ZEN

A new toolbox for computing in finite extensions of finite rings

ZENFACT

Factorization of polynomials

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 – Example

All the procedures described here form the libzenfact.a library. These procedures are all macros defined in zenfact.h. The installation procedure is the same as that for ZEN.

See the "user manual" of ZEN for more details on the following compilation procedure.

1. To decompress ZENFACT.x.y.tgz at the same level as ZMAKE with
   $ gzip -d < ZENFACT.x.y.tgz |tar xvf .
2. To go in the ZENFACT directory with
   $ cd ZENFACT.x.y.
3. To compile with
   $ make.
4. To get the documentation with
   $ make doc.

The purpose of ZENFACT is to provide functions that can be used in any ZENRing. These functions are generic and do not depend on the representation used. They constitute a good example of what can be done using ZEN.

We now give an example of program that use some features of ZENFACT, in order to obtain different representations of finite fields. This example can also be used to understand the use of precomputations and clones in ZEN.

The purpose of this program is to perform a bench test of low level computations in some finite fields, using different representations.

1.1 Preliminaries

The first thing to do is to include some standard libraries header files. We here include the ZENFACT header file, as the procedures to find polynomials in order to represent extensions are in this package.

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

```
#include "zenfact.h"
```

Contrary to our first example in ZEN's user manual, we here try to define every structure we use dynamically. Hence, we need some variables that we declare global for simplicity.

```
int DEG,EXT;
```

1.2 Measures of efficiency

Procedure 1 *Bench function*

```
void TestComputation(R)
   ZENRing R;
```

Input: *A* ZENRing.
Side effect: *We here perform our test which consists in computing $y = ax + b$ for a certain number of random variables.*

As the representation will differ, the number of tests must be large enough to decrease the variability of speed due to the repersentation of an element.

```
#ifndef NBTESTS
# define NBTESTS 1000
#endif
```

We use a table of ZENElt because we want to separate the allocation and definition of variables from operation itself.

```
  ZENElt Y,A[NBTESTS+2];
  int i;
  double tt;
```

We will use a sliding method to perform computation. The tests will be as follows:

$$\begin{array}{rcl} y_0 & = & a_0 a_1 + a_2 \\ y_1 & = & a_1 a_2 + a_3 \\ y_2 & = & a_2 a_3 + a_4 \\ & \vdots & \\ y_n & = & a_n a_{n+1} + a_{n+2} \end{array}$$

This method saves memory for computers that lacks some. We first define the operands and give time used by this operation.

```
  ZENEltAlloc(Y,R);
  tt=runtime();
  for(i=0;i<NBTESTS+2;i++)
    {
      ZENEltAlloc(A[i],R);
      ZENEltSetRandom(A[i],R);
    }
  tt=runtime()-tt;
  printf("%d allocations : %f ms CPU/allocation\n",i,tt/i);
```

We now perform the test itself.

```
  tt=runtime();
  for(i=0;i<NBTESTS;i++)
    {
      ZENEltMultiply(Y,A[i+1],A[i],R);
      ZENEltAdd(Y,A[i+2],R);
    }
  tt=runtime()-tt;
  printf("%d tests : %f ms CPU/test\n",i,tt/i);
```

We endly free the memory we used

```
  for(i=0;i<NBTESTS+2;i++)
    ZENEltFree(A[i],R);
  ZENEltFree(Y,R);
```

1.3 Building the finite fields

What seems interesting is to compare the speed of computations in fields of almost the same cardinality, mainly for a given prime p, $\mathbb{F}_p$ and $\mathbb{F}_{2^{\log_2 p}}$. We also need to compare the different representations provided by clones and see the impact of precomputations.

The beginning of the program is somehow now classical and should not need explanation.

```
int main()
{
  BigNumDigit q=2;
  BigNum p;
  BigNumLength pl;
  ZENRing R,E,E1,C1,E2,EC2;
  ZENPoly P,P2;
  ZENCln cln;
  ZENPrc prc;
  double tt;
  char *poly2;
```

```
tt=runtime();
                     As q is a BigNumDigit, hence &q is a BigNum
ZENBaseRingAlloc(R,&q,(BigNumLength)1);

tt=runtime()-tt;
printf("GF(2) built in %f s CPU\n",tt/1000.0);
```

We now ask interactively the size of number we want ($\log p$).

```
printf("Enter the logarithm of field size\n");
scanf("%d",&DEG);
```

1.3.1 Extensions of $\mathbb{F}_2$

1.3.1.1 Standard representation

We first try to define an extension using a truly random irreducible polynomial.

```
ZENPolyAlloc(P,DEG,R);
tt=runtime();
ZENPolySetRandomIrreducible(P,DEG,R);
tt=runtime()-tt;
printf("Irreducible polynomial found in %f s CPU\n",
       tt/1000.0);
printf("P(X)=");
ZENPolyPrintToFile(stdout,P,10,R);
printf("\n");
tt=runtime();
ZENExtRingAlloc(E,P,R);
tt=runtime()-tt;
printf("Extension built in %f s CPU\n",tt/1000.0);
ZENPolyFree(P,R);
```

We have our first ring, so we can perform our test.

```
TestComputation(E);
```

1.3.1.1.1 Precomputations

We now perform precomputations in order to speed operations. Performing all precomputations is certainly a waste of time in this case, but that's simpler. We first make precomputations on $\mathbb{F}_2$.

```
ZENPrcSetNone(prc);
ZENPrcSet(prc,ZENPRC_ELT_MULTIPLY);
ZENPrcSet(prc,ZENPRC_FINITE_FIELD);
tt=runtime();
ZENRingAddPrc(R,prc);
```

```
tt=runtime()-tt;
printf("Precomputations on GF(2) done in %f s CPU\n",tt/1000.0);

TestComputation(E);
```

Now, we can make precomputations on the extension and see the result, that should be more impressive.

```
tt=runtime();
ZENRingAddPrc(E,prc);
tt=runtime()-tt;
printf("Precomputations on GF(2^%d) done in %f s CPU\n",
       DEG,tt/1000.0);

TestComputation(E);
```

1.3.1.1.2 More efficient representations

We will now see the effect of using an irreducible polynomial of the form $X^d + f(X)$ for the representation of $\mathbb{F}_{2^d}$, with f of small degree.

```
ZENRingClose(E);

ZENPolyAlloc(P,DEG,R);
tt=runtime();
ZENPolySetGoodIrreducible(P,DEG,R);
tt=runtime()-tt;
printf("Suited irreducible polynomial found in %f s CPU\n",
       tt/1000.0);
printf("P(X)=");
ZENPolyPrintToFile(stdout,P,10,R);
printf("\n");
tt=runtime();
ZENExtRingAlloc(E,P,R);
tt=runtime()-tt;
printf("Extension built in %f s CPU\n",tt/1000.0);
ZENPolyFree(P,R);

TestComputation(E);
```

It is worthwile to perform precomputations even when the representation is optimal.

```
tt=runtime();
ZENRingAddPrc(E,prc);
tt=runtime()-tt;
printf("Precomputations for GF(2^%d) done in %f s CPU\n",
       DEG,tt/1000.0);
```

```
TestComputation(E);
```

1.3.1.2 Double extension

We now try another representation for this field, using a double extension, that is to say we build

$$\mathbb{F}_{2^e}[X]/_{Q(X)}$$

with $Q(X)$ of degree d/e. We compute the first extension using the best possible polynomial of the above form. Note that EXT is the value of e and is asked interactively.

```
ZENPolyAlloc(P,EXT,R);
tt=runtime();
ZENPolySetSmallestIrreducible(P,EXT,R);
tt=runtime()-tt;
printf("Best irreducible polynomial found in %f s CPU\n",
       tt/1000.0);
printf("P1(X)=");
ZENPolyPrintToFile(stdout,P,10,R);
printf("\n");
tt=runtime();
ZENExtRingAlloc(E1,P,R);
tt=runtime()-tt;
printf("Extension built in %f s CPU\n",tt/1000.0);
ZENPolyFree(P,R);
```

We here prepare also the optimization of this second representation, using clones. If e is small enough, then the tabulation of operations can be performed, which could possibly give faster operations for the double extension representation.

```
ZENClnSetAll(cln);
tt=runtime();
C1=ZENRingClone(E1,cln);
tt=runtime()-tt;
printf("Clone of GF(2^%d) done in %f s CPU\n",
       EXT,tt/1000.0);
```

We now build the second extension using a scheme now classical.

```
ZENPolyAlloc(P2,DEG/EXT,C1);
tt=runtime();
ZENPolySetRandomIrreducible(P2,DEG/EXT,C1);
tt=runtime()-tt;
printf("Irreducible polynomial found in %f s CPU\n",
       tt/1000.0);
```

We here use a string representation of the polynomial because the internal representation will be different in the clone.

```
poly2=ZENPolyPrintToString(P2,10,C1);
printf("P2(X)= %s\n",poly2);
ZENPolyFree(P2,C1);
```

We first use the normal representation of $\mathbb{F}_{2^e}$.

```
ZENPolyReadFromString(P2,poly2,10,E1);
tt=runtime();
ZENExtRingAlloc(E2,P2,E1);
tt=runtime()-tt;
ZENPolyFree(P2,E1);
printf("Extension built in %f s CPU\n",tt/1000.0);

TestComputation(E2);
ZENRingClose(E2);
```

And now we use the clone as basis for our extension.

```
ZENPolyReadFromString(P2,poly2,10,C1);
tt=runtime();
ZENExtRingAlloc(EC2,P2,C1);
tt=runtime()-tt;
printf("Extension on clone built in %f s CPU\n",
       tt/1000.0);

TestComputation(EC2);
```

As this clearly will be the fastest representation for double extension, we take the time to perform precomputations on both the clone and the second extension itself.

```
tt=runtime();
ZENRingAddPrc(C1,prc);
tt=runtime()-tt;
printf("Precomputations on clone of GF(2^%d) done in %f s CPU\n",
       EXT,tt/1000.0);

TestComputation(EC2);

tt=runtime();
ZENRingAddPrc(EC2,prc);
tt=runtime()-tt;
printf("Precomputations on GF(2^%d) done in %f s CPU\n",
       DEG,tt/1000.0);

TestComputation(EC2);
ZENRingClose(EC2);
```

We continue to test different representations, using good polynomials. As this is always based on the same schemes, we skip this part of the program in the documentation.

1.3.2 Modular field $\mathbb{F}_p$

1.3.2.1 Standard representation

We now find a random prime of the good size.

```
tt=runtime();
pl=divSizeBloc(DEG-1)+1;          gives the number of digits
p=ZBNC(pl);
ZBNSetRandom(p,pl);
p[pl-1] |= BlocExp2(modSizeBloc(DEG-1));
                                  ensure the good number of bits
ZBNNextPrime(&p,&pl);
tt=runtime()-tt;
ZBNPrintToFile(stdout,p,pl,10);
printf("\nPrime of %d bits found in %f s CPU\n",DEG,tt/1000.0);
```

We don't test whether the number of bits has changed, because for large numbers, the probability to have a carry is small. We can now build the modular field $\mathbb{F}_p = \mathbb{Z}/_{p\mathbb{Z}}$.

```
ZENBaseRingAlloc(R,p,pl);
TestComputation(R);
```

We try to speed up computations using precomputations.

```
tt=runtime();
ZENRingAddPrc(R,prc);
tt=runtime()-tt;
printf("Precomputations on GF(p) done in %f s CPU\n",
       tt/1000.0);

TestComputation(R);
```

1.3.2.2 Montgomery's representation

We can now try to clone the field. If the prime p is large, this will result in the use of Montgomery's representation.

```
tt=runtime();
C1=ZENRingClone(R,cln);
tt=runtime()-tt;
printf("Clone of GF(p) done in %f s CPU\n",tt/1000.0);
```

```
TestComputation(C1);
```

We also need to use precomputations for the clone, because they are not done by default for the clone, even if they were done for the initial ring.

```
tt=runtime();
ZENRingAddPrc(C1,prc);
tt=runtime()-tt;
printf("Precomputations on the clone of GF(p) done in %f s CPU\n",
       tt/1000.0);

TestComputation(C1);
```

1.4 Compilation

The compilation of this program is done by the standard make command of zenfact. The options used can give you hints for optimizing your programms on your architecture.

1.5 Results

We here give an example of what is obtained by the above program. The results were obtained on a PC running Linux on a P100 with 16 MO RAM, using GMP with the raised flags HasGcc, and OptimizingCode. They are gathered in the following table.

Extension	$\mathbb{F}_2$	$\mathbb{F}_{2^{1024}}$	CPU time	
			precomputations (s)	operation (ms)
simple	–	–	–	9.6
	P	–	–	9.6
	P	P	0.21	9.4
	P	S	–	6.8
	P	SP	0.26	6.7
double	$\mathbb{F}_{2^8}$	$\mathbb{F}_{2^{1024}}$		
	S	–	–	419
	C	–	7.5	26
	CP	–	7.5	26
	CP	P	7.9	26
	S	S	–	98
	CP	S	7.5	12
	CP	SP	7.8	12
	$\mathbb{F}_{2^{16}}$	$\mathbb{F}_{2^{1024}}$		
	S	–	–	128
	C	–	43	9.8
	CP	–	43	9.8
	CP	P	43	9.8
	S	S	–	41
	CP	S	43	5.2
	CP	SP	43	5.2
Modular field	$\mathbb{F}_p$			
$\log p = 1024$			precomputations (s)	operation (ms)
	–		–	0.45
	P		–	0.45
	C		–	0.45
	C		–	0.45

In this table, P stands for the case in which precomputations are done, S states that an irreducible polynomial of low sub-degree is used, C indicates a cloning of the ring. The precomputations time don't take in account the computations needed to find the definition polynomials. The indicative times for a standard execution are as follows:

- random irreducible polynomial over $\mathbb{F}_2$ of degree 1024 found in 100 s.
- irreducible polynomial over $\mathbb{F}_2$ of degree 1024 of small subdegree found in 59 s[1].
- irreducible polynomial over $\mathbb{F}_2$ of degree 8 of smallest subdegree found in 0.01 s[2].
- random irreducible polynomial over $\mathbb{F}_{2^8}$ of degree 128 found in 625 s.
- irreducible polynomial over $\mathbb{F}_{2^8}$ of degree 128 of small subdegree found in 16 s.

[1]The result was $X^{1024} + X^9 + X^8 + X^7 + X^5 + X + 1$.
[2]The result was $X^8 + X^4 + X^3 + X + 1$.

- irreducible polynomial over $\mathbb{F}_2$ of degree 16 of smallest subdegree found in 0.01 s[3].
- irreducible polynomial over $\mathbb{F}_{2^{16}}$ of degree 64 found in 5 s.
- irreducible polynomial over $\mathbb{F}_{2^{16}}$ of degree 64 of small subdegree found in 6 s.
- random prime of 1024 bits found in 166 s[4].

1.5.1 Simple extension

Our reference will be standard representation of $\mathbb{F}_{2^{1024}}$ using a random irreducible polynomial $P(X)$ of degree 1024. In this case, initialization is negligible, and multiplying two elements takes about 10 ms. Precomputations do not improve a lot this result..

On the contrary, using a special polynom $P(X) = X^{1024} + f(X)$, we obtain a gain of about 25%. Precomputing further improves this result to reach an overall gain of 33%. Thus, one obtain a final CPU time of about 7 ms.

1.5.2 Double extension

1.5.2.1 First extension of degree 8

When using a double extension with a first extension of degree 8, one can notice that using a small subdegree polynomial and cloning of the first extension results in a huge gain. Cloning improves performance by a factor 16, whereas using a special polynomial gives a factor 4. These two factors are not fully cumulative but the overall gain which is obtained is 34.

1.5.2.2 First extension of degree 16

When using a double extension with a first extension of degree 16, one can notice the same behaviour as above. Cloning improves performance by a factor 13, and using a special polynomial gives a factor 3. The overall gain obtained reaches a factor 25. The final result is better than the standard representation using a special polynomial by a 25% gain. This confirms a recent result [3]. However, this result is fully computer dependant. For instance, using a Sparc station, one will find that the standard representation may be the best (see [1, 2]). Hence, optimizing the representation of a finite field will depend on the application and the computer used.

Using ZEN, it will be very easy to optimize the performance, as the program will remain the same. The only thing to change will be the construction of the ZENRings.

[3]The result was $X^{16} + X^5 + X^3 + X + 1$.

[4]The result was 10181171201750041960330048543588830430025225685504787771517496717186373087920038507546238347795500519

Chapter 2

Enumerating the functions of `ZENFACT`

2.1 Roots of polynomials.

These functions aim at finding the roots of any polynomial P. The main function is ZENPolyRoots that calls the function ZENPolyRootsBerlekamp if its degree is greater than 2 or ZENPolyRootsDegree2 otherwise.

Procedure 2 *Roots of any polynomial.*

```
int ZENPolyRoots(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 degree(*P*)-1, the coefficients of which are the roots of* P.

Note: roots *must be allocated for at least degree of* P *minus one. We must have* P $\neq$ roots. *Valid only in finite fields.*

Procedure 3 *Roots of a polynomial whose degree is equal to its number of roots.*

```
int ZENPolyRootsEDF(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 degree(*P*)-1, the coefficients of which are the roots of* P.

Note: roots *must be allocated for at least degree of* P *minus one. We must have* P $\neq$ roots. *Valid only in finite fields.*

Procedure 4 *Roots of any polynomial in finite fields of even characteristic.*

```
int ZENPolyRootsBerlekampEvenChar(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 degree(*P*)-1, the coefficients of which are the roots of* P.

Note: roots *must be allocated for at least degree of* P *minus one. We must have* P $\neq$ roots. *Valid only in finite fields.*

Procedure 5 *Roots of any polynomial in finite fields of odd characteristic.*

```
int ZENPolyRootsBerlekampOddChar(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 degree(*P*)-1, the coefficients of which are the roots of* P.
Note: roots *must be allocated for at least degree of* P *minus one. We must have* P $\neq$ roots. *Valid only in finite fields.*

Procedure 6 *A root of a polynomial whose degree is equal to its number of roots.*

```
int ZENPolyOneRootEDF(root, P, Rg)
   ZENElt root;
   ZENPoly P;
   ZENRing Rg;
```

Input: *A* ZENPoly P *and an element* root *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: root *contains a root of* P
Note: *Valid only in finite fields.*

Procedure 7 *A root of a polynomial whose degree is equal to its number of roots in finite fields of even characteristic.*

```
int ZENPolyOneRootBerlekampEvenChar(root, P, Rg)
   ZENElt root;
   ZENPoly P;
   ZENRing Rg;
```

Input: *A* ZENPoly P *and an element* root *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: root *contains a root of* P
Note: *Valid only in finite fields.*

Procedure 8 *A root of a polynomial whose degree is equal to its number of roots in finite fields of odd characteristic.*

```
int ZENPolyOneRootBerlekampOddChar(root, P, Rg)
   ZENElt root;
   ZENPoly P;
   ZENRing Rg;
```

Input: *A* ZENPoly P *and an element* root *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: root *contains a root of* P
Note: *Valid only in finite fields.*

.

Procedure 9 *d-roots in finite fields*

```
int ZENEltDRoot(R, d, A, Rg)
   BigNumDigit d;
   ZENElt R, A;
   ZENRing Rg;
```

Input: *Two elements* R *and* A *of a finite field* Rg.
Output: ZENERR *if an error occurred, 0 if* A *is a d-root, 1 otherwise.*
Side effect: R *is filled with the d-root of* A *if 0 is returned.*
Note: *This procedure is valid only in finite fields.*

Procedure 10 *Is a d-root in finite fields.*

```
int ZENEltIsADRoot(d, A, Rg)
   BigNumDigit d;
   ZENElt A;
   ZENRing Rg;
```

Input: *An element* A *of a finite ring* Rg.
Output: ZENERR *if an error occurred, 1 if* A *is a d-root, 0 otherwise.*
Note: *This procedure is valid only in finite fields. The algorithm uses an exponantiation.*

2.2 Polynomial factorization.

All the following procedures to factorize a polynomial in $GF(2^n)$ will return a list of polynomials. After describing functions to manipulate such lists, we give the procedure to factorize polynomials.

A list of polynomial is a pointer on a struct which contains a polynomial, its multiplicity and a pointer on an other ceil of the list. We can append a polynomial at the beginning of a list with Gf2PolyListAppendFirst or remove a polynomial at the beginning of the list Gf2PolyListAppendFirst.

Procedure 11 *Creating a list of polynomials.*

```
void ZENPolyFactCreate(lfact, Rg)
   ZENPolyFact lfact;
   ZENRing Rg;
```

Input: *A list of polynomials* lfact.
Side effect: lfact *is allocated*

Procedure 12 *Freeing a list of polynomials.*

```
void ZENPolyFactFree(lfact, Rg)
   ZENPolyFact lfact;
   ZENRing Rg;
```

Input: *A list of polynomials* lfact.
Side effect: lfact *is freed*

Procedure 13 *Freeing a polynomial structure.*

```
void ZENPolyListFree(list, Rg)
   ZENPolyList list;
   ZENRing Rg;
```

Input: *A polynomial structure* list.
Side effect: list *is allocated*

---

pointer to P.

Procedure 14 *Insert a polynomial into a list.*

```
int ZENPolyFactAppend(lfact, list, Rg)
   ZENPolyFact lfact;
   ZENPolyList list;
   ZENRing Rg;
```

Input: *A polynomial* P *and a multiplicity* mult.
Side effect: *(*P, mult*) is inserted in* lfact.

Procedure 15 *Removing a polynomial of a list.*

```
int ZENPolyFactRemove(list, lfact, index, Rg)
   ZENPolyList list;
   ZENPolyFact lfact;
   int index;
   ZENRing Rg;
```

Input: *A list* lfact *and a pointer on a multiplicity* p_mult *and a polynomial* p_P.
Output: *-1 if* index *is greater than the size of the list, 0 otherwise.*
Side effect: *The list is updated and* $\star$p_P, $\star$p_m *are initialized if no error occured.*

Three main steps must be done to factorize a polynomial P:

- The square free step done by ZENPolySFFactor,
- The distict degree step done by ZENPolyDDFactor,
- The equal degree step done by ZENPolyEDFactor.

Procedure 16 *Factorization of a polynomial.*

```
int ZENPolyFactor(lfact, P, Rg)
   ZENPolyFact lfact;
   ZENPoly P;
   ZENRing Rg;
```

Input: *A polynomial* P *to factorize.*
Side effect: *The factors of* P *are put in* p_lfact.
Note: .

The square free factorization of a polynomial $P(X)$ is a decomposition $P = P_1 P_2^2 P_3^3 \dots P_k^k$, with $P_1, \dots, P_k$ square free and $(P_i, P_j) = 1$. The following algorithm is described in [4].

Procedure 17 *Square free factorization of a polynomial.*

```
int ZENPolySFFactor(lfact, P, Rg)
   ZENPolyFact lfact;
   ZENPoly P;
   ZENRing Rg;
```

Input: *A polynomial* P *to factorize.*
Side effect: *The factors are put in* lfact.

Once got a square free polynomial $P(X)$, we split the polynomial $P(X)$ in polynomaials of distinct degrees. This is the distinct degree factorization step. In fact, we get the factor of degree d of $P(X)$ by $\gcd(X^{2^{nd}} - X, P(X))$. Each factor is then completely factorized by ZENPolyEDFactor if the flag edf equals to 1. This function stops once a factor is found if the flag of is set.

Procedure 18 *Distinct degree factorization of a polynomial.*

```
int ZENPolyDDFactor(lfact, P, Rg)
   ZENPolyFact lfact;
   ZENPoly P;
   ZENRing Rg;
```

Input: *A polynomial* P *to factorize.*
Side effect: *The factors are put in* $\star$p_lfact.

To split a polynomial $P(X)$ in factors of degree k, we use a algorithm due to Thion Ly [6].

Procedure 19 *Equal degree polynomial factorization.*

```
int ZENPolyEDFactor(lfact, P, XqmodP, k, Rg)
   ZENPolyFact lfact;
   ZENElt P, XqmodP;
   int k;
   ZENRing Rg;
```

Input: *A polynomial* P *to factorize, a polynomial* XqmodP *equal to* $X^{2^n} \bmod P(X)$*, the degree* k *of the factors of* $P(X)$*.*

Side effect: *The factors are put in* $\star$p_lfact*.*

The Thiong Ly's algorithm first find a i such that $TX^i = X^i + \ldots + X^i$ is different than a constante modulo $P(X)$ and then a j such that $u(X) = Tr(T^jTX^i)$ is different than a constant modulo $P(X)$. Finally $\gcd(u(X), P(X))$ is a non trivial factor of $P(X)$.

Procedure 20 .

```
ZENPoly ZENPolyThiongLy(P, XqmodP, k, Rg)
   ZENPoly P, XqmodP;
   int k;
   ZENRing Rg;
```

Input: *A polynomial* P *to factorize,* $X^{2^n} \bmod P(X)$ *and* k*, the degree of the factors of* P*.*

Output: *A factor of* P *is returned.*

Procedure 21 .

```
ZENPoly ZENPolyCamion(P, k, Rg)
   ZENPoly P;
   int k;
   ZENRing Rg;
```

Input: *A polynomial* P *to factorize in a finite field of odd characteristic,* $X^{2^n} \bmod P(X)$ *and* k*, the degree of the factors of* P*.*

Output: *A factor of* P *is returned.*

A recursive algorithm due to Shoop [?] computes traces of polynomial faster.

Procedure 22 *Computing* $Q(X) + Q(X)^{2^n} + ... + Q(X)^(2^n(k-1))$ mod $P(X)$.

```
int ZENShoopTrace(T, Q, k, XqmodP, KRg)
   ZENElt T, Q, XqmodP;
   int k;
   ZENRing KRg;
```

Input: *The polynomial* Q, *the modulo* P, XqmodP$=X^{2^n}$ mod $P(X)$ *and the integer* k.

Side effect: *The trace is put in* T.

2.3 Irreducible polynomials.

Procedure 23 *Testing irreducibility of a polynomial*

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

Input: *A polynomial* PX *defined over a finite ring* Rg.

Output:

ZENERR *if an error occured,*

ZEN_NO_INVERSE *if an inverse was impossible to compute,*

ZENPOLY_HAS_ZERO_ROOT *if the polynomial has zero as root,*

ZENPOLY_HAS_ONE_ROOT *if the polynomial has one as root,*

ZENPOLY_HAS_REPEATED_FACTORS *if the polynomial has repeated factors,*

ZENPOLY_IS_P_POWER *if* P$(X) = g(X)^p$, *with* p *the characteristic of the* ZENRing,

ZENPOLY_IS_IRREDUCIBLE *if the polynomial is irreducible,*

ZENPOLY_IS_COMPOSED *otherwise.*

Note: *Valid only in finite fields. The Berlekamp's algorithm is used*

2.3.1 Low-level irreducibility functions

Two algorithms are implemented to test irreducibility of a polynomial. DDF algorithm depends a lot on the shape of the polynomial. Berlekamp's algorithm is usually faster for small degree polynomials, and its behaviour is much more regular. For this reason, Berlekamp's algorithm is used in all the high level functions of the library. It is up to the user to use _ZENPolyIsNotIrreducible with DDF or any other algorithm he might want to use.

Procedure 24 *Testing irreducibility of a polynomial*

```
int _ZENPolyIsNotIrreducible(P,Algo,R)
   ZENPoly P;
   int (*Algo) ___((ZENPoly, ZENRing));
   ZENRing R;
```

Input: *A polynomial* PX *defined over a finite ring* Rg*, and a function pointer on the algorithm to use (*Berlekamp *and* DDF *are available).*

Output:

ZENERR *if an error occured,*

ZEN_NO_INVERSE *if an inverse was impossible to compute,*

ZENPOLY_HAS_ZERO_ROOT *if the polynomial has zero as root,*

ZENPOLY_HAS_ONE_ROOT *if the polynomial has one as root,*

ZENPOLY_HAS_REPEATED_FACTORS *if the polynomial has repeated factors,*

ZENPOLY_IS_P_POWER *if* $\mathsf{P}(X) = g(X)^p$*, with* p *the characteristic of the* ZENRing*,*

ZENPOLY_IS_IRREDUCIBLE *if the polynomial is irreducible,*

ZENPOLY_IS_COMPOSED *otherwise.*

Note: *This function performs what is usually called the ERF phase (Elimination of Repeated Factors) of a polynomial factorization. This is done before calling the indicated algorithm to complete a DDF phase (Distinct-Degree Factorization). All these operations stop as soon as the polynomial is found composed. The complete factorization is therefore not done.*

Procedure 25 *Distinct-Degree Factorization*

```
int Berlekamp(P,R)
   ZENPoly P;
   ZENRing R;
```

Input: *A* ZENPoly *defined over a* ZENRing*, that must have no repeated factors.*

Output:

ZENERR *if an error occured,*

ZEN_NO_INVERSE *if an inverse was impossible to compute,*

ZENPOLY_IS_IRREDUCIBLE *if the polynomial is irreducible,*

ZENPOLY_IS_COMPOSED *otherwise.*

Note: *[description of the algorithm]. We add two tests to this algorithm, which correspond to the first two iterations of* DDF *algorithm. This gives the best results.*

Procedure 26 *Distinct-Degree Factorization*

```
int DDF(P,R)
   ZENPoly P;
   ZENRing R;
```

Input: *A* ZENPoly *defined over a* ZENRing*, that must have no repeated factors.*

Output:

ZENERR *if an error occured,*

ZEN_NO_INVERSE *if an inverse was impossible to compute,*

ZENPOLY_IS_IRREDUCIBLE *if the polynomial is irreducible,*

ZENPOLY_IS_COMPOSED *otherwise.*

Note: *[description of the algorithm]*

We wrote two functions to find irreducible polynomials.

Procedure 27 *Setting a polynomial to an irreducible random one*

```
int ZENPolySetRandomIrreducible(P,deg,R)
   ZENPoly P;
   int deg;
   ZENRing R;
```

Input: *A polynomial, a degree, a* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The polynomial is set randomly to a monic polynomial of given degree that is irreducible over* R.
Note: *This procedure may take a while. It is valid only in finite fields. The Berlekamp's algorithm is used.*

Procedure 28 *Finding the smallest polynomial $f(X)$ such that $X^d + f(X)$ is an irreducible polynomial.*

```
int ZENPolySetSmallestIrreducible(P,deg,R)
   ZENPoly P;
   int deg;
   ZENRing R;
```

Input: *A degree* d *and an allocated polynomial* P *of size at least* d *of a* ZENRing R.
Output: *-1 if an error occured, 0 otherwise*
Side effect: *The polynomial is set to a polynomial of given degree that is irreducible over* R
Note: *This procedure may take a while. It uses Berlekamp's algorithm.*

Procedure 29 *Finding a small random polynomial $f(X)$ such that $X^d + f(X)$ is an irreducible polynomial.*

```
int ZENPolySetGoodIrreducible(P,deg,R)
   ZENPoly P;
   int deg;
   ZENRing R;
```

Input: *A degree* d *and an allocated polynomial* P *of size at least* d *of a* ZENRing R.
Output: *-1 if an error occured, 0 otherwise*
Side effect: *The polynomial is set to a polynomial of given degree that is irreducible over* R
Note: *This procedure may take a while. It uses Berlekamp's algorithm.*

Procedure 30 *Setting a polynomial to an irreducible random one*

```
int _ZENPolySetRandomIrreducible(P,deg,Algo,R)
   ZENPoly P;
   int deg;
   int (*Algo) ___((ZENPoly, ZENRing));
   ZENRing R;
```

Input: *A polynomial, a degree, a* ZENRing*, and a function pointer on the algorithm to use (*Berlekamp *and* DDF *are available: see* ZENPolyIsIrreducible*).*
Output: ZENERR *if an error occured, 0 otherwise*
Side effect: *The polynomial is set randomly to a monic polynomial of given degree that is irreducible over* R
Note: *This procedure may take a while*

Procedure 31 *Finding the smallest polynomial $f(X)$ such that $X^d + f(X)$ is an irreducible polynomial.*

```
int _ZENPolySetSmallestIrreducible(P,deg,Algo,R)
   ZENPoly P;
   int deg;
   int (*Algo) ___((ZENPoly, ZENRing));
   ZENRing R;
```

Input: *A degree* d*, an allocated polynomial* P *of size at least* d *of a* ZENRing R*, and a function pointer on the algorithm to use (*Berlekamp *and* DDF *are available: see* ZENPolyIsIrreducible*).*
Output: *-1 if an error occured, 0 otherwise*
Side effect: *The polynomial is set to a polynomial of given degree that is irreducible over* R
Note: *This procedure may take a while.*

Procedure 32 *Finding a small random polynomial $f(X)$ such that $X^d + f(X)$ is an irreducible polynomial.*

```
int _ZENPolySetGoodIrreducible(P,deg,Algo,R)
   ZENPoly P;
   int deg;
   int (*Algo) ___((ZENPoly, ZENRing));
   ZENRing R;
```

Input: *A degree* d*, an allocated polynomial* P *of size at least* d *of a* ZENRing R*, and a function pointer on the algorithm to use (*Berlekamp *and* DDF *are available: see* ZENPolyIsIrreducible*).*
Output: *-1 if an error occured, 0 otherwise*
Side effect: *The polynomial is set to a polynomial of given degree that is irreducible over* R
Note: *This procedure may take a while.*

2.4 Pseudo-prime integers.

For integers, only a pseudo primality test based on a Miller Rabin algorithm was implemented.

Procedure 33 *Miller Rabin test*

```
int ZENMillerRabin(Rg, e)
   ZENRing Rg;
   BigNumDigit e;
```

Input: *A finite ring* Rg *and a base* e*.*
Output: ZENERR *if an error occured, 1 if* Rg *is a strong finite field in base* e*, 0 otherwise.*
Note: *The first integer which is a strong pseudo prime for* e $= 2$*, 3, 5, 7, and 11, but not a prime is larger than* 2510^{12}

The number of bases used in the Miller-Rabin test can be adjusted in the file psprime.c.

```
#ifndef NB_BASIS_MILLER_RABIN
# define NB_BASIS_MILLER_RABIN 39  maximal number is 39
#endif
  int base[40];
  base[0]=2; base[1]=3; base[2]=5; base[3]=7; base[4]=11;
  base[5]=13;base[6]=17;base[7]=19;base[8]=23;base[9]=29;
  base[10]=31;base[11]=37;base[12]=41;base[13]=43;base[14]=47;
  base[15]=53;base[16]=59;base[17]=61;base[18]=67;base[19]=71;
  base[20]=73; base[21]=79; base[22]=83; base[23]=89; base[24]=97;
```

```
base[25]=101;base[26]=103;base[27]=107;base[28]=109;base[29]=113;
base[30]=127;base[31]=131;base[32]=137;base[33]=139;base[34]=149;
base[35]=151;base[36]=157;base[37]=163;base[38]=167;base[39]=173;
```

Procedure 34 *Is it a pseudo finite field.*

```
int ZENRingIsAPseudoFF(Rg)
    ZENRing Rg;
```

Input: *A finite ring* Rg.

Output: ZENERR *if an error occurred, 1 if* Rg *is a pseudo finite field, 0 otherwise.*

Note: *This procedure test if the definition ring is a finite field and then, if necessary, if the definition polynomial is irreducible. When* Rg *is a prime field, this procedure test if it is a field with the Miller Rabin test applied to several basis. With 2, 3, 5, 7, 11 and 13, the answer is exact for all* $n < 10^{12}$

Procedure 35 *Is an integer prime*

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

Input: *A* BigNum *and its length.*

Output: ZENERR *if an error occurs, 1 if the integer is pseudo-prime, 0 otherwise*

Note: *The test uses* ZENRingIsAPseudoFF

Procedure 36 *Find the prime immediatly following*

```
int ZBNNextPrime(p_n,p_nl)
    BigNum *p_n;
    BigNumLength *p_nl;
```

Input: *A pointer to an allocated* BigNum *and a pointer to its length.*

Output: ZENERR *if an error occurs, the number of steps performed otherwise*

Side effect: *The* BigNum *is modified accordingly. If a reallocation is needed, it is performed after freeing the initial* BigNum.

Note: *The test uses* ZBNIsPrime. *If the number is already prime, it is unchanged.*

2.5 Finite field construction

We provide high-level functions to easily build a representation of any finite field. This representation is not always the most efficient, but compromise should be good enough for most purposes.

Procedure 37 *Generating a finite field*

```
int ZENFieldAlloc(R,q,ql,d)
   ZENRing R;
   BigNum q;
   BigNumLength ql;
   int d;
```

Input: *A* ZENRing *to build, a number* (q,ql) *and the degree* d *of the extension.*

Output: ZENErr *if an error occured, 0 otherwise.*

Side effect: *The* ZENRing

$$R = \mathbb{Z}/_{p\mathbb{Z}}[X]/_{P(X)}$$

is built using an irreducible polynomial of degree d *generated by* ZENPolySetGoodIrreducible. *The prime p is the smallest prime greater or equal than* (q,ql).

Note: *No precomputation is done. If* d< 2 *the built field is* $R = \mathbb{Z}/_{p\mathbb{Z}}$. *The base ring is cloned, in order to improve efficiency, using* ZENClnSetAll *and* ZENRingClone. *The extension is cloned in the same way. The obtained structure can be freed using* ZENRingFullClose.

Procedure 38 *Generating interactively a finite ring*

```
ZENRing ZENRingDefine(IN,OUT)
   FILE *IN,*OUT;
```

Input: *Two file descriptors that can be* stdin *and* stdout.

Output: ZENNULL *if an unrecoverable error occured or the newly defined* ZENRing.

Side effect: *All the possible clonings are performed, but no precomputation is done.*

Note: *If* OUT *is set to* NULL, *the interactive questions are not output.*

Procedure 39 *Precomputing.*

```
int ZENRingRecursiveAddPrc(R, prc)
   ZENRing R;
   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. These precomputations are done recursively for all the underlying* ZENRing*s. If a clone is encountered, precomputations are added to the clone, not to the original ring, because its operations are not used.*
Note: *Precomputations can take time... This function can be used for instance on the output of a* _ZENRingDefine *call.*

Procedure 40 *Suppressing precomputations.*

```
void ZENRingRecursiveRmPrc(R, prc)
   ZENRing R;
   ZENPrc prc;
```

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

Procedure 41 *Writing to file in a human comprehensive representation.*

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

Input: *A stream* file*, the level of ring, an allocated* ZENRing Rg.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Printing a representation of* Rg *to* file.

Procedure 42 *Printing to stderr.*

```
int ZRP(R)
  ZENRing R;
```

Input: *An allocated* ZENRing.
Output: ZENERR *if an error occured, 0 otherwise.*
Side effect: *Printing a representation of* R *to* stderr.

Appendix A

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] F. Chabaud, and R. Lercier, Representations of $GF(2^n)$ – An example using ZEN library,
`http://lix.polytechnique.fr/~zen/example.html`

[3] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem and J. Vandewalle. *A fast software implementation for arithmetic operations in* $\mathbb{F}_{2^n}$, Advances in cryptology, *ASIACRYPT'96,* Springer-Verlag, *LNCS* 1163, 65–76, 1996.

[4] K. Geddes, S. Czapor, and G. Labahn

[5] Rudolf Lidl, and Harald Niederreiter. *Finite fields.* Encyclopedia of mathematics and its applications. Addison-Wesley Publising Company, 1983.

[6] A.Thiong ly. A deterministic algorithm for factorizing polynomials over extensions $GF(p^m)$ of $GF(p)$, p a small prime. *J. of Information & Optimization Sciences*, Vol. 10(2): 337–344, 1989.

Appendix B

Concepts index