMatSetToOne(M,R)
    ZENMat M;
    ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: $\mathsf{M}_{i,j} = \delta_{i,j}$, *with* δ *the Kronecker's symbol.*
Note: *Permutation matrix can be used.*

Procedure 117 *Assigning randomly*

```
int ZENMatSetRandom(M,R)
    ZENMat M;
    ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *The matrix is randomly set.*
Note: *Permutation matrix can be used.*

Procedure 118 *Assigning a coefficient*

```
int ZENMatSetCoeff(M,r,c,Z,R)
    ZENMat M;
    Dim r,c;
    ZENElt Z;
    ZENRing R;
```

Input: *A* ZENMat*, two indexes, a* ZENElt *and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: $\mathsf{M}_{\mathsf{r},\mathsf{c}} = Z$
Note: *Permutation matrix can be used.*

Procedure 119 *Getting a coefficient*

```
int ZENMatGetCoeff(Z,M,r,c,R)
    ZENElt Z;
    ZENMat M;
    Dim r,c;
    ZENRing R;
```

Input: *An allocated* ZENElt*, a* ZENMat*, two indexes and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: $Z = \mathsf{M}_{\mathsf{r},\mathsf{c}}$
Note: *Permutation matrix can be used.*

Procedure 120 *Assigning a submatrix*

```
int ZENMatSetSubMat(M,r,c,S,R)
    ZENMat M,S;
    Dim r,c;
    ZENRing R;
```

Input: *Two* ZENMat*, two indexes, a* ZENElt *and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *The matrix* S *is assigned at position* (r,c) *in matrix* M.
Note: *Permutation matrix cannot be used.*

Procedure 121 *Getting a submatrix*

```
int ZENMatGetSubMat(S,M,r,c,R)
    ZENMat M,S;
    Dim r,c;
    ZENRing R;
```

Input: *Two allocated* ZENMat*, two indexes and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *The submatrix of* M *at position* (r,c) *of size those of* S *is extracted and assigned to* S.
Note: *Permutation matrix cannot be used.*

Procedure 122 *Getting a coefficient pointer*

```
int ZENMatGetCoeffPtr(M,r,c,R)
    ZENMat M;
    Dim r,c;
    ZENRing R;
```

Input: *A* ZENMat*, two indexes and a* ZENRing.
Output: *A pointer on* $\mathsf{M}_{\mathsf{r,c}}$.
Note: *Do NOT use the result of this function to change the coefficient of the matrix. This may not work. Use* ZENMatSetCoeff *instead. In the same way, do NOT assign the result of this function directly (like* z = ZENMatGetCoeffPtr();*) because further assignments such as* ZENEltAssign(z, x, Rg) *may modify the initial coefficient of the matrix. Use* ZENMatGetCoeff *instead. This function is intended to do fast read-only access to the coefficients of a matrix.*

4.4.4 Permuting rows or columns

Procedure 123 *Permute rows*

```
int ZENMatPermuteRow(M,r1,r2,R)
    ZENMat M;
    Dim r1,r2;
    ZENRing R;
```

Input: *A* ZENMat*, two indexes and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Rows* r1 *and* r2 *are permuted.*
Note: ZENMatTypeRowPermutation *can be used.*

Procedure 124 *Permute columns*

```
int ZENMatPermuteCol(M,c1,c2,R)
   ZENMat M;
   Dim c1,c2;
   ZENRing R;
```

Input: *A* ZENMat*, two indexes and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Rows* c1 *and* c2 *are permuted.*
Note: ZENMatTypeColPermutation *can be used.*

4.4.5 Tests

Procedure 125 *Equality*

```
int ZENMatAreEqual(A,B,R)
   ZENMat A,B;
   ZENRing R;
```

Input: *Two* ZENMat *and a* ZENRing
Output: A == B.
Note: *Permutation matrices can be used.*

Procedure 126 *Equality to zero*

```
int ZENMatIsZero(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: M == 0

Procedure 127 *Equality to identity*

```
int ZENMatIsOne(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing
Output: M == *Identity.*
Note: *The matrix is checked to be square.*

4.4.6 Arithmetic

Procedure 128 *Addition*

```
int ZENMatAdd(A,B,R)
   ZENMat A,B;
   ZENRing R;
```

Input: *Two* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: A += B
Note: B *can be a permutation.*

Procedure 129 *Addition of a scalar*

```
int ZENMatAddScalar(A,b,R)
   ZENMat A;
   ZENElt b;
   ZENRing R;
```

Input: *A* ZENMat, *a* ZENElt *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *All the elements of* A *are incremented by* b
Note: A *cannot be a permutation.*

Procedure 130 *Negation*

```
int ZENMatNegate(A,B,R)
   ZENMat A,B;
   ZENRing R;
```

Input: *Two* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: A = −B.
Note: B *can be a permutation.*

Procedure 131 *Subtraction*

```
int ZENMatSubtract(A,B,R)
   ZENMat A,B;
   ZENRing R;
```

Input: *Two* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: A -= B
Note: B *can be a permutation.*

Procedure 132 *Multiplication*

```
int ZENMatMultiply(X,A,B,R)
    ZENMat X,A,B;
    ZENRing R;
```

Input: *Three* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: X = AB.
Note: A *can be a* ZENMatTypeRowPermutation *and* B *can be a* ZENMatTypeColPermutation. *One can have* A=B *but* X *must be distinct from* A *and* B.

Procedure 133 *Multiplication of a scalar*

```
int ZENMatMultiplyScalar(A,b,R)
    ZENMat A;
    ZENElt b;
    ZENRing R;
```

Input: *A* ZENMat, *a* ZENElt *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *All the elements of* A *are multiplied by* b
Note: A *cannot be a permutation.*

Procedure 134 *Multiplication*

```
int ZENMatMultiplyPlain(X,A,B,R)
    ZENMat X,A,B;
    ZENRing R;
```

Input: *Three* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: X = AB.
Note: *Permutation cannot be used. The classical algorithm is used.*

Procedure 135 *Multiplication*

```
int ZENMatWinograd(X,A,B,R)
    ZENRing R;
```

Input: *Three* ZENMat *and a* ZENRing
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: X = AB.
Note: *Permutation cannot be used. Winograd's algorithm is used.*

Procedure 136 *Gaussian elimination*

```
int ZENMatGaussPlain(D,S,P,p_rk,R)
   ZENMat D,S,P;
   Dim *p_rk;
   ZENRing R;
```

Input: *Three* ZENMat *a pointer on a dimension and a* ZENRing
Output: ZENERR *if an error occurred,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned,* ZENMAT_HAS_MAXIMAL_RANK *if the matrix is of maximal rank,* ZENMAT_HAS_KERNEL *otherwise.*
Side effect: D *is diagonalised.* S *and* P *are modified accordingly (see below).* ⋆p_rk *is set to the rank of the matrix* D.
Note:

- *If* D *is a* $r \times c$ ZENMat *of type* TypeRow, *then* S *must be a* $r \times r$ ZENMat *of type* TypeRow *and* P *a* $c \times c$ ZENMat *of type* TypeCol, *or a* TypeColPermutation ZENMat *of size* c. *Then,* D *is diagonalized on its rows and* $S^{-1}DP^{-1}$ *remains invariant.*
- *If* D *is a* $r \times c$ ZENMat *of type* TypeCol, *then* S *must be a* $c \times c$ ZENMat *of type* TypeCol *and* P *a* $r \times r$ ZENMat *of type* TypeRow, *or a* TypeRowPermutation ZENMat *of size* r. *Then,* D *is diagonalized on its columns and* $P^{-1}DS^{-1}$ *remains invariant.*

A tricky feature is also possible, but should be used with care: it is possible to change the ZENMatNbDigit(D,R) *field of* D *before calling a gaussian elimination procedure. The gaussian elimination will therefore be performed only on the specified number of digits, but the corresponding modifications are done on all the matrix, whose real size is known through* ZENMatNbBloc(D,R). *This allows partial trigonaliation if needed. Don't forget to reset the correct value after the call.*

Procedure 137 *Rank of a matrix*

```
int ZENMatRank(D,p_rk,R)
    ZENMat D;
    Dim *p_rk;
    ZENRing R;
```

Input: *A* ZENMat *a pointer on a dimension and a* ZENRing
Output: ZENERR *if an error occurred,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned,* ZENMAT_HAS_MAXIMAL_RANK *if the matrix is of maximal rank,* ZENMAT_HAS_KERNEL *otherwise.*
Side effect: ⋆p_rk *is set to the rank of the matrix* D.

Procedure 138 *Determinant of a matrix*

```
int ZENMatDet(D,det,R)
    ZENMat D;
    ZENElt det;
    ZENRing R;
```

Input: *A* ZENMat *an allocated* ZENElt *and a* ZENRing
Output: ZENERR *if an error occurred,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned,* ZENMAT_HAS_INVERSE *otherwise.*
Side effect: ⋆p_det *is set to the determinant of the matrix* D.

Procedure 139 *Inverse*

```
int ZENMatInverse(I,M,R)
    ZENMat I,M;
    ZENRing R;
```

Input: *Two* ZENMat *and a* ZENRing
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise.*
Side effect: $\mathsf{I} = \mathsf{M}^{-1}$. ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*
Note: I *and* M *must be different but can be permutations.*

Procedure 140 *Kernel*

```
int ZENMatKernel(K,M,R)
   ZENRing R;
```

Input: *An unallocated* ZENMat, *an allocated* ZENMat *and a* ZENRing
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise.*
Side effect: K *is allocated and set to a basis of the kernel of* M. *One has* $MK^t = 0$. ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*
Note: K *can be returned set to zero.*

Procedure 141 *Transpose*

```
void ZENMatTranspose(M,R)
   ZENMat M;
   ZENRing R;
```

Input: *A* ZENMat *and a* ZENRing.
Side effect: *The matrix is transposed*
Note: *Permutation matrix can be used*

4.4.7 Input/output

Procedure 142 *Printing to string*

```
char *ZENMatPrintToString(r,base,Rg)
   ZENMat r;
   int base;
   ZENRing Rg;
```

Input: *A* ZENMat, *an integer between 2 and 16 and a* ZENRing.
Output: ZENNULL *if an error occured, a string representing* r *in base* base *otherwise*
Note: *Permutation matrices will be written as plain matrices.*

Procedure 143 *Reading from string*

```
int ZENMatReadFromString(r,s,base,Rg)
    ZENMat r;
    char *s;
    int base;
    ZENRing Rg;
```

Input: *An unallocated* ZENMat*, a string, an integer between 2 and 16 and a* ZENRing.
Output: ZENERR *if an error occured, the number of read characters otherwise*
Side effect: *The matrix is allocated and read from string* s *in base* base.

Procedure 144 *Printing to file*

```
int ZENMatPrintToFile(file,r,base,Rg)
    FILE *file;
    ZENMat r;
    int base;
    ZENRing Rg;
```

Input: *A file, a* ZENMat*, an integer between 2 and 16 and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise*
Note: *Permutation matrices will be written as plain matrices.*

Procedure 145 *Reading from file*

```
int ZENMatReadFromFile(r,file,base,Rg)
    ZENMat r;
    FILE *file;
    int base;
    ZENRing Rg;
```

Input: *An unallocated* ZENMat*, a file, an integer between 2 and 16 and a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The matrix is allocated and read from file* file *in base* base.

Procedure 146 *Printing to string in internal format*

```
int ZENMatPutToString(r,Rg)
   ZENMat r;
   ZENRing Rg;
```

Input: *A* ZENMat*, and a* ZENRing*.*
Output: ZENNULL *if an error occured, a string representing* r *otherwise*
Note: *Permutation matrices cannot be used.*

Procedure 147 *Reading from string in internal format*

```
int ZENMatGetFromString(r,s,Rg)
   ZENMat r;
   char *s;
   ZENRing Rg;
```

Input: *An unallocated* ZENMat*, a string, and a* ZENRing*.*
Output: ZENERR *if an error occured, the number of read characters otherwise*
Side effect: *The matrix is allocated and read from string* s*.*

Procedure 148 *Printing to file in internal format*

```
int ZENMatPutToFile(file,r,Rg)
   FILE *file;
   ZENMat r;
   ZENRing Rg;
```

Input: *A file, a* ZENMat*, and a* ZENRing*.*
Output: ZENERR *if an error occured, 0 otherwise*
Note: *Permutation matrices cannot be used.*

Procedure 149 *Reading from file*

```
int ZENMatGetFromFile(r,file,Rg)
   ZENMat r;
   FILE *file;
   ZENRing Rg;
```

Input: *An unallocated* ZENMat*, a file, and a* ZENRing*.*
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The matrix is allocated and read from file* file*.*

4.4.8 Matrix conversion

The following functions are intended to allow conversion of matrices between an extension and its definition ZENRing.

Procedure 150 *Simple assignment*

```
int ZENMatConvert(M1,R1,M2,R2)
    ZENMat M1,M2;
    ZENRing R1,R2;
```

Input: *Two couples (*ZENMat, ZENRing*).*
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Matrix* M1 *is assigned with* M2.
Note: *The coefficients of* M2 *must be elements of* R1. *The two matrices must be of same sizes.*

Now consider the following example: given a vector defined over $\mathbb{F}_{q^n}$

$$m = \begin{pmatrix} a_{0,0} + a_{0,1}x + \cdots + a_{0,n-1}x^{n-1} \\ \vdots \\ a_{k-1,0} + a_{k-1,1}x + \cdots + a_{k-1,n-1}x^{n-1} \end{pmatrix}$$

we have two matrices over $\mathbb{F}_q$ that we can need to consider:

$$M1 = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ \vdots & \vdots & & \vdots \\ a_{k-1,0} & a_{k-1,1} & \cdots & a_{k-1,n-1} \end{pmatrix} \text{ and } M2 = \begin{pmatrix} a_{0,0} \\ a_{0,1} \\ \vdots \\ a_{0,n-1} \\ \vdots \\ a_{k-1,0} \\ a_{k-1,1} \\ \vdots \\ a_{k-1,n-1} \end{pmatrix}.$$

The following functions allow such conversion. The principle is that if the input matrix on, say $\mathbb{F}_q$, is of row type the conversion will give a $M1$ type matrix. Otherwise, it will be a $M2$ type matrix. Sizes of the matrices must be chosen accordingly.

Procedure 151 *Extension conversion*

```
int ZENVect2Mat(m,M,R)
    ZENMat M,m;
    ZENRing R;
```

Input: *Two matrices* ZENMat *and an extension* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Matrix* M *is assigned with* m.
Note: *If the* ZENRing *is not an extension, this is equivalent to* ZENMatAssign.

Procedure 152 *Extension conversion*

```
int ZENMat2Vect(M,m,R)
   ZENMat M,m;
   ZENRing R;
```

Input: *Two matrices* ZENMat *and an extension* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Matrix* m *is assigned with* M.
Note: *If the* ZENRing *is not an extension, this is equivalent to* ZENMatAssign.

4.5 Procedures to handle series over finite rings

These procedures are current operations on series over finite rings.

4.5.1 Allocation

Procedure 153 *Creation.*

```
int ZENSrAlloc(S,length, Rg)
   ZENSr S;
   int length;
   ZENRing Rg;
```

Input: *An unallocated series, a finite ring* Rg *and a length* length.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The series is allocated with EXACTLY* length *allocated coefficients.*
Note: *The series IS NOT set to zero.*

Procedure 154 *Degree.*

```
int ZENSrDeg(PX, Rg)
   ZENSr PX;
   ZENRing Rg;
```

Input: *An allocated series* PX *of a finite ring* Rg.
Output: *The degree of* PX.
Note: *This procedure is a macro which returns one field of* PX. *You can assign* ZENSrDeg(PX, Rg).

Procedure 155 *Valuation.*

```
int ZENSrVal(PX, Rg)
    ZENSr PX;
    ZENRing Rg;
```

Input: *An allocated series* PX *of a finite ring* Rg.
Output: *The valuation of* PX.
Note: *This procedure is a macro which returns one field of* PX. *You can assign* ZENSrVal(PX, Rg)

Procedure 156 *Length.*

```
BigNum ZENSrLgt(PX, Rg)
   ZENSr PX;
   ZENRing Rg;
```

Input: *An allocated series* PX *of a finite ring* Rg.
Output: *The maximal size of* PX, *ie the maximal difference* degree(PX)-valuation(PX).
Note: *This procedure is a macro which returns one field of* PX.

Procedure 157 *Copying an allocated series.*

```
ZENSr ZENSrCopy (PX, Rg)
   ZENSr PX;
   ZENRing Rg;
```

Input: *An allocated series* PX *of a finite ring* Rg.
Output: ZENNULL *if an error occurred, a copy of* PX *otherwise.*

Procedure 158 *Freeing.*

```
void ZENSrFree(PX,Rg)
   ZENSr PX;
   ZENRing Rg;
```

Input: *An allocated series* PX *of a finite ring* Rg.
Side effect: PX *is disallocated.*

4.5.2 Assigning

Procedure 159 *Assigning.*

```
void ZENSrAssign(RX, PX, Rg)
   ZENSr RX, PX;
   ZENRing Rg;
```

Input: *Two allocated series* PX *and* RX *of a finite ring* Rg.
Side effect: RX *is filled with* PX.

Procedure 160 *Setting to zero.*

```
void ZENSrSetToZero(RX, Rg)
   ZENSr RX;
   ZENRing Rg;
```

Input: *An allocated series* RX *of a finite ring* Rg.
Side effect: RX *is set to zero.*

Procedure 161 *Setting to random.*

```
void ZENSrSetRandom(RX, val, deg, Rg)
   ZENSr RX;
   int deg;
   ZENRing Rg;
```

Input: *An allocated series* RX*, a valuation* val*, a degree* deg *and a finite ring* Rg.
Side effect: RX *is set randomly to a series of valuation* val *and degree* deg.

Procedure 162 *Setting a coefficient.*

```
void ZENSrSetCoeff(RX, d, b, Rg)
   ZENSr RX;
   int d;
   ZENElt b;
   ZENRing Rg;
```

Input: *An allocated series* RX *and an element* b *of a finite ring* Rg*, the degree* d *of the coefficient to set.*
Side effect: *The coefficient of* X^d *in* RX *is set to* b.
Note: *The degree or valuation of* RX *is not recomputed.*

Procedure 163 *Getting a coefficient.*

```
void ZENSrGetCoeff(b, RX, d, Rg)
   ZENElt b;
   ZENSr RX;
   int d;
   ZENRing Rg;
```

Input: *An allocated series* RX *of a finite ring* Rg*, the degree* d *of the coefficient to get and an element* b *to assign.*
Side effect: b *is filled with the coefficient of* X^d *in* RX.

Procedure 164 *Extracting a coefficient.*

```
ZENElt ZENSrGetCoeffPtr(RX, d, Rg)
   ZENSr RX;
   int d;
   ZENRing Rg;
```

Input: *An allocated series* RX *of a finite ring* Rg*, the degree* d *of the coefficient to set.*
Output: *The pointer to the coefficient of* X^d *in* RX.
Side effect: *You must not disallocate the output.*

Procedure 165 *Updating valuation.*

```
void ZENSrUpdateValuation(RX, Rg)
   ZENSr RX;
   ZENRing Rg;
```

Input: *An allocated series* RX *of a finite ring* Rg.
Side effect: *The valuation is updated if it increased.*

4.5.3 Test

Procedure 166 *Equality.*

```
int ZENSrAreEqual(RX, PX, Rg)
   ZENSr RX, PX;
   ZENRing Rg;
```

Input: *Two allocated series* RX *and* PX *of a finite ring* Rg.
Output: *The predicate* RX = PX.

Procedure 167 *Is zero.*

```
int ZENSrIsZero(RX, Rg)
   ZENSr RX;
   ZENRing Rg;
```

Input: *An allocated series* RX *of a finite ring* Rg.
Output: *The predicate* RX = 0.

4.5.4 Arithmetic

Procedure 168 *Addition.*

```
void ZENSrAdd(RX, PX, Rg)
   ZENSr RX, PX;
   ZENRing Rg;
```

Input: *Two allocated series* RX *and* PX *of a finite ring* Rg.
Side effect: RX+ = PX.

Procedure 169 *Negation.*

```
void ZENSrNegate(RX, PX, Rg)
   ZENSr RX, PX;
   ZENRing Rg;
```

Input: *Two allocated series* RX *and* PX *of a finite ring* Rg.
Side effect: RX = −PX.

Procedure 170 *Subtraction.*

```
void ZENSrSubtract(RX, PX, Rg)
   ZENSr RX, PX;
   ZENRing Rg;
```

Input: *Two allocated series* RX *and* PX *of a finite ring* Rg.
Side effect: RX− = PX.

Procedure 171 *Squaring.*

```
int ZENSrSquare(RX, PX, Rg)
   ZENSr RX, PX;
   ZENRing Rg;
```

Input: *Two allocated series* RX *and* PX *of a finite ring* Rg.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: RX = PX^2.

Procedure 172 *Multiplication.*

```
int ZENSrMultiply(RX, PX, QX, Rg)
   ZENSr RX, PX, QX;
   ZENRing Rg;
```

Input: *Three allocated series* RX, PX *and* QX *of a finite ring* Rg.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: RX = PX × QX.
Note: *One must have* RX ≠ PX.

Procedure 173 *Multiplication by a scalar.*

```
void ZENSrMultiplyScalar(RX, PX, e, Rg)
   ZENSr RX, PX;
   ZENElt e;
   ZENRing Rg;
```

Input: *Two series* PX, RX *and an element* e *of a finite ring* Rg.
Side effect: RX = ePX.

Procedure 174 *Divide.*

```
int ZENSrDivide(RX, PX, QX, Rg)
   ZENSr RX, PX, QX;
   ZENRing Rg;
```

Input: *Two series* PX *and* QX *to divide in a finite ring* Rg. RX *will be the quotient.*
Output: ZEN_HAS_INVERSE *if no error occurred,* ZEN_NO_INVERSE *if a factor of a modulo was discovered,* ZENERR *for an error*
Note: *One must have* valuation(PX) ≥ valuation(QX) *and* RX ≠ QX. *Call eventually* ZENSrUpdateValuation(QX, Rg) *to get the real valuation of* QX

4.5.5 Input/Output

Procedure 175 *Converting from string.*

```
int ZENSrReadFromString(PX, s, base, Rg)
   ZENSr PX;
   char *s;
   int base;
   ZENRing Rg;
```

Input: *A finite ring* Rg, *an unallocated series* PX *and a string* s *representing a series in base* base$\in \{2, \ldots, 16\}$.
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: PX *is allocated and filled with* s *if the output is not* ZENERR.
Note: *Maple's format is used. The biggest monomial must be at the beginning of the string*

Procedure 176 *Converting to string.*

```
char *ZENSrPrintToString(PX, base, Rg)
   ZENSr PX;
   int base;
   ZENRing Rg;
```

Input: *An allocated series* PX *of a finite ring* Rg *and a base* base$\in \{2, \ldots, 16\}$.
Output: *An allocated string representing* PX *in base* base *or* ZENNULL *if an error occurred.*
Note: *Maple's format is used.*

Procedure 177 *Reading from file.*

```
int ZENSrReadFromFile(PX, file, base, Rg)
   ZENSr PX;
   FILE *file;
   int base;
   ZENRing Rg;
```

Input: *A stream* file *and an unallocated series* PX *of a finite ring* Rg.
Output: *0 if no error occurred,* ZENERR *otherwise.*
Side effect: PX *is allocated and filled with the series read in* file *if no error occurred.*
Note: *Maple's format is used. The biggest monomial must be at the beginning of the stream*

Procedure 178 *Printing to file.*

```
int ZENSrPrintToFile(file, PX, base, Rg)
   FILE *file;
   ZENSr PX;
   int base;
   ZENRing Rg;
```

Input: *A stream* file*, an allocated series* PX *of a finite ring* Rg *and a base* base$\in \{2, \ldots, 16\}$.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* PX *in base* base *to* file.
Note: *Maple's format is used.*

Procedure 179 *Converting from a string yo an internal representation.*

```
int ZENSrGetFromString(PX, s, Rg)
   char *s;
   ZENSr PX;
   ZENRing Rg;
```

Input: *An unallocated series* PX *of a finite ring* Rg *and a string* s *representing a series to an internal representation.*
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: PX *is allocated and filled with* s *if the output is different from* ZENERR

Procedure 180 *Converting to string to an internal representation.*

```
char *ZENSrPutToString(PX, Rg)
   ZENSr PX;
   ZENRing Rg;
```

Input: *An allocated series* PX *of a finite ring* Rg*.*
Output: *An allocated string representing* PX *to an internal representation or* ZENNULL *if an error occurred.*

Procedure 181 *Getting from file to an internal representation.*

```
int ZENSrGetFromFile(PX, file, Rg)
   ZENSr PX;
   FILE *file;
   ZENRing Rg;
```

Input: *A stream* file, *and an unallocated series* PX *of a finite ring* Rg.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: PX *is filled with the series read in* file

Procedure 182 *Writing to file to an internal representation.*

```
int ZENSrPutToFile(file, PX, Rg)
   FILE *file;
   ZENSr PX;
   ZENRing Rg;
```

Input: *A stream* file, *an allocated series* PX *of a finite ring* Rg.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* PX *to* file.

4.6 Procedures to handle elliptic curves.

These routines basically implement the group law defined on an elliptic curve. In zen, such an elliptic curve is given by its Weierstrass parametrization,

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6.$$

For efficieny, ZEN procedures do not work directly with these parameters but on an internal Weierstrass parametrization. This internal parametrization (isomorphic to the user parametrization) depends on the "characteristic" and on the invariant of the curve. This internal parametrization is:

Characteristic 2, invariant $= 0$: $Y^2 + a_3Y = X^3 + a_4X + a_6$.

Characteristic 2, invariant $\neq 0$: $Y^2 + XY = X^3 + a_4X + a_6$.

Characteristic 3, invariant $= 0$: $Y^2 = X^3 + a_4X + a_6$.

Characteristic 3, invariant $\neq 0$: $Y^2 = X^3 + a_2X^2 + a_6$.

Characteristic > 3: $Y^2 = X^3 + a_4X + a_6$.

The conversion between this internal parametrization and the user parametrization is done once at the initialization of the curve with ZENEcInitialize and each time an I/O operation is performed, for instance ZENEcPtSetX, ZENEcPtSetXY, ZENEcPrintToString,...

4.6.1 Elliptic curve

Procedure 183 *Allocation of an elliptic curve.*

```
int ZENEcAlloc(E,Rg)
   ZENEc E;
   ZENRing Rg;
```

Input: *An unallocated elliptic curve and a finite ring* Rg.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The elliptic curve is allocated.*

Procedure 184 *Initialization of an elliptic curve.*

```
int ZENEcInitialize(E,a1,a2,a3,a4,a6)
   ZENEc E;
   ZENElt a1, a2, a3, a4, a6;
```

Input: *Five parameters* a1,a2, a3, a4, a6 *of an elliptic curve the equation of which is* $Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a4_X + a6$.
Output: ZEN_HAS_INVERSE *if the curve was succesfully initialized,* ZEN_NO_INVERSE *if a factor of a modulo was found,* ZENERR *otherwise.*
Side effect: *The curve* E *is initialized. Its internal parametrization is computed.*

Procedure 185 *Disallocating an elliptic curve*

```
void ZENEcFree(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Side effect: E *is freed.*

Procedure 186 a_1

```
ZENElt ZENEcA1(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Output: *The coefficient* a1 *of the curve.*
Note: *This procedure is a macro, do not disallocate its output.*

Procedure 187 a_2

```
ZENElt ZENEcA2(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Output: *The coefficient* a2 *of the curve.*
Note: *This procedure is a macro, do not disallocate its output.*

Procedure 188 a_3

```
ZENElt ZENEcA3(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Output: *The coefficient* a3 *of the curve.*
Note: *This procedure is a macro, do not disallocate its output.*

Procedure 189 a_4

```
ZENElt ZENEcA4(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Output: *The coefficient* a4 *of the curve.*
Note: *This procedure is a macro, do not disallocate its output.*

Procedure 190 a_6

```
ZENElt ZENEcA6(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Output: *The coefficient* a6 *of the curve.*
Note: *This procedure is a macro, do not disallocate its output.*

Procedure 191 *The discriminant.*

```
ZENElt ZENEcD(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Output: *The discrminant of the curve.*
Note: *This procedure is a macro, do not disallocate its output.*

Procedure 192 *The invariant.*

```
ZENElt ZENEcJ(E)
   ZENEc E;
```

Input: *An elliptic curve* E.
Output: *The invariant of the curve.*
Note: *This procedure is a macro, do not disallocate its output.*

4.6.2 Allocation

Procedure 193 *Point allocation*

```
int ZENEcPtAlloc(Pt,E)
   ZENEcPt Pt;
   ZENEc E;
```

Input: *An unallocated* ZENEcPt *and an elliptic curve* E.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The point is allocated.*

Procedure 194 *Point freeing*

```
void ZENEcPtFree(P, E)
   ZENEcPt P;
   ZENEc E;
```

Input: *An allocated point* P *of an elliptic curve* E.
Side effect: P *is disallocated.*

4.6.3 Assigning

Procedure 195 *Setting a point to zero*

```
void ZENEcPtSetToZero(P,E)
   ZENEcPt P;
   ZENEc E;
```

Input: *An allocated point* P *of an elliptic curve* E.
Side effect: P *is set to the identity element of the curve.*

Procedure 196 *Enumerating points on a curve.*

```
int ZENEcPtSetNext(P, Q, E)
    ZENEcPt P;
    ZENEcPt Q;
    ZENEc E;
```

Input: *Two points* P *and* Q *of a curve* E.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse of a modulo is found ,* ZEN_HAS_INVERSE *otherwise.*
Side effect: P *is assigned with a point whose abcissa is obtained from the abcissa of* Q *by* ZENEltSetNext.
Note: *Valid only in finite fields. This procedure loops for ever if there is no point on the curve !*

Procedure 197 *Assigning a point with another point.*

```
void ZENEcPtAssign(R, P, E)
    ZENEcPt R;
    ZENEcPt P;
    ZENEc E;
```

Input: *Two allocated points* P *and* Q *of an elliptic curve* E.
Side effect: P *is filled with* Q.

Procedure 198 *Getting the abcissa.*

```
void ZENEcPtGetX(a, P, E)
    ZENElt a;
    ZENEcPt P;
    ZENEc E;
```

Input: *An allocated point* P *of an elliptic curve* E *and an element* a.
Side effect: a *is filled with the abscissa of* P.

Procedure 199 *Getting the ordinate.*

```
void ZENEcPtGetY(a, P, E)
    ZENElt a;
    ZENEcPt P;
    ZENEc E;
```

Input: *An allocated point* P *of an elliptic curve* E *and an element* a.
Side effect: a *is filled with the abscissa of* P.

Procedure 200 *Setting the abcissa and ordinate of a point.*

```
void ZENEcPtSetXY(P,x,y,E)
   ZENEcPt P;
   ZENElt x,y;
   ZENEc E;
```

Input: *An allocated point* P *of an elliptic curve* E *and two elements* x, y.
Output: *The abcissa and ordiante of* P *are filled respectively with* x *and* y.

4.6.4 Tests

Procedure 201 *Is a point equal to another point ?*

```
int ZENEcPtAreEqual(P, Q, E)
   ZENEcPt P, Q;
   ZENEc E;
```

Input: *Two allocated points* P *and* Q *of an elliptic curve* E.
Output: *The predicate* $\mathsf{P} = \mathsf{Q}$

Procedure 202 *Is a point on the curve ?*

```
int ZENEcPtIsOnEc(P, E)
   ZENEcPt P;
   ZENEc E;
```

Input: *An allocated point* P *of an elliptic curve* E.
Output: *The predicate* $\mathsf{P} \in \mathsf{E}$

Procedure 203 *Is a point equal to zero ?*

```
int ZENEcPtIsZero(P, E)
   ZENEcPt P;
   ZENEc E;
```

Input: *An allocated point* P *of an elliptic curve* E.
Output: *The predicate* $\mathsf{P} = 0$

4.6.5 Arithmetic

Procedure 204 *Addition.*

```
int ZENEcPtAdd(R, P, Q, E)
   ZENEcPt R;
   ZENEcPt P;
   ZENEcPt Q;
   ZENEc E;
```

Input: *Three allocated points* P, Q *and* R *of an elliptic curve* E.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: R = P + Q *if* ZEN_HAS_INVERSE *is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*
Note: *We must have* $R \neq P$

Procedure 205 *Subtraction.*

```
int ZENEcPtSubtract(R, P, Q, E)
   ZENEcPt R;
   ZENEcPt P;
   ZENEcPt Q;
   ZENEc E;
```

Input: *Three allocated points* P, Q *and* R *of an elliptic curve* E.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: R = P − Q *if* ZEN_HAS_INVERSE *is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*
Note: *We must have* $R \neq P$

Procedure 206 *Doubling.*

```
int ZENEcPtDouble(R, P, E)
   ZENEcPt R;
   ZENEcPt P;
   ZENEc E;
```

Input: *Two allocated points* P *and* R *of an elliptic curve* E.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: R = 2P *if* ZEN_HAS_INVERSE *is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*

Procedure 207 *Opposite.*

```
void ZENEcPtNegate(R, P, E)
   ZENEcPt R;
   ZENEcPt P;
   ZENEc E;
```

Input: *Two allocated points* P *and* R *of an elliptic curve* E.
Side effect: R = −P *if* ZEN_HAS_INVERSE *is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*

Procedure 208 *Multiplication.*

```
int ZENEcPtMult(R, k, kl, P, E)
   ZENEcPt R;
   BigNum k;
   BigNumLength kl;
   ZENEcPt P;
   ZENEc E;
```

Input: *Two allocated points* P *and* R *of an elliptic curve* E *and a large integer* k *of size* kl.
Output: ZENERR *if an error occurred,* ZEN_NO_INVERSE *if an inverse was impossible to compute,* ZEN_HAS_INVERSE *otherwise*
Side effect: R = kP *if* ZEN_HAS_INVERSE *is returned,* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*

Procedure 209 *Adding several points at the same time on elliptic curves.*

```
int ZENMEcPtAdd(p_R, p_P, p_Q, p_E, type, nb, Rg)
   ZENEcPt *p_R, *p_P, *p_Q;
   ZENEc *p_E;
   char *type;
   int nb;
   ZENRing Rg;
```

Input: *2 arrays of* nb *points to add on different curves. 3 different additions are possible following* type*:*

- type[i]=MEc_No*: Nothing is done.*
- type[i]=MEc_Add*:* ⋆p_P *and* ⋆p_Q *are added.*
- type[i]=MEc_Double*:* ⋆p_P *is doubled.*

Output: *-1 if an error occured,* ZEN_NO_INVERSE *if a factor is found,* ZEN_HAS_INVERSE *otherwise.*
Side effect: *The result is put in* ⋆p_R*.*

Procedure 210 *Multiplying several points by integers at the same time on elliptic curves.*

```
int ZENMEcPtMult(p_R, p_k, p_kl, p_P, p_E, nb, Rg)
   ZENEcPt *p_R, *p_P;
   BigNum *p_k;
   BigNumLength *p_kl;
   ZENEc *p_E;
   int nb;
   ZENRing Rg;
```

Input: *An array of points* p_P *and an array of BigNum (*p_k*,* p_kl*).*
Output: *-1 if an error occured,* ZEN_NO_INVERSE *if a factor is found,* ZEN_HAS_INVERSE *otherwise.*
Side effect: *The products of* p_P *by the BigNums* p_k *are put in* p_R*.*

4.6.6 Input/Output

Procedure 211 *Converting from string.*

```
int ZENEcPtReadFromString(R, s, base, E)
   ZENEcPt R;
   char *s;
   int base;
   ZENEc E;
```

Input: *An elliptic curve* E*, an allocated point* R *and a string* s *representing a point in base* base $\in \{2, \ldots, 16\}$.
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: R *is filled with* s *if the output is not* ZENERR.

Procedure 212 *Converting to string.*

```
char* ZENEcPtPrintToString(R, base, E)
   ZENEcPt R;
   int base;
   ZENEc E;
```

Input: *An allocated point* R *of an elliptic curve* E *and a base* base $\in \{2, \ldots, 16\}$.
Output: *An allocated string representing* R *in base* base *or* ZENNULL *if an error occurred.*

Procedure 213 *Reading from file.*

```
int ZENEcPtReadFromFile(R, file, base, E)
   ZENEcPt R;
   FILE *file;
   int base;
   ZENEc E;
```

Input: *A stream* file*, an allocated point* R *of an elliptic curve* E *and a base* base $\in \{2, \ldots, 16\}$.
Output: *0 if no error occurred,* ZENERR *otherwise.*
Side effect: R *is filled with the point read in* file *if no error occurred.*

Procedure 214 *Printing to file.*

```
int ZENEcPtPrintToFile(file, R, base, E)
   FILE *file;
   ZENEcPt R;
   int base;
   ZENEc E;
```

Input: *A stream* file, *an allocated point* R *of an elliptic curve* E *and a base* base $\in \{2, \ldots, 16\}$.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* R *in base* base *to* file.

Procedure 215 *Converting from a string to an internal representation.*

```
int ZENEcPtGetFromString(R, s, E)
   ZENEcPt R;
   char *s;
   ZENEc E;
```

Input: *An allocated point* R *of an elliptic curve* E *and a string* s *representing a point to an internal representation.*
Output: *The number of character read in* s *or* ZENERR *if an error occurred.*
Side effect: R *is filled with* s *if the output is different from* ZENNULL

Procedure 216 *Converting to string to an internal representation.*

```
char* ZENEcPtPutToString(R, E)
   ZENEcPt R;
   ZENEc E;
```

Input: *An allocated point* R *of an elliptic curve* E.
Output: *An allocated string representing* R *to an internal representation or* ZENNULL *if an error occurred.*

Procedure 217 *Getting from file to an internal representation.*

```
int ZENEcPtGetFromFile(R, file, E)
   ZENEcPt R;
   FILE *file;
   ZENEc E;
```

Input: *A stream* file, *and an point* R *of an elliptic curve* E.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: R *is filled with the point read in* file

Procedure 218 *Writing to file to an internal representation.*

```
int ZENEcPtPutToFile(file, R, E)
   FILE *file;
   ZENEcPt R;
   ZENEc E;
```

Input: *A stream* file, *an allocated point* R *of an elliptic curve* E.
Output: ZENERR *if an error occurred, 0 otherwise.*
Side effect: *Printing a representation of* R *to* file.

4.7 Optimizations

4.7.1 Precomputations

Precomputations needed to speed up procedures must be set in a ZENPrc structure before giving it to ZENRingAddPrc() and ZENRingRmPrc().

The available flags are:

ZENPRC_FINITE_FIELD: Use some optimizations specific to finite fields. This flag is used only to indicate that the ring is a finite field. If set, some of the following precomputation flags performs better optimizations.

ZENPRC_ELT_MULTIPLY: Speeds up multiplications in finite fields. In modular rings with large modulus try to use Karatsuba's divide and conqueer algorithm. In an extension, try to use Newton's method based on the Karatsuba's algorithm to compute modulos.

ZENPRC_ELT_EXP: Speeds up ZENEltExp(). Provides reduction of the exponent modulo the cardinality minus one in finite fields (ZENPRC_FINITE_FIELD must be set).

ZENPRC_TRACE: Speeds up traces in finite fields (ZENPRC_FINITE_FIELD must be set).

ZENPRC_POLY_ROOTS_CANONICAL: Speeds up the procedure PolyRootsCanonical() in finite fields (ZENPRC_FINITE_FIELD must be set).

Procedure 219 *Setting all the precomputation flags.*

```
void ZENPrcSetAll(Prc)
   ZENPrc Prc;
```

Input: *A precomputation structure* Prc.
Side effect: *All the precomputation flags are set in* Prc *EXCEPT the* ZENPRC_FINITE_FIELD *flag.*

Procedure 220 *Setting no precomputation flag.*

```
void ZENPrcSetNone(Prc)
   ZENPrc Prc;
```

Input: *A precomputation structure* Prc.
Side effect: *No precomputation flag is set in* Prc.

Procedure 221 *Adding a precomputation flag.*

```
void ZENPrcSet(Prc, prp)
   ZENPrc Prc;
   int prp;
```

Input: *A precomputation flag* prp *and a structure* Prc.
Side effect: *The precomputation flag* prp *is set in* Prc.

Procedure 222 *Removing a precomputation flag.*

```
void ZENPrcUnset(Prc, prp)
   ZENPrc Prc;
   int prp;
```

Input: *A precomputation flag* prp *and a structure* Prc.
Side effect: *The precomputation flag* prp *is suppressed from* Prc.

Procedure 223 *Is a precomputation flag already set.*

```
int ZENPrcIsSet(Prc, prp)
   ZENPrc Prc;
   int prp;
```

Input: *A precomputation flag* prp *and a structure* Prc.
Output: *1 if* prp *is in* Prc, *0 otherwise.*

Procedure 224 *Assigning a precomputation flag in another flag.*

```
void ZENPrcAssign(prc1, prc2)
   ZENPrc prc1, prc2;
```

Input: *Two precomputation structures* prc1 *and* prc2.
Side effect: prc2 *is equal to* prc1.

Procedure 225 *Intersection of 2 precomputation flags.*

```
void ZENPrcAnd(prc1, prc2)
   ZENPrc prc1, prc2;
```

Input: *Two precomputation structures* prc1, prc2.
Side effect: prc2& = prc1.

Procedure 226 *Union of 2 precomputation flags.*

```
void ZENPrcOr(prc1, prc2)
   ZENPrc prc1, prc2;
```

Input: *Two precomputation structures* prc1, prc2.
Side effect: prc2| = prc1.

Procedure 227 *Negation of a precomputation flag.*

```
void ZENPrcNeg(prc)
   ZENPrc prc;
```

Input: *A precomputation structure* prc.
Side effect: prc = ~prc.

Procedure 228 *Precomputing.*

```
int ZENRingAddPrc(Rg, Prc)
   ZENRing Rg;
   ZENPrc Prc;
```

Input: *A finite ring* Rg *and precomputations* Prc.
Output: ZENERR *if an error occured,* ZEN_NO_INVERSE *if a factor of a modulo was discovered,* ZEN_HAS_INVERSE *otherwise.*
Side effect: *Precomputations asked in the flags of* Prc *are done in order to speed up the corresponding procedures.* ZENRingFact(Rg) *is filled with a factor of a modulo if* ZEN_NO_INVERSE *is returned.*
Note: *Precomputations can take time...*

Procedure 229 *Suppressing precomputations.*

```
void ZENRingRmPrc(Rg, Prc)
   ZENRing Rg;
   ZENPrc Prc;
```

Input: *A finite ring* Rg *and precomputations* Prc.
Side effect: *Structures already allocated by* ZENRingAddPrc() *are disallocated.*

Procedure 230 *The precomputations of* Rg.

```
ZENPrc ZENRingPrcp(Rg)
   ZENRing Rg;
```

Input: *A finite ring* Rg.
Output: *The precomputations already done in* Rg.
Note: *This procedure is a macro which returns one field of* Rg.

4.7.2 Clones

Type of clones must be set in a ZENCln structure before giving it to ZENRingClone().

The available flags are:

ZENCLN_LOG: Use a generator to store results of operations in tables. This is possible when the number of elements in the finite field is smaller than 2^{16}. This overrides ZENCLN_TABULATE and ZENCLN_MONTGOMERY when several flags are set.

ZENCLN_TABULATE: Store results of operations in tables whenever it's possible, mainly when the number of elements in the ring is smaller than 2^8.

ZENCLN_MONTGOMERY: In modular rings, use Montgomery's reduction.

Procedure 231 *Setting all the clone flags.*

```
void ZENClnSetAll(Cln)
   ZENCln Cln;
```

Input: *A clone structure* Cln.
Side effect: *All the clone flags are set in* Cln.

Procedure 232 *Setting no clone flag.*

```
void ZENClnSetNone(Cln)
   ZENCln Cln;
```

Input: *A clone structure* Cln.
Side effect: *No clone flag is set in* Cln.

Procedure 233 *Adding a clone flag.*

```
void ZENClnSet(Cln, prp)
    ZENCln Cln;
    int prp;
```

Input: *A clone flag* prp *and a structure* Cln.
Side effect: *The clone flag* prp *is set in* Cln.

Procedure 234 *Removing a clone flag.*

```
void ZENClnUnset(Cln, prp)
   ZENCln Cln;
   int prp;
```

Input: *A clone flag* prp *and a structure* Cln.
Side effect: *The clone flag* prp *is suppressed from* Cln.

Procedure 235 *Is a clone flag already set.*

```
int ZENClnIsSet(Cln, prp)
    ZENCln Cln;
    int prp;
```

Input: *A clone flag* prp *and a structure* Cln.
Output: *1 if* prp *is in* Cln*, 0 otherwise.*

Procedure 236 *Removing a clone flag.*

```
void ZENClnUnset(Cln, prp)
   ZENCln Cln;
   int prp;
```

Input: *A clone flag* prp *and a structure* Cln.
Side effect: *The clone flag* prp *is suppressed from* Cln.

Procedure 237 *Assigning a clone flag in another flag.*

```
void ZENClnAssign(cln1, cln2)
   ZENCln cln1, cln2;
```

Input: *Two clone structures* cln1 *and* cln2.
Side effect: cln2 *is equal to* cln1.

Procedure 238 *Intersection of 2 clone flags.*

```
void ZENClnAnd(cln1, cln2)
   ZENCln cln1, cln2;
```

Input: *Two clone structures* cln1, cln2.
Side effect: cln2& = cln1.

Procedure 239 *Union of 2 clone flags.*

```
void ZENClnOr(cln1, cln2)
   ZENCln cln1, cln2;
```

Input: *Two clone structures* cln1, cln2.
Side effect: cln2| = cln1.

Procedure 240 *Negation of a clone flag.*

```
void ZENClnNeg(cln)
   ZENCln cln;
```

Input: *A clone structure* cln.
Side effect: cln = ~cln.

Procedure 241 *Clone of a ring.*

```
ZENRing ZENRingClone(Rg, cln)
   ZENRing Rg;
   ZENCln cln;
```

Input: *A* ZENRing Rg *and the type of the clone* cln.
Output: *A ring* R *if everything was succesfully initialized,* ZENNULL *otherwise.*
Note: *Cloning can take time. If no flag is set in* cln, *a call to* ZENRingCopy() *is performed. If several flags are set, this procedure can try to make "good choices".*

In fact, two rings are allocated, the returned one and its ZENRingOrigin(). For instance, if PR[0] is $\mathbb{Z}/_{3\mathbb{Z}}$ and PR[1] is $\mathbb{Z}/_{5\mathbb{Z}}$, then ZENRingChinese(R,2,PR) will affect to R $\mathbb{Z}/_{3\mathbb{Z}} \times \mathbb{Z}/_{5\mathbb{Z}}$, while ZENRingOrigin(R) will be set to $\mathbb{Z}/_{15\mathbb{Z}}$. A subsequent call to ZENRingFree will also free this ring, but R1=ZENRingCopy(R) can be helpful to keep this original ring.

Procedure 242 *Chinese ring.*

```
int ZENRingChinese(R, N, PR)
    int N;
    ZENRing R, *PR;
```

Input: *An unallocated* ZENRing*, and an array of* ZENRings PR *of size* N.
Output: *0 if the ring* R *is succesfully initialized,* ZENNULL *otherwise.*
Note: *The obtained ring is the product of the* N *inputs. These rings must be of the same type: either* N *modular rings, or* N *polynomial extensions over a same ring. Nevertheless, some or all of these ring can be clones.*

Procedure 243 *Testing whether two rings are equal.*

```
int ZENRingAreEqual(R1,R2)
    ZENRing R1,R2;
```

Input: *Two* ZENRings.
Output: ZENERR *if an error occured, 1 if the two* ZENRings *are the same, 0 otherwise.*
Note: *A clone is equal to its original ring.*

4.8 Big integers layer of ZEN

The ZEN library was primarily built upon the BigNum library [2] (version 1.0-b).

Then, we have extended ZEN so that it can use GMP (from version 2.0), the Gnu Multiprecision Package [3]. In order to achieve this final goal, all the big integers functions were called in ZEN through a reformatted version of both BigNum and GMP' functions. These macros were prefixed by ZBN.

Since only few functions from BigNum or GMP are needed, we finally decided to design a specific big integers layer for ZEN, the ZBN layer (from version 3.0).

4.8.1 Caracteristics of the ZBN layer

Most of ZBN procedures are working on arrays of digits. The digits are of type BigNumDigit. An array of digits is of type BigNum. And the size of these arrays is of type BigNumLength. The sizes of these types depends on the computer you are using (generally, 32 or 64 bits). BigNumDigit must be an unsigned type. BigNumLength must be a signed.

4.8.1.1 Assembler directives

Unlike the library BigNum or the library GMP, one important principle followed while designing ZBN is to minimize as much as possible code written in assembler. There are two important benefits for such an approach:

- It is no more a problem to port the library to a new architecture. In particular, to optimize it since we only have to write only few assembler sentences.
- Compilers are becoming more and more powerfull. They are able to optimize codes much more efficiently than us (unrolling loops, managing cache misses, ...).

That's why, we limited our assembler directives to the few arithmetic assembler instructions which are not easily accessed in C. The coresponding macros are described below.

Procedure 244 *Adding two digits*

```
void zbnadd(c1, c0, a1, a0, b1, b0)
   BigNumDigit c1, c0;
   BigNumDigit a1, a0;
   BigNumDigit b1, b0;
```

Input: *Four digits* a0 *and* b0, a1 *and* b1 *to add.*
Side effect: $c0 = a0 + b0 \bmod 2^{SIZE\_BLOC}$ *and* $c1 = a1 + b1 + ((a0 + b0) \div 2^{SIZE\_BLOC}) \bmod 2^{SIZE\_BLOC}$.
Note: *This macro is sometimes an assembler directive.*

Procedure 245 *Subtracting two digits*

```
void zbnsub(c1, c0, a1, a0, b1, b0)
   BigNumDigit c1, c0;
   BigNumDigit a1, a0;
   BigNumDigit b1, b0;
```

Input: *Four digits* a0 *and* b0, a1 *and* b1 *to subtract.*
Side effect: $c0 = a0 - b0 \bmod 2^{SIZE\_BLOC}$ *and* $c1 = a1 - b1 - ((a0 - b0 \bmod 2^{SIZE\_BLOC+1}) \div 2^{SIZE\_BLOC}) \bmod 2^{SIZE\_BLOC}$.
Note: *This macro is sometimes an assembler directive.*

Procedure 246 *Multiplying digits*

```
void zbnmul(c1, c0, a, b)
   BigNumDigit c1, c0;
   BigNumDigit a;
   BigNumDigit b;
```

Input: *Two digits* a *and* b.
Side effect: $\mathsf{c0} = \mathsf{a0} \times \mathsf{b0} \bmod 2^{\mathsf{SIZE\_BLOC}}$ *and* $\mathsf{c1} = \mathsf{a1} \times \mathsf{b1} + ((\mathsf{a0} \times \mathsf{b0}) \div 2^{\mathsf{SIZE\_BLOC}}) \bmod 2^{\mathsf{SIZE\_BLOC}}$.
Note: *This macro is sometimes an assembler directive.*

Procedure 247 *Dividing 2 digits*

```
void zbndiv(q, r, n1, n0, d)
   BigNumDigit q;
   BigNumDigit r;
   BigNumDigit n1, n0;
   BigNumDigit d;
```

Input: *Two digits* n1 *and* n0 *to be divided by* d. *One must have* $\mathsf{d} \div 2^{\mathsf{SIZE\_BLOC}-1} = 1$.
Side effect: $\mathsf{q} = (\mathsf{n1} \times 2^{\mathsf{SIZE\_BLOC}} + \mathsf{n0}) \div \mathsf{d}$ *and* $\mathsf{r} = (\mathsf{n1} \times 2^{\mathsf{SIZE\_BLOC}} + \mathsf{n0}) \bmod \mathsf{d}$.
Note: *This macro is sometimes an assembler directive.*

Procedure 248 *Hamming weight of one digit*

```
void zbnwght(w, d)
   BigNumLength w;
   BigNumDigit d;
```

Input: *A digit* d.
Side effect: *The Hamming weight of* d *is stored in* w.
Note: *This is a macro which is sometimes an assembler directive. When none assembler directive is available, the algorithm used depends on the flag* LOW_MEMORY.

4.8.1.2 Memory limitation

Some of the functionalities of the ZBN layer, need a lot of memory at compilation time. You may decrease the amount of memory needed by using the following compilation flag which is set in zbn.h.

```
# define LOW_MEMORY 0
```

4.8.1.3 Karatsuba's multiplication on integers

The Karatsuba's multiplication on big integers is used by default in ZEN. This can be disabled using the KARA parameter.

```
# define KARA 1
```

4.8.2 Constants

Three constants are used in ZEN:

SIZE_BLOC depends on the arithmetic of computers. It is the size in bits of a BigNumDigit (usually 32 or 64 bits).

SIZE_CHAR is the size in bits of a char (usually 8 bits).

SIZE_SHORT is the size in bits of a short (usually 16 bits).

4.8.3 BigNum allocation

Procedure 249 *Allocation of a* BigNum

```
BigNum ZBNC(nl)
 BigNumLength nl;
```

Input: *A length* nl.
Output: *An allocated* BigNum *of size* nl *or* NULL *if an error occured.*

Procedure 250 *Freeing a* BigNum

```
void ZBNF(n)
   BigNum n;
```

Input: *An allocated* BigNum.
Side effect: n *is freed.*

4.8.4 Basic functions on BigNumDigits

We defined the following macros to help manipulating a BigNum.

Procedure 251 *Reduction modulo a bloc*

```
BigNumLength modSizeBloc(x)
   BigNumLength x;
```

Input: *An integer* x.
Output: *The remainder of* x/SIZE_BLOC, *that is to say* x mod SIZE_BLOC

Procedure 252 *Number of blocs*

```
BigNumLength divSizeBloc(x)
   BigNumLength x;
```

Input: *An integer* x.
Output: *The quotient* x/SIZE_BLOC

The two following functions have an equivalent meaning, once replaced SIZE_BLOC by SIZE_CHAR.

```
BigNumLength modSizeChar(x)
```

```
BigNumLength divSizeChar(x)
```

Procedure 253 *Shifting 1*

```
BigNumDigit BlocExp2(x)
   unsigned int x;
```

Input: *An integer* x.
Output: *The* BigNumDigit *representing* 2^{x} *if* $\mathsf{x} < \mathsf{SIZE\_BLOC}$, *0 otherwise.*

Procedure 254 *Getting least significant digit of a* BigNum

```
BigNumDigit ZBNGetDigit(n)
   BigNum n;
```

Input: *An allocated* BigNum.
Output: *The least significant digit of* n.

Procedure 255 *Assigning least significant digit of a* BigNum

```
void ZBNSetDigit(n,d)
   BigNum n;
   BigNumDigit d;
```

Input: *An allocated* BigNum *and a* BigNumDigit.
Side effect: *The least significant digit of* n *is set to* d.

Procedure 256 *Getting i^{th} digit of a* BigNum

```
BigNumDigit ZBNDigit(n,i)
   BigNum n;
   int i;
```

Input: *An integer and an allocated* BigNum.
Output: *The* i*th digit of* n.
Note: *The 0th digit is the least significant one. If* n *is of size* nl, *the (*nl*-1)th is the most significant one. This is a macro that can be a lvalue in an assignment.*

Procedure 257 *Number of unsignificant bits in a digit*

```
BigNumLength ZBNNumLeadingZeroBitsInDigit(d)
   BigNumDigit d;
```

Input: *A digit* d.
Output: *The number* k *of most significant bits set to zero in* d.
Note: *If* d *is zero,* k = SIZE_BLOC.

Procedure 258 *Number of significative bits*

```
unsigned int ZBNNumBitsInDigit(d)
   BigNumDigit d;
```

Input: *A digit* d.
Output: *The number* k *of significative bits in* d.
Note: *If* d *is zero,* k = 0.

Procedure 259 *Significant length of a* BigNum

```
BigNumLength ZBNNumDigits(n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum *and its length.*
Output: *The number of significant digits of* (n,nl).
Note: *If* (n,nl) *is zero, returns 1.*

4.8.5 Assigning

Procedure 260 *Set a* BigNum *to zero*

```
void ZBNSetToZero (n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum *and its length.*
Side effect: *The* BigNum *is filled with zeros.*

Procedure 261 *Set a* BigNum *to one*

```
void ZBNSetToOne (n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum *and its length.*
Side effect: *The* BigNum *is filled with zeros execept the first digit which is set to one.*

Procedure 262 *Assigning* BigNum*s*

```
void ZBNAssign(m, n, nl)
   BigNum m;
   BigNum n;
   BigNumLength nl;
```

Input: *Two allocated* BigNum*s and a length.*
Side effect: *The* BigNum m *is filled with* (n,nl).
Note: *All kinds of overlapping are possible. No side effect if* nl *is zero.*

4.8.6 Random assignment

In the same spirit as for allocation functions, we allow customized random functions.

Procedure 263 *Pseudo-Random Generator (PRG) initialization with a BigNumDigit*

```
void zbnsrandom(seed)
   BigNumDigit seed;
```

Input: *A* BigNumDigit *which serves as seed for the internal PRG.*
Note: *The PRG behavior is determined by this seed. The programs included in* ZEN *use this function to initialize the PRG. By default, this function calls the native* ZENNativeSrand *function. This can be overwritten by* ZENSetRandomFunctions. *This function is provided for compatibility purposes with older version of ZEN (cf.* ZENNativeSRand*).*

Procedure 264 *Pseudo-Random Generation of a BigNumDigit*

```
BigNumDigit zbnrandom()
```

Output: *A pseudo-random* BigNumDigit.
Note: *This function is used all over the* ZEN *and* ZENFACT *libraries when random generation is needed. By default, this function calls the native* ZENNativeRand *function. This can be overwritten by* ZENSetRandomFunctions.

Procedure 265 *Set a* BigNum *randomly*

```
void ZBNSetRandom(n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum *and its length.*
Side effect: *The* BigNum *is filled randomly.*

4.8.7 Comparisons

Procedure 266 *Test if a big integer value is equal to zero*

```
int ZBNIsZero (n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum *and its length.*
Output: *The predicate* (n,nl) == 0

Procedure 267 *Test if a big integer value is equal to one*

```
int ZBNIsOne (n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum *and its length.*
Output: *The predicate* (n,nl) == 1

Procedure 268 *Comparison of bignums*

```
int ZBNCompare(m, ml, n, nl)
   BigNum m;
   BigNumLength ml;
   BigNum n;
   BigNumLength nl;
```

Input: *Two allocated* BigNums *and their lengths*
Output:

- 1 *if* m > n*;*
- *zero if* m = n*;*
- −1 *if* m < n.

Procedure 269 *Test if two bignums are equal*

```
int ZBNAreEqual(m, ml, n, nl)
   BigNum n, m;
   BigNumLength nl, ml;
```

Input: *Two allocated* BigNums *and their lengths*
Output: *The predicate* (n,nl) == (m,ml)

Procedure 270 *Comparison of significant bignums*

```
int ZBNCompareLazy(m, ml, n, nl)
   BigNum m, n;
   BigNumLength ml, nl;
```

Input: *Two bignums*
Output:

- 1 *if* m > n*;*
- *zero if* m = n*;*
- −1 *if* m < n.

Note: *One must have* m[ml-1] ≠ 0 *and* n[nl-1] ≠ 0.

4.8.8 Binary operations

Procedure 271 *And*

```
void ZBNAnd(m, n, nl)
   BigNum m, n;
   BigNumLength nl;
```

Input: *Two allocated* BigNum*s and a length.*
Side effect: (m,nl) = (m,nl) *and* (n,nl).
Note: *One must have the allocated size of* m *which must be greater (or equal) than* nl.

Procedure 272 *Or*

```
void ZBNOr(m, n, nl)
   BigNum m, n;
   BigNumLength nl;
```

Input: *Two allocated* BigNum*s and a length.*
Side effect: (m,nl) = (m,nl) *or* (n,nl).
Note: *One must have the allocated size of* m *which must be greater than* nl.

Procedure 273 *Xor*

```
void ZBNXor(m, n, nl)
   BigNum m, n;
   BigNumLength nl;
```

Input: *Two allocated* BigNum*s and a length.*
Side effect: (m,nl) = (m,nl) *xor* (n,nl).
Note: *One must have the allocated size of* m *which must be greater than* nl.

4.8.9 Addition

Procedure 274 *Incrementation*

```
BigNumDigit ZBNAddCarry(m, ml, carry)
   BigNum m;
   BigNumLength ml;
   BigNumDigit carry;
```

Input: *An allocated* BigNum *and its length, and a carry in that must be 0 or 1.*
Output: *The carry out.*
Side effect: (m,ml)+ = carry
Note: *If* ml *is zero, nothing is done and the returned value is* carry.

Procedure 275 *Addition*

```
BigNumDigit ZBNAdd (m, ml, n, nl, carry)
   BigNum m;
   BigNumLength ml;
   BigNum n;
   BigNumLength nl;
   BigNumDigit carry;
```

Input: *Two allocated* BigNum*s, their lengths and a carry in that must be be 0 or 1.*
Output: *The carry out*
Side effect: (m,ml)+ = (n,nl) + carry.
Note: *One must have* nl $\leq$ ml. *If* nl $= 0$, *then* ZBNAdd(m,ml,n,0,c) *behaves like* ZBNAddCarry(m,ml,c).

4.8.9.1 Subtraction

Procedure 276 *Additive inverse*

```
void ZBNComplement(n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum
Side effect: $(n,nl) \leftarrow 2^{SIZE\_BLOC \times nl} - 1 - (n,nl)$
Note: *If* nl *equals zero, does nothing.*

Procedure 277 *Decrementation*

```
BigNumDigit ZBNSubtractBorrow(m, ml, carry)
   BigNum m;
   BigNumLength ml;
   BigNumDigit carry;
```

Input: *An allocated* BigNum*, its length and a borrow in that must be 0 or 1.*
Output: *The borrow out.*
Side effect: $(m,ml) -= (1 - carry)$
Note: *If* ml *is zero, nothing is done and the return value is* carry*.*

Procedure 278 *Subtraction*

```
BigNumDigit ZBNSubtract(m, ml, n, nl, carry)
   BigNum m;
   BigNumLength ml;
   BigNum n;
   BigNumLength nl;
   BigNumDigit carry;
```

Input: *Two allocated* BigNums*, their length and a borrow in that must be be 0 or 1.*
Output: *The borrow out.*
Side effect: $(m,ml) -= (n,nl) + (1 - carry)$
Note: *One must have* $nl \leq ml$*. If* $nl = 0$*, then* ZBNSubtract(m,ml,n,0,b) *behaves like* ZBNSubtractBorrow(m,ml,b)*.*

4.8.9.2 Shifting

4.8.9.3 Shifting a BigNum

Procedure 279 *Shifting left of a small amount*

```
BigNumDigit ZBNShiftLeft(n, nl, l)
   BigNum n;
   BigNumLength nl;
   BigNumLength l;
```

Input: *An allocated* BigNum, *its length and a positive integer* $0 \leq$ l $<$ SIZE_BLOC.
Output: *The digit shifted out*
Side effect: (n,nl) <<= l
Note: *If* l *equals zero, nothing is done and the returned value is 0.*

Procedure 280 *Shifting right of a small amount*

```
BigNumDigit ZBNShiftRight(n, nl, l)
   BigNum n;
   BigNumLength nl;
   BigNumLength l;
```

Input: *An allocated* BigNum, *its length and a positive integer* $0 \leq$ l $<$ SIZE_BLOC.
Output: *The digit shifted out*
Side effect: (n,nl) >>= l
Note: *If* l *equals zero, nothing is done and the returned value is 0.*

Procedure 281 *Shifting left*

```
void ZBNAnyShiftLeft(r, p_rl, n, nl, nnn)
   BigNum r,n;
   BigNumLength *p_rl,nl;
   unsigned int nnn;
```

Input: *Two allocated* BigNums, *a pointer on a length, and an integer* nnn.
Side effect: r = n << nnn
Note: *The size of* r *must be at least* nl + divSizeBloc(nnn), *and such that the result of shift can be stored. Hence, an allocated size of* nl + divSizeBloc(nnn) + 1 *is always sufficient. The* BigNum n *can be part of* r. *The length of* r *is stored in* ⋆p_rl. *The pointer* p_rl *can be the address of the external value of* nl.

Procedure 282 *Shifting right.*

```
void ZBNAnyShiftRight(n,nl,nnn)
    BigNum n;
    BigNumLength nl;
    unsigned int nnn;
```

Input: *An allocated* BigNum n *of* nl BigNumDigits *and an integer* nnn.
Side effect: n >>= nnn
Note: *Low degree bits are lost*

4.8.10 Multiplication

Procedure 283 *Multiplication by a digit*

```
BigNumDigit ZBNMultiplyDigit(m, ml, n, nl, d)
    BigNum m; BigNumLength ml;
    BigNum n; BigNumLength nl;
    BigNumDigit d;
```

Input: *Two allocated* BigNums *and a digit.*
Output: *A carry out.*
Side effect: (n,nl)+ = (m,ml) × d.
Note: *One must have* nl ≥ ml + 1. *If* ml *equals zero, the return is 0 and nothing is done.*

Procedure 284 *Multiplication and Add*

```
BigNumDigit ZBNMultiply(p, pl, m, ml, n, nl)
    BigNum p, m, n;
    BigNumLength pl, ml, nl;
```

Input: *Three allocated* BigNums *and their lengths*
Output: *A carry out.*
Side effect: (p,pl)+ = (m,ml) × (n,nl).
Note: *One must have* pl ≥ ml + nl. *If* nl *equals zero, there is no side effect and the carry out is 0.*

Procedure 285 *Multiplication*

```
void ZBNMult(n, a, al, b, bl)
   BigNum n, a, b;
   BigNumLength al, bl;
```

Input: *Three* BigNum*s and their lengths*
Side effect: (n,nl) $\leftarrow$ (a,al) $\times$ (b,bl).
Note: *One must have* nl $\geq$ al + bl, al *and* bl *non zeros.*

Procedure 286 *Squaring and Add*

```
BigNumDigit ZBNSquare(n, nl, a, al)
   BigNum n, a;
   BigNumLength nl,al;
```

Input: *Two allocated* BigNum*s and their lengths*
Output: *A carry out.*
Side effect: (n,nl)+ = $(\mathsf{a,al})^2$.
Note: *One must have* nl $\geq$ 2al. *If* al *equals zero, there is no side effect and the carry out is 0.*

Procedure 287 *Squaring*

```
void ZBNSqu(n, a, al)
   BigNum n, a;
   BigNumLength al;
```

Input: *Two* BigNum*s and a length*
Side effect: (n,nl) $\leftarrow$ $(\mathsf{a,al})^2$.
Note: *One must have* n *distinct from* a, nl $\geq$ 2al, *and* al > 0.

Procedure 288 *Karatsuba's multiplication of two* BigNum*s.*

```
BigNumDigit ZBNMultiplyKaratsuba(x,xl,a,al,b,bl,lim)
   BigNum x,a,b;
   BigNumLength xl,al,bl,lim;
```

Input: *Three allocated* BigNum*s with* xl $\geq$ al + bl *and the limit of size under which* ZBNMultiplyPlain *must be used.*
Output: *The possible carry out, or* ZENERR *if an error occured.*
Side effect: x += a $\star$ b
Note: *A buffer is allocated and freed.*

Procedure 289 *Karatsuba's multiplication*

```
void ZBNMultiplyKaratsubaBuffer(x,a,al,b,bl,buf,lim)
   BigNum x,a,b,buf;
   BigNumLength al,bl,lim;
```

Input: *Four* BigNum*s:* x = 0 *of size greater or equal than* al + bl, *a buffer of size at least* ZBNSizeBufKaraM(bl), *and the limit of size under which* ZBNMultiply *must be used.*
Side effect: x = a × b

Procedure 290 *Size of buffer needed for Karatsuba's multiplication*

```
BigNumLength ZBNSizeBufKaraM(l)
   BigNumLength l;
```

Input: *The length of smallest operand.*
Output: *The length of the buffer needed in Karatsuba's routines*

Procedure 291 *Karatsuba's squaring of two* BigNum*s.*

```
BigNumDigit ZBNSquareKaratsuba(x,xl,a,al,lim)
   BigNum x,a;
   BigNumLength xl,al,lim;
```

Input: *Two allocated* BigNum*s with* xl ≥ 2al *and the limit of size under which* ZBNSquare *must be used.*
Output: *The possible carry out, or* ZENERR *if an error occured.*
Side effect: x += a ⋆ a
Note: *A buffer is allocated and freed.*

Procedure 292 *Karatsuba's squaring*

```
void ZBNSquareKaratsubaBuffer(x,a,al,buf,lim)
   BigNum x,a,buf;
   BigNumLength al,lim;
```

Input: *Three* BigNum*s:* x = 0 *of size greater or equal than* 2 × al, *a buffer of size at least* ZBNSizeBufKaraM(bl), *and the limit of size under which* ZBNMultiply *must be used.*
Side effect: $x = a^2$

Procedure 293 *Size of buffer needed for Karatsuba's squaring*

```
BigNumLength ZBNSizeBufKaraS(l)
   BigNumLength l;
```

Input: *The length of smallest operand and the limit under which standard multiplication is performed.*
Output: *The length of the buffer needed in Karatsuba's routines*

Procedure 294 *Setting of Karatsuba's cutoff*

```
BigNumLength ZBNKaratsubaMultiplyCutoff(size)
   BigNumLength size;
```

Input: *The size (≥ 2) in* BigNumDigit*s of the numbers to multiply*
Output: ZENERR *if an error occured, 0 if Karatsuba's operations are slower, the cutoff to use otherwise.*
Note: *For all possible values of cutoff, a number of randomized multiplications are performed. The fastest parameter is returned. Karatsuba's operations are only chosen if a speed increase of at least 10% is observed. This is made because optimizers can often include simple operations but not complex ones. If difference is tiny between Karatsuba's and standard algorithm, standard algorithm often leads to final best results.*

Procedure 295 *Setting of Karatsuba's cutoff*

```
BigNumLength ZBNKaratsubaSquareCutoff(size)
   BigNumLength size;
```

Input: *The size in* BigNumDigit*s of the numbers to square*
Output: ZENERR *if an error occured, 0 if Karatsuba's operations are slower than* ZBNSquarePlain*, the cutoff to use otherwise.*
Note: *For all possible values of cutoff, a number of randomized squares are performed. The fastest parameter is returned.*

4.8.11 Division

Procedure 296 *Division by a digit*

```
BigNumDigit ZBNDivideDigit(q, n, nl, d)
    BigNum q, n;
    BigNumLength nl;
    BigNumDigit d;
```

Input: *Two allocated* BigNum*s, a* BigNumLength *and a* BigNumDigit.
Output: *The remainder of the division* n/d.
Side effect: *The* BigNum (q,nl-1) *is filled with the quotient* $\frac{n}{d}$.
Note: *The value* d *must be greater than* ZBNDigit(n,nl-1).

Procedure 297 *Division*

```
void ZBNDivide(n, nl, d, dl)
    BigNum n, d;
    BigNumLength nl, dl;
```

Input: *Two allocated* BigNum*s and their lengths.*
Side effect: *The quotient and the remainder of* (n,nl)/(d,dl) *are stored in* n. *The* dl *least significant digits contains the remainder, whereas the* nl − dl *most significant digits contains the quotient.*
Note: *The value* ZBNDigit(d,dl-1) *must be strictly greater than* ZBNDigit(n,nl-1).

4.8.12 Logarithms

.

Procedure 298 *Logarithm in base e of a* BigNum

```
double ZBNLog(n, nl)
    BigNum n;
    BigNumLength nl;
```

Input: *An allocated* BigNum n *of length* nl.
Output: *The logarithm in base e of (*n, nl*)*

4.8.13 Square root

.

Procedure 299 *Square root of a* BigNum

```
BigNumLength ZBNSqrt(r, a, al)
   BigNum r, a;
   BigNumLength al;
```

Input: *An allocated* BigNum a *of length* al *and another* BigNum r.
Output: *0 if an error occured, the size of the square root otherwise*
Side effect: r *is filled with floor(sqrt(a, al)).*

Conversions between the internal representation and the ASCII representation of a BigNum are done by ZBNPrintToString and ZBNReadFromString. ZBNPrintToFile and ZBNReadFromFile allows to write/read a BigNum in a file using these functions.

One can moreover write to and then read from a file a BigNum in an internal format with ZBNPutToFile and ZBNGetFromFile.

The Lazy forms of these functions assume that the BigNum is of known size and perform no allocation.

The new version of BigNum has similar functionalities with BnFromString, BnToString, BnFillString, BnPrintToFile and BnReadFromFile. Of course GMP has also such features with the mpz_[inp\vertout]_[str\vertraw] functions. We however keep the following functions that were present in the first version of ZEN. This should ensure compatibility for reading data files written with programs that were using first version of ZEN.

Note 4 *These functions cannot manage correctly* BigNums *of length larger than* 2^{32}.

Procedure 300 *Converting to string*

```
char *ZBNPrintToString(n,nl,base)
   BigNumLength nl;
   int base;
   BigNum n;
```

Input: *An allocated* BigNum n *of length* nl *and a base* base$\in \{2,\ldots,16\}$.
Output: *An allocated string representing* n *in base* base, *or* ZENNULL *if an error occured.*

Procedure 301 *Converting from string*

```
int ZBNReadFromStringLazy(m, ml, s, base)
    BigNum m;
    BigNumLength ml;
    char *s;
    int base;
```

Input: *An allocated* BigNum *and its length* nl *and a string* s *representing a* BigNum *in base* base$\in \{2, \ldots, 16\}$.
Output: *The number of character read if no error occured,* ZENERR *otherwise*
Side effect: n *is filled with* s

Procedure 302 *Converting from string*

```
int ZBNReadFromString(p_n, p_nl, s, base)
    BigNum *p_n;
    BigNumLength *p_nl;
    char *s;
    int base;
```

Input: *A pointer* p_n *on a non allocated* BigNum, *a pointer* p_nl *to a* BigNumLength *and a string* s *representing a* BigNum *in base* base$\in \{2, \ldots, 16\}$.
Output: *The number of character read if no error occured,* ZENERR *otherwise*
Side effect: p_n *is allocated and is filled with* s

Procedure 303 *Converting to string in an internal representation*

```
char *ZBNPutToStringLazy(n,nl)
    BigNum n;
    BigNumLength nl;
```

Input: *An allocated* BigNum n *of length* nl.
Output: *An allocated string representing* n *in base* base *if no error occured,* ZENNULL *otherwise.*
Note: *The size* nl *is not saved in the string.*

Procedure 304 *Converting to string in an internal representation*

```
char *ZBNPutToString(n,nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum n *of length* nl.
Output: *An allocated string representing* n *in base* base *if no error occured,* ZENNULL *otherwise.*
Note: *The size of the* BigNum *is saved in the string.*

Procedure 305 *Converting from string in an internal representation*

```
int ZBNGetFromStringLazy(n, nl, s)
   BigNum n;
   BigNumLength nl;
   char *s;
```

Input: *An allocated* BigNum *and its length* nl *and a string* s *representing a* BigNum *in an internal representation.*
Output: *The number of character read if no error occured,* ZENERR *otherwise*
Side effect: n *is filled with* s

Procedure 306 *Converting from string in an internal representation*

```
int ZBNGetFromString(p_n, p_nl, s)
   BigNum *p_n;
   BigNumLength *p_nl;
   char *s;
```

Input: *A pointer* p_n *on a non allocated* BigNum, *a pointer* p_nl *to an integer and a string* s *representing a* BigNum *in an internal representation.*
Output: *The number of character read if no error occured,* ZENERR *otherwise*
Side effect: p_n *is allocated and is filled with* s

The syntax of file functions is the same, using a FILE *file instead of a string.

```
int ZBNPrintToFile(file,n,nl,base)
```

```
int ZBNReadFromFileLazy(n, nl, file, base)
```

```
int ZBNReadFromFile(p_n, p_nl, file, base)

int ZBNPutToFileLazy(file,n,nl)

int ZBNPutToFile(file,n,nl)

int ZBNGetFromFileLazy(n, nl, file)

int ZBNGetFromFile(p_n, p_nl, file)
```

4.8.14 Greatest Common Divisor

Two families of procedures are provided. The goal of the first family is the computation of greatest common divisor of integers and the goal of the second is the computation of extended gcd.

Procedure 307 *Integer gcd of two* BigNumDigit*s*

```
BigNumDigit ZBNDigitGCD(a_in, b_in)
   BigNumDigit a_in, b_in;
```

Input: *Two* BigNumDigit*s* a_in *and* b_in.
Output: *The gcd of* a_in *and* b_in.
Note: *A binary algorithm is used.*

Procedure 308 *Gcd of two* BigNum*s*

```
int ZBNGcd(g, p_gl, n, nl, m, ml)
   BigNum g, n, m;
   BigNumLength *p_gl, nl, ml;
```

Input: *Two allocated* BigNum*s* n *and* m *of size* nl *and* ml, *a* BigNum g *of allocated size greater than* nl *and* ml.
Output: *The real size of the gcd or* ZENERR *if an allocation error occured.*
Side effect: g *receives the gcd of* n *and* m.
Note: *One can have* g=n=m.

Procedure 309 *Extended gcd of two* BigNum*s*

```
int ZBNEea(g, p_gl, a, p_al, n, nl, m, ml)
   BigNum g, a, n, m;
   BigNumLength *p_gl, *p_al, nl, ml;
```

Input: *Two allocated* BigNum*s (*n, nl*) and (*m, ml*) with* n≥m *and* nl≥ml*, two* BigNum*s* a *and* g *of allocated size* ≥ nl.
Output: *If the gcd is equal to 1,* ZBN_HAS_INVERSE *is returned and* a *is equal to* 1/m *mod* n*, if the gcd is greater than one,* ZBN_NO_INVERSE *is returned and* g *is the gcd. When an allocation error occured,* ZENERR *is returned.*
Note: *This porcedure is a front head for* ZBNEeaLehmer *if* LEHMER *is set at the compilation or* ZBNEeaPlain *otherwise.*

4.8.15 Modular operations

Procedure 310 *Reduction of a* BigNum *modulo another*

```
BigNumLength ZBNModulo(p_m,n,nl,m,ml)
   BigNum *p_m,n,m;
   BigNumLength nl,ml;
```

Input: *Two allocated* BigNum*s and their lengths, and a pointer on a non allocated* BigNum.
Output: ZENERR *if an error occured, the length of the result otherwise.*
Side effect: ⋆p_m *is allocated and set to* n mod m.

Procedure 311 *Reduction of a* BigNum *modulo another*

```
BigNumLength ZBNModuloLazy(p_m,n,nl,m,ml)
   BigNum *p_m,n,m;
   BigNumLength nl,ml;
```

Input: *Two allocated* BigNum*s and their lengths, and a pointer on a non allocated* BigNum.
Output: *0 if an error occured, the length of the result otherwise.*
Side effect: ⋆p_m *is allocated and set to* n mod m.
Note: *One must have* ZBNDigit(n,nl-1) $\neq 0$ *and* ZBNDigit(m,ml-1) $\neq 0$.

Procedure 312 *Modular addition*

```
void ZBNModAdd(a,al,b,bl,m,ml)
   BigNum a,b,m;
   BigNumLength al,bl,ml;
```

Input: *Three allocated* BigNum*s.*
Side effect: $a = a + b \bmod m$
Note: *One must have* ml = al, a < m, b < m *and* ZBNDigit(m,ml-1) $\neq 0$. *One can have* a = b.

Procedure 313 *Modular negation*

```
void ZBNModNegate(a,al,b,bl,m,ml)
   BigNum a,b,m;
   BigNumLength al,bl,ml;
```

Input: *Three distinct allocated* BigNum*s.*
Side effect: $a = -b \bmod (m + 1)$
Note: *One must have* $al \geq ml$, $b \leq m$ *and* $\star(m+ml) \neq 0$.

Procedure 314 *Modular subtraction*

```
void ZBNModSubtract(a,al,b,bl,m,ml)
   BigNum a,b,m;
   BigNumLength al,bl,ml;
```

Input: *Three distincts allocated* BigNum*s.*
Side effect: $a = a - b \bmod (m + 1)$
Note: *One must have* $ml \geq al \geq sizeof(m - 1)$, $a \leq m$ *and* $b \leq m$.

Procedure 315 *Modular multiplication*

```
void ZBNModMultiply(x,xl,a,al,b,bl,m,ml)
   BigNum x,a,b,m;
   BigNumLength xl,al,bl,ml;
```

Input: *Four allocated* BigNum*s.*
Side effect: $x = a.b \bmod m$
Note: *One must have* $xl \geq al + bl + 1$, $\star(a+al-1) \neq 0$ *and* $\star(b+bl-1) \neq 0$. *One can have* a = b.

Procedure 316 *Modular squaring*

```
void ZBNModSquare(x,xl,a,al,m,ml)
   BigNum x,a,m;
   BigNumLength xl,al,ml;
```

Input: *Three allocated* BigNum*s.*
Side effect: x = $a^2 \bmod m$
Note: *One must have* xl $\geq 2\times$al+1, $\star$(a+al-1) $\neq 0$ *and* $\star$(b+bl-1) $\neq 0$.

4.8.16 Hamming weight of a BigNum

Procedure 317 *Number of bits of a digit*

```
BigNumLength ZBNWeightDigitFast(d)
   BigNumDigit d;
```

Input: *A* BigNumDigit
Output: *The number of non zero bits in* d
Note: *This function is only provided for compatibility. Using directly the macro* zbnwght *is probably slightly faster.*

Procedure 318 *Weight of a BigNum*

```
BigNumLength ZBNWeight(n, nl)
   BigNum n;
   BigNumLength nl;
```

Input: *An allocated* BigNum n *of size* nl.
Output: *The number of 1 in the binary representation of* n.

Chapter 5

Implementation principles of ZEN.

5.1 How to write a new arithmetic

This section is subtitled "How to write a new arithmetic" as this is certainly the best way to understand the principles of ZEN.

An arithmetic can be mathematically defined by a set of elements that contains at least two elements 0 and 1, and two operations + and × verifying the classical properties.

Writing a new arithmetic will therefore consist in implementing these operations. Of course, technical reasons impose a much larger set of operations. In fact, the basic point is the initialization routine that will produce a ZENRing with all of the function pointers set to appropriate values.

Basically, all the Elt functions have to be specifically written for every new arithmetic. For the remaining functions, one can first try the general functions of the zed/ and zeg/ directories that will provide operations on polynomials, matrices, series and elliptic curves, and therefore, allow extensions based on the new arithmetic.

A modification of the _ZENBaseRingCreate function is also needed to take in account the modification. Writing a new extension arithmetic is also possible — for instance if you need very efficient operations on $\mathbb{F}_{33^3}$ and you know how to improve existing operations in this case — but the above modification will of course take place in the _ZENExtRingCreate function

Eventually, other specific functions can be written to replace the _ZEN or Zed functions.

5.2 Sub-directories of ZEN

The sub-directories of ZEN can be ordered in three different sets:

- The system ones content some definitions and/or procedures useful for the whole library.
- The motor ones constitute the heart of the library.

- The modules were added one by one to improve the performances of some functions.

5.2.1 System sub-directories

sub-directories	— contents
sys/	— system features like error handling, runtime procedure and documentation extraction.
zbn/	— big integers procedures.
prgm/	— test and bench programs for the library.

5.2.2 The general functions

sub-directories	— contents
zed/	— default functions to manipulate the types of ZEN.
zeg/	— general functions for any arithmetic. These functions are able to replace any specific function written in the directories of 5.2.3. These functions only use functions of zed.
zext/	— the recursive functions for any extension over a ring.
zer/	— the ZENRing definition functions.

5.2.3 Arithmetics

These are the specific arithmetics already implemented in ZEN. For each of these arithmetic, all the functions Elt are implemented, but only some of the other functions are. See the corresponding chapters of the "advanced user manual" for the specific functions implemented.

sub-directories	— contents
ze2/	— The arithmetic of the Galois field with two elements $\mathbb{F}_2$. See the "advanced user manual"
zep/	— The arithmetic of the modular ring $\mathbb{Z}/p\mathbb{Z}$. See the "advanced user manual".
zef/	— The arithmetic of the ring $\mathbb{Q}$. See the "advanced user manual".
zeps/	— The arithmetic of the modular ring $\mathbb{Z}/p\mathbb{Z}$ with $2 < p < 2^{\text{SIZE\_BLOC}}$. See the "advanced user manual".
zetab/	— The arithmetic of tabulated clones. Every extension with p^m elements such that $3 \leq p^m \leq 2^{\text{SIZE\_CHAR}}$ can be tabulated.
zec/	— The arithmetic of clones using Chinese remainder theorems.
zem/	— The arithmetic of modular clones using Montgomery's idea.
zelog/	— The arithmetic of clones with Zech's logarithms. Every finite field with p^m elements such that $3 \leq p^m \leq 2^{\text{SIZE\_SHORT}}$ can be cloned with this arithmetic.

5.3 Generic functions

5.3.1 Ring initialization

The zer directory contains the ring initialization functions.

ring.c	contains the initialization procedures for generic functions for polynomials, matrices, and series. It also provides some basic operations that are called by every ring initialization functions.
base_ring.c	contains the ZENBaseRingCreate() function, which chooses the best arithmetic to use.
ext_ring.c	contains the ZENExtRingCreate() function, which chooses the best arithmetic to use (for the moment, there is only one (the zext arithmetic, but this should be sufficient as it uses the underlying arithmetic operations).
clone_ring.c	contains the ZENRingClone() function, which chooses the best arithmetic to use, when precomputations are allowed.

5.3.2 Extensions

The zext arithmetic provides generic operations for extensions over a previuously defined ZENRing. It interfaces the polynomials operations of the basic rings with the element operations of the new extension. Using ze2 for instance, this gives efficient operations in every characteristic 2 finite field at no extra cost.

5.3.3 General functions

The zeg directory provides all types of operations for polynomials, matrices, series and elliptic curves. These operations can be used as is in every case (even the future arithmetics). However, the main goal of specific arithmetics is to provide faster functions in some cases.

5.3.4 Default functions

The zed directory provides generic data structures of polynomials, matrices and series. However, the main goal of specific arithmetics may be to provide better data structure in some cases.

5.4 Arithmetics

5.4.1 Modular rings

The arithmetic zep is devoted to modular arithmetic with unlimited size modulus. It is called by ZENBaseRingCreate() function when none of the following arithmetics is available or pertinent. In this arithmetic, an element is stored as a BigNum and the operations are mainly the modular operations provided by the zbn operations.

5.4.2 Modular rings with small modulus

When the modulus is small, it is unefficient to use the heavy BigNum representation. The arithmetic zeps provides modular arithmetic when the modulus fits in a single computer word (BigNumDigit). It is called by ZENBaseRingCreate() function when this condition is true, that is to say when the modulus is strictly less than the following bounds:

65536	for a 16-bits computer, (but we have never tested the library in this case)
4294967296	for a 32-bits computer,
18446744073709551616	for a 64-bits computer.

In this arithmetic, an element is also stored as a BigNumDigit and the operations are mainly the modular operations provided by the standard C syntax.

5.4.3 $\mathbb{F}_2$ case

The binary case must be implemented in a very different way than the previous ones, in order to be efficient. The arithmetic ze2 can be called by ZENBaseRingCreate() function. The element arithmetic is trivial, but the main differences appear in polynomial and matricial functions. These mathematical objects are stored in BigNums, and the corresponding operations directly use the zbn functions.

5.4.4 Rationals

The arithmetic zef is quite different than the previous ones. It is only experimental. It provides a double BigNum structure for fractions (based on classical numerator/denominator representation) and the corresponding operations based on zbn functions. There is a lot of compatibility problems with this arithmetic, and efficiency is not guarenteed but it may be useful for ponctual applications.

5.4.5 Clones

The purpose of clones is to perform when it is possible some precomputations in order to speed up arithmetic. These precomputations are made upon initialization and can take some time. Therefore, cloning can be efficient only if a lot of operations are performed.

5.4.5.1 Tabulated clone

A small finite ring can be cloned using the index representation. All the elements Z of a ZENRing R are represented in the clone C obtained from R by the result (n,nl) of ZENEltToZ(n,p_nl,Z,R). That is to say, each element of a ring is ordered by the integer value it takes once evaluated in the characteristic.

Addition, multiplication, negation and inversion are tabulated at the initialization. Therefore, all the subsequent operations will take constant time. The limit size is that of an unsigned char, that is to say 256 elements. Polynomials and matrices use also the same representation which saves memory .

5.4.5.2 Logarithm clone

A small finite field $\mathbb{F}$ can be cloned using the logarithm representation. The first operation performed is to find a generator α of the finite field. Then, a table of all the logarithms is computed. The adopted representation in ZEN is the following:

Element of $\mathbb{F}$	$\mapsto$	ZEN representation
0	$\mapsto$	0
1	$\mapsto$	1
α	$\mapsto$	2
α^i	$\mapsto$	$i+1$

Hence, multiplication and inversion are easily performed by a modular addition on the exponent, assuming that a first test of equality to zero is performed on each operand:

$$\begin{array}{rcl} \alpha^i \times \alpha^j & \mapsto & (i+1)+(j+1)-1 \\ (\alpha^i)^{-1} & \mapsto & -(i+1)+2 \end{array}$$

For negation, the table of this operation is computed at the initialization of the clone. For addition, another table is computed that stores all the exponent of each element incremented by one. Addition of two elements can then be performed by a multiplication using the formula

$$\alpha^i + \alpha^j = \alpha^i(1+\alpha^{j-i}).$$

The limit size is that of an unsigned short, that is to say at most 65536 elements. Polynomials and matrices use also the same representation which saves memory.

5.4.5.3 Use of chinese remainder theorem

A ZENRing can be built upon two ZENRings using the Chinese remainder theorem.

Theorem *Let m and n be two natural integers, m prime with n. The two rings $\mathbb{Z}/_{(mn)\mathbb{Z}}$ and $\mathbb{Z}/_{m\mathbb{Z}} \times \mathbb{Z}/_{n\mathbb{Z}}$ are isomorphic. More precisely, the application*

$$\begin{array}{rcccl} \theta & : & \mathbb{Z}/_{mn\mathbb{Z}} & \to & \mathbb{Z}/_{m\mathbb{Z}} \times \mathbb{Z}/_{n\mathbb{Z}} \\ & & x & \mapsto & (x \bmod m, x \bmod n) \end{array}$$

is isomorphic and its reciprocal is

$$\theta^{-1}(x_m, x_n) = x_m n(n^{-1} \bmod m) + x_n m(m^{-1} \bmod n) \bmod mn.$$

The same kind of result can be stated for polynomials.

The implementation of these results in ZEN is more general: one can use N ZENRings to build the two isomorphics ZENRings. The N ZENRings must be of same level (N modular rings, or N extensions over same ring). The representation of an element in such a ring, is the N-array of the N projections of this element in the N subrings.

The function ZENChineseRingCreate() performs such a construction.

5.5 Testing the library

When adding some new arithmetics to ZEN, it is necessary to test these new features. This is done using the standard zentest program.

At the compilation time, a test program is compiled. To use it, you only have to type bin/arch/zentest. This program tests all the functions of the library (and so can take time ...).

The five sets of procedures are tested. That is to say:

- Functions on elements.
- Functions on polynomials.
- Functions on matrices.
- Functions on series.
- Functions on elliptic curves.

The procedures of this program are not in libzen.a but can be good examples of procedures written with ZEN functions. The same tests are performed for different finite rings following user options specified at the execution of the process. For a detailed description of these options, type bin/arch/zentest -h.

Appendix A

Installing ZEN

A.1 The principle

You probably got the ZEN package via anonymous ftp in a file probably named something like ZEN.x.y.tgz. The compilation of ZEN needs the package ZMAKE which should be available via anonymous ftp too. Then, what you have to do is:

1. To decompress ZEN.x.y.tgz at the same level as ZMAKE with
   $ gzip -d < ZEN.x.y.tgz |tar xvf - .
2. To go in the ZEN directory with
   $ cd ZEN.x.y.
3. To compile with
   $ make.
4. To get the documentation with
   $ make doc.

At the end of step 3, a library file libzen.a is compiled in lib/arch where arch depends on your system and the file zen.h is in include. Moreover, test executables are in bin/arch.

At the end of step 4, the user manual is in the file doc/dvi/doc.dvi. Moreover, you can possibly get the "advanced user manual" in the file doc/dvi/advanced.dvi after make adoc.

Figure A.1 enumerates the currently tested architectures. Usually, the operation sequence of figure A.2 will produce the library without problems.

Vendor	Processor	Operating system	Compilers
SUN	sparc	Solaris	cc,gcc
DEC	alpha	OSF	cc, gcc
HP	PA-RISC	HP-UX	cc, gcc
IBM	RS6000	AIX	cc, gcc
PC	486	linux	gcc
		Windows 95/98/2000/NT	gcc (cygnus)

Figure A.1: Portability of ZEN

A.2 Configuring the compiler options

It is possible to configure the compiler options you need for your convenience. The file used for that is *site.h* in zmake. It contains a few flags that can be set to YES or NO in order to activate or not the corresponding feature. This file is read at the very beginning of the compilation sequence by the zmake. Therefore, it should be edited before this command. The following are the compilation flags defined:

OptimizingCode	use the compiler optimizer, and skip some parts of testing code inside the library
DebuggingMalloc	use debugging features like -fbounds-checking or Purify.
ProfilingCode	use the compiler profiling options
UseAssembler	use the assembler opcodes whenever it's possible
UseLongLong	use the long long type if available
ExpandingNames	output filenames can indicate which of the options where used. This is useful especially if you want to keep a debugging and an optimized version of the library
HasPurify	If you intend to use Purify, you can raise this flag. Otherwise -fbounds-checking is used with gcc. This flag is only significant when DebuggingMalloc is raised.
HasOMP	If you intend to use compile C Open MP programs, you can raise this flag.
HasLargeTmp	large tmp is available

A.3 Customized memory allocation functions

Following a GMP mechanism, we allows customized allocation functions.

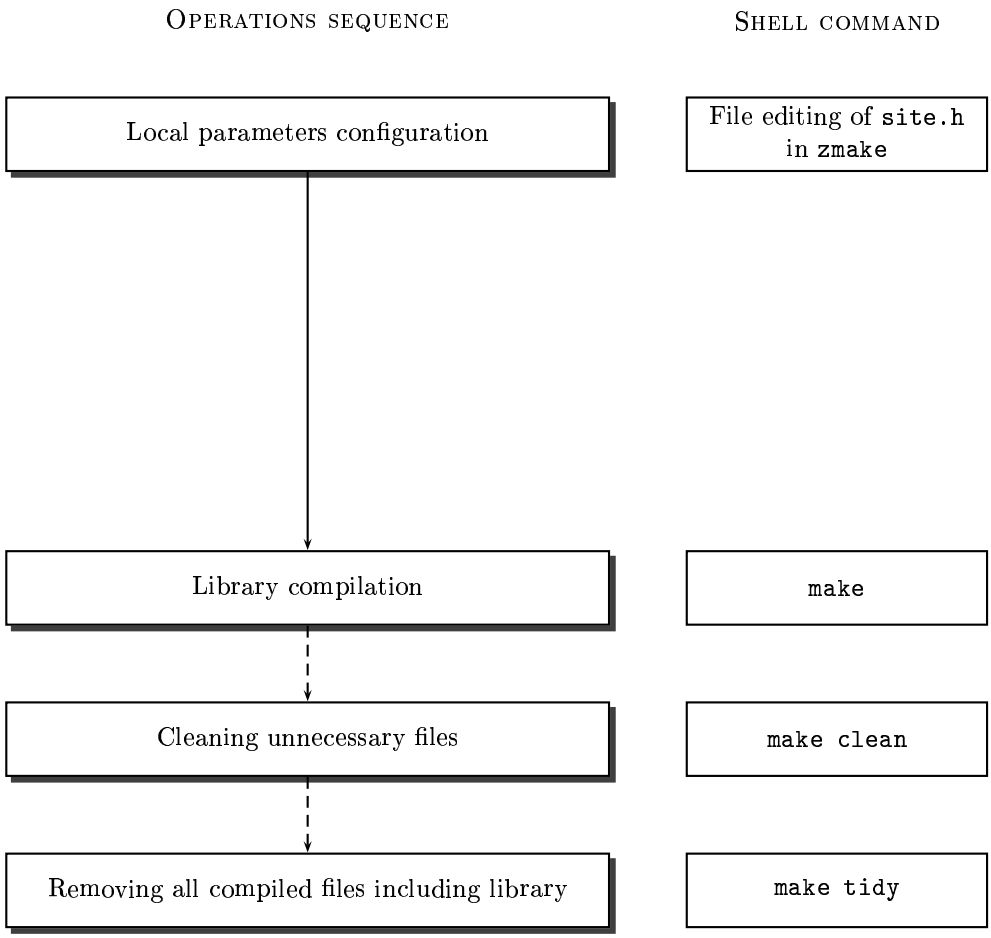

Figure A.2: Compilation of the library.

Procedure 319 *Allocating memory*

```
void * ZENMalloc(size)
   size_t size;
```

Input: *The size of memory to allocate*
Output: *A pointer on the allocated array or* NULL *if an error occured.*
Note: *This macro is used in the whole* ZEN *and* ZENFACT *libraries when memory allocation is needed. By default, a call to the native* malloc *function is performed. This can be overwritten by* ZENSetMemoryFunctions.

Procedure 320 *Reallocating memory*

```
void * ZENRealloc(oldptr,newsize)
   void * oldptr;
   size_t newsize;
```

Input: *The pointer to a previously allocated memory array, obtained by* ZENMalloc *, and the new size needed.*
Output: *A pointer on the newly allocated array, or* NULL *if an error occured.*
Side effect: *The values stored in the first array designed by* oldptr *are copied in the new array.*
Note: *This macro is used in the whole* ZEN *and* ZENFACT *libraries when memory reallocation is needed. By default, a call to the native* realloc *function is performed. This can be overwritten by* ZENSetMemoryFunctions.

Procedure 321 *Freeing memory*

```
void ZENFree(ptr)
   void * ptr;
```

Input: *A pointer to a previously allocated memory array, obtained by* ZENMalloc.
Side effect: *The memory pointed to by* ptr *is freed.*
Note: *This macro is used in the whole* ZEN *and* ZENFACT *libraries when freeing memory is needed. By default, a call to the native* free *function is performed. This can be overwritten by* ZENSetMemoryFunctions.

Procedure 322 *Setting custom memory allocation functions*

```
void ZENSetMemoryFunctions(ma,re,fr)
  void * (* ma) __((size_t));
  void * (* re) __((void *, size_t));
  void (* fr) __((void *));
```

Input: *The three memory allocation functions performing* malloc, realloc, *and* free.
Side effect: *The internal memory allocation functions are customized*
Note: *By default, the standard* malloc, realloc, *and* free *functions are used. This function should be called at the very beginning of a program.*

Appendix B

Bibliography

[1] F. Chabaud, Recherche de performance dans les corps finis – Applications à la crypotgraphie *Thèse de doctorat*, École Polytechnique, octobre 1996. `http://www.dmi.ens.fr/~chabaud/data/these.ps.gz`

[2] B. Serpette, J. Vuillemin, and J.C. Hervé. *BigNum: a portable and efficient package for arbitrary-precision arithmetic,* PRL Research Report #2, 1989. `ftp://ftp.digital.com/pub/DEC/PRL/research-reports/PRL-RR-2.ps.Z`

[3] T. Granlund. *The GNU Multiple Precision arithmetic library – 2.0.2*, `ftp://prep.ai.mit.edu/pub/gnu/gmp-M.N.tar.gz`

[4] R. Lercier. *Algorithmique des courbes elliptiques dans les corps finis*, PhD thesis, École Polytechnique 91128 Palaiseau France, Jun. 1997.

[5] F. Morain. Courbes elliptiques et tests de primalité. *Thèse de doctorat*, Univ. Claude Bernard – Lyon I, septembre 1990

Appendix C

Concepts index