

**Contents**

[Security Advisory](#)  
[Project Wiki/Documentation](#)  
[Benchmarks and Subs](#)  
[CertiVox Standard Curves](#)  
[Elliptic Curves](#)  
[IEEE 1363](#)  
[MIRACL Reference Manual](#)  
[MIRACL SDK Licencing](#)  
[MIRACL User's Manual](#)  
[Source Code Access](#)  
[Support Mailing List](#)

**Attachments**

## MIRACL Reference Manual

---

- [1 MIRACL Reference Manual](#)
- [2 MIRACL Module Documentation](#)
  - [2.1 Low level routines](#)
  - [2.2 Advanced arithmetic routines](#)
  - [2.3 Montgomery arithmetic routines](#)
  - [2.4 ZZn2 arithmetic routines](#)
  - [2.5 Encryption routines](#)
  - [2.6 Elliptic curve routines](#)
  - [2.7 Floating-slash routines](#)
- [3 MIRACL Data Structure Documentation](#)
  - [3.1 MIRACL Structure Reference](#)

### 1 MIRACL Reference Manual

Note: In these routines a big parameter can also be used wherever a flash is specified, but not viceversa. Further information may be gleaned from the (lightly) commented source code. An asterisk after the name indicates that the function does not take a mip parameter if MR\_GENERIC\_MT is defined in mirdef.h.

- [Low-level routines](#)
- [Advanced arithmetic routines](#)
- [Montgomery arithmetic routines](#)
- [ZZn2 arithmetic routines](#)
- [Elliptic curve routines](#)
- [Encryption routines](#)
- [Floating-slash routines](#)

### 2 MIRACL Module Documentation

#### 2.1 Low level routines

##### Functions

- void absol\* (flash x, flash y)
- void add (big x, big y, big z)
- int big\_to\_bytes (int max, big x, char (\*ptr, BOOL justify)
- void bigbits (int n, big x)
- mr\_small brand (void)
- void bytes\_to\_big (int len, char \*ptr, big x)
- int cinnum (flash x, FILE \*filep)
- int cinstr (flash x, char \*string)
- int compare\* (big x, big y)
- void convert (int n, big x)
- void copy\* (flash x, flash y)
- int cotnum (flash x, FILE \*filep)
- int cotstr (flash x, char \*string)
- void decr (big x, int n, big z)
- void divide (big x, big y, big z)
- BOOL divisible (big x, big y)
- void \* ecp\_memalloc (int num)
- void ecp\_memkill (char \*mem, int num)
- int exsign\* (flash x)
- miracl \* get\_mip ()
- int getdig (big x, int i)
- unsigned int igcd\* (unsigned int x, unsigned int y)
- void incr (big x, int n, big z)
- BOOL init\_big\_from\_rom (big x, int len, const mr\_small \*rom, int romsize, int \*romptr)
- BOOL init\_point\_from\_rom (epoint \*P, int len, const mr\_small \*rom, int romsize, int \*romptr)
- int innum (flash x, FILE \*filep)

- void insign\* (int s, flash x)
- int instr (flash x, char \*string)
- void irand (mr\_unsign32 seed)
- void lgconv (long n, big x)
- void mad (big x, big y, big z, big w, big q, big r)
- void \* memalloc (int num)
- void memkill (char \*mem, int len)
- void mirexit (void)
- void mirkill\* (big x)
- miracl \* mirsys (int nd, mr\_small nb)
- flash mirvar (int iv)
- flash mirvar\_mem (char \*mem, int index)
- void multiply (big x, big y, big z)
- void negify\* (flash x, flash y)
- mr\_small normalise (big x, big y)
- int numdig (big x)
- int otum (flash x, FILE \*filep)
- int ostr (flash x, char \*string)
- void premult (big x, int n, big z)
- void putdig (int n, big x, int i)
- int remain (big x, int n)
- void set\_io\_buffer\_size (int len)
- void set\_user\_function (BOOL(\*user)(void))
- int size\* (big x)
- int subdiv (big x, int n, big z)
- BOOL subdivisible (big x, int n)
- void subtract (big x, big y, big z)
- void zero\* (flash x)

### 2.1.1 Function Documentation

#### 2.1.1.1 void absol\* (flash x, flash y)

Gives absolute value of a big or flash number

Parameters:

←-|x| The number whose absolute value is to be computed  
→y = |x|

#### 2.1.1.2 void add (big x, big y, big z)

Adds two big numbers

Parameters:

←-x  
→y  
→z = x + y

Example:

```
add(x, x, x); // This doubles the value of x
```

#### 2.1.1.3 int big\_to\_bytes (int max, big x, char \* ptr, BOOL justify)

Converts a positive big number into a binary octet string. Error checking is carried out to ensure that the function does not write beyond the limits of ptr if max > 0. If max = 0, no checking is carried out. If max > 0 and justify = TRUE, the output is right-justified, otherwise leading zeros are suppressed

Parameters:

←-max Maximum number of octets to be written in ptr  
←-x The original big number  
→ptr Destination of the binary octet string  
→justify If TRUE, the output is right-justified, otherwise leading zeros are suppressed

Returns:

The number of bytes generated in ptr. If justify = TRUE then the return value is max

Precondition:

max must be greater than 0 if justify = TRUE

#### 2.1.1.4 void bigbits (int n, big x)

Generates a big random number of given length. Uses the built-in simple random number generator initialised by `irand()`

**Parameters:**

←-n The desired length of the random big number  
→x The random number

#### 2.1.1.5 `mr_small brand (void)`

Generates random integer number

**Returns:**

A random integer number

**Precondition:**

First use must be preceded by an initial call to `irand()`

**Warning:**

This generator is not cryptographically strong. For cryptographic applications, use the `strong_rng()` routine

#### 2.1.1.6 `void bytes_to_big (int len, char * ptr, big x)`

Converts a binary octet string to a big number. Binary to big conversion

**Parameters:**

←-len Length of ptr  
←-ptr Byte array of the binary octet string  
→x Big result

**Example:**

```
#include <stdio.h>
#include "miracl.h"

int main()
{
    int i, len;
    miracl *mip = mirsys(100, 0);
    big x, y;
    char b[200]; // b needs space allocated to it
    x = mirvar(0); // all big variables need to be "mirvar"ed
    y = mirvar(0);

    expb2(100, x);
    incr(x, 3, x); // x = 2^100 + 3

    len = big_to_bytes(200, x, b, FALSE);
    // Now b contains big number x in raw binary
    // It is len bytes in length

    // now print out the raw binary number b in hex
    for (i = 0; i < len; i++) printf("%02x", b[i]);
    printf("\n");

    // now convert it back to big format, and print it out again
    bytes_to_big(len, b, y);
    mip->IOBASE = 16;
    cotnum(y, stdout);

    return 0;
}
```

#### 2.1.1.7 `int cinnum (flash x, FILE * filep)`

Inputs a flash/big number from the keyboard or a file, using as number base the current value of the instance variable `miracl::IOBASE`. Flash numbers can be entered using either a slash '/' to indicate numerator and denominator, or with a radix point

**Parameters:**

→x Big/flash number  
←-filep File descriptor. For input from the keyboard specify `stdin`, otherwise as the descriptor of some other opened file

**Note:**

To force input of a fixed number of bytes, set the instance variable `miracl::INPLEN` to the required number, just before calling `cinnum()`

**Example:**

```
mip->IOBASE = 256;
mip->INPLEN = 14; // this inputs 14 bytes from fp and
cinnum(x, fp); // converts them into big number x
```

**2.1.1.8 int cinstr (flash x, char \* string)**

Inputs a flash/big number from a character string, using as number base the current value of the instance variable `miracl::IOBASE`. Flash numbers can be input using a slash '/' to indicate numerator and denominator, or with a radix point

**Parameters:**

→x  
←string

**Returns:**

The number of input characters

**Example:**

```
// input large hex number into big x
mip->IOBASE = 16;
cinstr(x, "AF12398065BFE4C96DB723A");
```

**2.1.1.9 int compare\* (big x, big y)**

Compares two big numbers

**Parameters:**

←x  
→y

**Returns:**

+1 if  $x > y$ ; 0 if  $x = y$ ; -1 if  $x < y$

**2.1.1.10 void convert (int n, big x)**

Converts an integer number to big number format

**Parameters:**

←n  
→x

**2.1.1.11 void copy\* (flash x, flash y)**

Copies a big/flash number to another

**Parameters:**

←x  
→y= x

**Note:**

If x and y are the same variable, no operation is performed

**2.1.1.12 int cotnum (flash x, FILE \* filep)**

Outputs a big/flash number to the screen or to a file, using as number base the value currently assigned to the instance variable `miracl::IOBASE`. A flash number will be converted to radix-point representation if the instance variable `miracl::RPOINT` = ON. Otherwise it will output as a fraction

**Parameters:**

←x Big/flash number to be output  
→filep File descriptor. If stdout then output will be to the screen, otherwise to the file opened with descriptor filep

**Returns:**

Number of output characters

**Example:**

```
// This outputs x in hex, to the file associated with fp
mip->IOBASE = 16;
cotnum(x, fp);
```

**2.1.1.13 int cotstr (flash x, char \* string)**

Outputs a big/flash number to the specified string, using as number base the value currently assigned to the instance variable `miracl::IOBASE`. A flash number will be converted to radix-point representation if the instance variable `miracl::RPOINT = ON`. Otherwise it will be output as a fraction

**Parameters:**

←x  
→string

**Returns:**

Number of output characters

**Warning:**

Note that there is nothing to prevent this routine from overflowing the limits of the user supplied character array string, causing obscure runtime problems. It is the programmer's responsibility to ensure that string is big enough to contain the number output to it. Alternatively use the internally declared instance string `miracl::IOBUFF`, which is of size `miracl::IOBSIZ`. If this array overflows a MIRACL error will be flagged

**2.1.1.14 void decr (big x, int n, big z)**

Decrements a big number by an integer amount

**Parameters:**

←x  
←n  
→z = x - n

**2.1.1.15 void divide (big x, big y, big z)**

Divides one big number by another:  $z = x/y$ ,  $x = x \pmod{y}$ . The quotient only is returned if x and z are the same, the remainder only if y and z are the same

**Parameters:**

←→x  
→y  
→z

**Precondition:**

Parameters x and y must be different, and y must be non-zero

**See also:**

`normalise()`

**2.1.1.16 BOOL divisible (big x, big y)**

Tests a big number for divisibility by another

**Parameters:**

←x  
→y

**Returns:**

TRUE if y divides x exactly, otherwise FALSE

**Precondition:**

The parameter y must be non-zero

**2.1.1.17 void\* ecp\_memalloc (int num)**

Reserves space for a number elliptic curve points in one heap access. Individual points can subsequently be initialised from this memory by calling epoint\_init\_mem()

**Parameters:**

←num The number of elliptic curve points to reserve space for

**Returns:**

A pointer to the allocated memory

**2.1.1.18 void ecp\_memkill (char \* mem, int num)**

Deletes and sets to zero the memory previously allocated by ecp\_memalloc()

**Parameters:**

→mem Pointer to the memory to be erased and deleted

←num The size of the memory in elliptic curve points

**Precondition:**

Must be preceded by a call to ecp\_memalloc()

**2.1.1.19 int exsign\* (flash x)**

Extracts the sign of a big/flash number

**Parameters:**

←x A big/flash number

**Returns:**

The sign of x, i.e. -1 if x is negative, +1 if x is zero or positive

**2.1.1.20 miracl\* get\_mip ()**

Gets the current Miracl Instance Pointer

**Returns:**

The mip (Miracl Instance Pointer) for the current thread

**Precondition:**

This function does not exist if MR\_GENERIC\_MT is defined

**2.1.1.21 int getdig (big x, int i)**

Extracts a digit from a big number

**Parameters:**

←x A big number

→i The position of the digit to be extracted from x

**Returns:**

The value of the requested digit

**Warning:**

Returns rubbish if required digit does not exist

**2.1.1.22 unsigned int igcd\* (unsigned int x, unsigned int y)**

Calculates the Greatest Common Divisor of two integers using Euclid's Method

**Parameters:**

←x

←y

**Returns:**

The GCD of x and y

**2.1.1.23 void incr (big x, int n, big z)**

Increments a big number

**Parameters:**

←x  
 ←n  
 →z = x + n

**Example:**

```
incr(x, 2, x); // This increments x by 2
```

**2.1.1.24 BOOL init\_big\_from\_rom (big x, int len, const mr\_small \* rom, int romsize, int \* romptr)**

Initialises a big variable from ROM memory

**Parameters:**

→x A big number  
 ←len Length of the big number in computer words  
 ←rom Address of ROM memory which stores up to romsize computer words  
 ←romsize  
 ←→romptr A pointer into ROM. This pointer is incremented internally as ROM memory is accessed to fill x

**Returns:**

TRUE if successful, or FALSE if an attempt is made to read beyond the end of the ROM

**2.1.1.25 BOOL init\_point\_from\_rom (epoint \* P, int len, const mr\_small \* rom, int romsize, int \* romptr)**

Initialises an elliptic curve point from ROM memory

**Parameters:**

→P An elliptic curve point  
 ←len Length of the two big coordinates of P, in computer words  
 ←rom Address of ROM memory which stores up to romsize computer words  
 ←romsize  
 ←→romptr A pointer into ROM. This pointer is incremented internally as ROM memory is accessed to fill P

**Returns:**

TRUE if successful, or FALSE if an attempt is made to read beyond the end of the ROM

**2.1.1.26 int innum (flash x, FILE \* filep)**

Inputs a big/flash number from a file or the keyboard, using as number base the value specified in the initial call to mirsys(). Flash numbers can be entered using either a slash '/' to indicate numerator and denominator, or with a radix point

**Parameters:**

→x A big/flash number  
 ←filep A file descriptor. For input from the keyboard specify stdin, otherwise the descriptor of some other opened file

**Returns:**

The number of characters input

**Precondition:**

The number base specified in mirsys() must be less than or equal to 256. If not use cinnum() instead

**Note:**

For fastest inputting of ASCII text to a big number, and if a full-width base is possible, use mirsys(...,256) initially. This has the same effect as specifying mirsys(...,0), except that now ASCII bytes may be input directly via innum(x, fp) without the time-consuming change of base implicit in the use of cinnum()

**2.1.1.27 void insign\* (int s, flash x)**

Forces a big/flash number to a particular sign

Parameters:

←s The sign the big/flash is to take  
→x = s|x|

Example:

```
insign(PLUS, x); // force x to be positive
```

#### 2.1.1.28 int instr (flash x, char \* string)

Inputs a big or flash number from a character string, using as number base the value specified in the initial call to mirsys(). Flash numbers can be entered using either a slash '/' to indicate numerator and denominator, or with a radix point

Parameters:

→x  
←string

Returns:

The number of characters input

Precondition:

The number base specified in mirsys() must be less than or equal to 256. If not use cinstr() instead

#### 2.1.1.29 void irand (mr\_unsign32 seed)

Initialises internal random number system. Long integer types are used internally to yield a generator with maximum period

Parameters:

←seed A seed used to start off the random number generator

#### 2.1.1.30 void lgconv (long n, big x)

Converts a long integer to big number format

Parameters:

←n  
→x

#### 2.1.1.31 void mad (big x, big y, big z, big w, big q, big r)

Multiplies, adds and divides big numbers. The initial product is stored in a double-length internal variable to avoid the possibility of overflow at this stage

Parameters:

←x  
←y  
←z  
←w  
→q = (xy + z)/w  
→r The remainder

Note:

If w and q are not distinct variables then only the remainder is returned; if q and r are not distinct then only the quotient is returned. The addition of z is not done if x and z (or y and z) are the same

Precondition:

Parameters w and r must be distinct. The value of w must not be zero

Example:

```
mad(x, x, x, w, x, x,); // x = x^2 / w
```

#### 2.1.1.32 void\* memalloc (int num)

Reserves space for big/flash variables in one heap access. Individual big/flash variables can subsequently be initialised from this memory by calling mirvar\_mem()



**Parameters:**

←num The number of big/flash variables to reserve space for

**Returns:**

A pointer to the allocated memory

**2.1.1.33 void memkill (char \* mem, int len)**

Deletes and sets to zero the memory previously allocated by memalloc()

**Parameters:**

→mem A pointer to the memory to be erased and deleted

←len The size of that memory in bigs

**Precondition:**

Must be preceded by a call to memalloc()

**2.1.1.34 void mirexit (void)**

Cleans up after the current instance of MIRACL, and frees all internal variables. A subsequent call to mirsys() will re-initialise the MIRACL system

**Precondition:**

Must be called after mirsys()

**2.1.1.35 void mirkill\* (big x)**

Securely kills off a big/flash number by zeroising it, and freeing its memory

**Parameters:**

←x

**2.1.1.36 miracl\* mirsys (int nd, mr\_small nb)**

Initialises the MIRACL system for the current program thread, as described below. Must be called before attempting to use any other MIRACL routines

1. The error tracing mechanism is initialised
2. The number of computer words to use for each big/flash number is calculated from nd and nb
3. Sixteen big work variables (four of them double length) are initialised
4. Certain instance variables are given default initial values
5. The random number generator is started by calling irand(0L)

**Parameters:**

←nd The number of digits to use for each big/flash variable. If negative, it is taken as indicating the size of big/flash numbers in 8-bit bytes

→nb The number base

**Returns:**

The Miracl Instance Pointer, via which all instance variables can be accessed, or NULL if there was not enough memory to create an instance

**Precondition:**

The number base nb should normally be greater than 1 and less than or equal to MAXBASE. A base of 0 implies that the 'full-width' number base should be used. The number of digits nd must be less than a certain maximum, depending on the underlying type mr\_utype and on whether or not MR\_FLASH is defined

**Example:**

```
// This initialises the MIRACL system to use 500 decimal digits for each
// big or flash number
miracl *mip = mirsys(500, 10);
```

**2.1.1.37 flash mirvar (int iv)**

Initialises a big/flash variable by reserving a suitable number of memory locations for it. This memory may be released by a subsequent call to the function mirkill()

**Parameters:**

←iv An integer initial value for the big/flash number

**Returns:**

A pointer to the reserved memory

**Example:**

```
flash x;
x = mirvar(8); // Creates a flash variable x = 8
```

**2.1.1.38 flash mirvar\_mem (char \* mem, int index)**

Initialises memory for a big/flash variable from a pre-allocated byte array mem. This array may be created from the heap by a call to memalloc(), or in some other way. This is quicker than multiple calls to mirvar()

**Parameters:**

←mem A pointer to the pre-allocated array  
 ←index An index into that array. Each index should be unique

**Returns:**

An initialised big/flash variable

**Precondition:**

Sufficient memory must have been allocated and pointed to by mem

**2.1.1.39 void multiply (big x, big y, big z)**

Multiplies two big numbers

**Parameters:**

←x  
 ←y  
 →z = xy

**2.1.1.40 void negify\* (flash x, flash y)**

Negates a big/flash number

**Parameters:**

←x  
 →y = - x

**Note:**

negify(x,x) is valid and sets x = - x

**2.1.1.41 mr\_small normalise (big x, big y)**

Multiplies a big number such that its most significant word is greater than half the number base. If such a number is used as a divisor by divide(), the division will be carried out faster. If many divisions by the same divisor are required, it makes sense to normalise the divisor just once beforehand

**Parameters:**

←x  
 →y = nx

**Returns:**

n, the normalising multiplier

**Warning:**

Use with care. Used internally

**2.1.1.42 int numdig (big x)**

Determines the number of digits in a big number

**Parameters:**

←x

**Returns:**

The number of digits in x

**2.1.1.43 int otnum (flash x, FILE \* filep)**

Outputs a big/flash number to the screen or to a file, using as number base the value specified in the initial call to mirsys(). A flash number will be converted to radix-point representation if the instance variable miracl::RPOINT = ON. Otherwise it will be output as a fraction

**Parameters:**

←x A big/flash number  
 ←filep A file descriptor. If stdout then output will be to the screen, otherwise to the file opened with descriptor filep

**Returns:**

Number of output characters

**Precondition:**

The number base specified in mirsys() must be less than or equal to 256. If not, use cotnum() instead

**2.1.1.44 int ostr (flash x, char \* string)**

Outputs a big or flash number to the specified string, using as number base the value specified in the initial call to mirsys(). A flash number will be converted to radix-point representation if the instance variable miracl::RPOINT = ON. Otherwise it will be output as a fraction

**Parameters:**

←x A big/flash number  
 →string A representation of x

**Returns:**

Number of output characters

**Precondition:**

The number base specified in mirsys() must be less than or equal to 256. If not, use cotstr() instead

**Warning:**

Note that there is nothing to prevent this routine from overflowing the limits of the user supplied character array string, causing obscure runtime problems. It is the programmer's responsibility to ensure that string is big enough to contain the number output to it. Alternatively use the internally declared instance string miracl::IOBUFF, which is of size miracl::IOBSIZ. If this array overflows a MIRACL error will be flagged

**2.1.1.45 void premult (big x, int n, big z)**

Multiplies a big number by an integer

**Parameters:**

←x  
 ←n  
 →z = nx

**2.1.1.46 void putdig (int n, big x, int i)**

Sets a digit of a big number to a given value

**Parameters:**

←n The new value for the digit  
 →x A big number  
 ←i A digit position

**Precondition:**

The digit indicated must exist

**2.1.1.47 int remain (big x, int n)**

Finds the integer remainder, when a big number is divided by an integer

**Parameters:**

←x  
←n

**Returns:**

The integer remainder

#### 2.1.1.48 void set\_io\_buffer\_size (int len)

Sets the size of the input/output buffer. By default this is set to 1024, but programs that need to handle very large numbers may require a larger I/O buffer

**Parameters:**

←len The size of I/O buffer required

**Warning:**

Destroys the current contents of the I/O buffer

#### 2.1.1.49 void set\_user\_function (BOOL\*)(void) user)

Supplies a user-specified function, which is periodically called during some of the more time-consuming MIRACL functions, particularly those involved in modular exponentiation and in finding large prime numbers. The supplied function must take no parameters and return a BOOL value. Normally this should be TRUE. If FALSE then MIRACL will attempt to abort its current operation. In this case the function should continue to return FALSE until control is returned to the calling program. The user-supplied function should normally include only a few instructions, and no loops, otherwise it may adversely impact the speed of MIRACL functions

Once MIRACL is initialised, this function may be called multiple times with a new supplied function. If no longer required, call with a NULL parameter

**Parameters:**

←user A pointer to a user-supplied function, or NULL if not required

**Example:**

```
// Windows Message Pump
static BOOL idle()
{
    MSG msg;
    if (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
    {
        if (msg.message != WM_QUIT)
        {
            if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                // do a Message Pump
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
        else
            return FALSE;
    }
    return TRUE;
}
...
set_user_function(idle);
```

#### 2.1.1.50 int size\* (big x)

Tries to convert big number to a simple integer. Also useful for testing the sign of big/flash variable as in:  
if (size(x) < 0) ...

**Parameters:**

←x A big number

**Returns:**

The value of  $x$  as an integer. If this is not possible (because  $x$  is too big) it returns the value plus or minus MR\_TOOBIG

#### 2.1.1.51 int subdiv (big x, int n, big z)

Divides a big number by an integer

Parameters:

$\leftarrow x$   
 $\leftarrow n$   
 $\rightarrow z = x/n$

Returns:

The integer remainder

Precondition:

The value of  $n$  must not be zero

#### 2.1.1.52 BOOL subdivisible (big x, int n)

Tests a big number for divisibility by an integer

Parameters:

$\leftarrow x$   
 $\leftarrow n$

Returns:

TRUE if  $n$  divides  $x$  exactly, otherwise FALSE

Precondition:

The value of  $n$  must not be zero

#### 2.1.1.53 void subtract (big x, big y, big z)

Subtracts two big numbers

Parameters:

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow z = x - y$

#### 2.1.1.54 void zero\* (flash x)

Sets a big/flash number to zero

Parameters:

$\rightarrow x$

## 2.2 Advanced arithmetic routines

Functions

- void bigdig (int n, int b, big x)
- void bigrand (big w, big x)
- void brick\_end\* (brick \*b)
- BOOL brick\_init (brick \*b, big g, big n, int window, int nb)
- void crt (big\_chinese \*c, big \*u, big x)
- void crt\_end\* (big\_chinese \*c)
- BOOL crt\_init (big\_chinese \*c, int r, big \*moduli)
- int egcd (big x, big y, big z)
- void expb2 (int n, big x)
- void expint (int b, int n, big x)
- void fft\_mult (big x, big y, big z)
- void gprime (int maxp)
- int hamming (big x)
- mr\_small invers\* (mr\_small x, mr\_small y)
- BOOL isprime (big x)
- int jac (mr\_small x, mr\_small n)

- int jack (big U, big V)
- int logb2 (big x)
- void lucas (big p, big r, big n, big vp, big v)
- BOOL multi\_inverse (int m, big \*x, big n, big \*w)
- BOOL nroot (big x, int n, big w)
- BOOL nxprime (big w, big x)
- BOOL nxsafeprime (int type, int subset, big w, big p)
- void pow\_brick (brick \*b, big e, big w)
- void power (big x, long n, big z, big w)
- int powltr (int x, big y, big n, big w)
- void powmod (big x, big y, big n, big w)
- void powmod2 (big x, big y, big a, big b, big n, big w)
- void powmodn (int n, big \*x, big \*y, big p, big w)
- void sqrt (small\_chinese \*c, mr\_utype \*u, big x)
- void sqrt\_end\* (small\_chinese \*c)
- BOOL sqrt\_init (small\_chinese \*c, int r, mr\_utype \*moduli)
- void sftbit (big x, int n, big z)
- mr\_small smul\* (mr\_small x, mr\_small y, mr\_small n)
- mr\_small spmd\* (mr\_small x, mr\_small n, mr\_small m)
- mr\_small sqrpm\* (mr\_small x, mr\_small m)
- BOOL sqroot (big x, big p, big w)
- int trial\_division (big x, big y)
- int xgcd (big x, big y, big xd, big yd, big z)

## 2.2.1 Function Documentation

### 2.2.1.1 void bigdig (int n, int b, big x)

Generates a big random number of given length. Uses the built-in simple random number generator initialised by irand()

Parameters:

- ←n
- ←b
- x A big random number n digits long to base b

Precondition:

The base b must be printable, that is  $2 \leq b \leq 256$

Example:

```
// This generates a 100 decimal digit random number
bigdig(100, 10, x);
```

### 2.2.1.2 void bigrand (big w, big x)

Generates a big random number. Uses the built-in simple random number generator initialised by irand()

Parameters:

- ←w
- x A big random number in the range  $0 \leq x < w$

### 2.2.1.3 void brick\_end\* (brick \* b)

Cleans up after an application of the Comb method

Parameters:

- ←b A pointer to the current instance

### 2.2.1.4 BOOL brick\_init (brick \* b, big g, big n, int window, int nb)

Initialises an instance of the Comb method for modular exponentiation with precomputation. Internally memory is allocated for  $2w$  big numbers which will be precomputed and stored. For bigger  $w$  more space is required, but the exponentiation is quicker. Try  $w = 8$

Parameters:

- ←→b A pointer to the current instance
- ←g The fixed generator
- ←n The modulus
- ←window The window size  $w$

←nb The maximum number of bits to be used in the exponent

**Returns:**

TRUE if successful, otherwise FALSE

**Note:**

If MR\_STATIC is defined in mirdef.h, then the g parameter in this function is replaced by an mr\_small pointer to a precomputed table. In this case the function returns a void

### 2.2.1.5 void crt (big\_chinese \* c, big \* u, big x)

Applies the Chinese Remainder Theorem

**Parameters:**

←c A pointer to the current instance  
 ←u An array of big remainders  
 →x The big number which yields the given remainders u when it is divided by the big moduli specified in a prior call to crt\_init()

**Precondition:**

The routine crt\_init() must be called first

### 2.2.1.6 void crt\_end\* (big\_chinese \* c)

Cleans up after an application of the Chinese Remainder Theorem

**Parameters:**

←c A pointer to the current instance

### 2.2.1.7 BOOL crt\_init (big\_chinese \* c, int r, big \* moduli)

Initialises an instance of the Chinese Remainder Theorem. Some internal workspace is allocated

**Parameters:**

→c A pointer to the current instance  
 ←r The number of co-prime moduli  
 ←moduli An array of at least two big moduli

**Returns:**

TRUE if successful, otherwise FALSE

### 2.2.1.8 int egcd (big x, big y, big z)

Calculates the Greatest Common Divisor of two big numbers

**Parameters:**

←x  
 ←y  
 ←z = gcd(x,y)

**Returns:**

GCD as integer, if possible, otherwise MR\_TOOBIG

### 2.2.1.9 void expb2 (int n, big x)

Calculates 2 to the power of an integer as a big

**Parameters:**

←n  
 →x = 2<sup>n</sup>

**Example:**

```
// This calculates and prints out the largest known prime number
// (on a true 32-bit computer with lots of memory!)
expb2(1398269, x);
decr(x, 1, x);
mip->IOBASE = 10;
cotnum(x, stdout);
```

**2.2.1.10 void expint (int b, int n, big x)**

Calculates an integer to the power of an integer as a big

**Parameters:**

$\leftarrow b$   
 $\leftarrow n$   
 $\rightarrow x = b^n$

**2.2.1.11 void fft\_mult (big x, big y, big z)**

Multiplies two big numbers, using the Fast Fourier Method. See [Pollard71]

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow z = xy$

**Note:**

Should only be used on a 32-bit computer when x and y are ver large, at least 1000 decimal digits.

**2.2.1.12 void gprime (int maxp)**

Generates all prime numbers up to a certain limit into the instance array miracl::PRIMES, terminated by zero. This array is used internally by the routines isprime() and nxprime()

**Parameters:**

$\leftarrow \text{maxp}$  A positive integer indicating the maximum prime number to be generated. If maxp = 0 the miracl::PRIMES array is deleted

**2.2.1.13 int hamming (big x)**

Calculates the hamming weight of a big number (in fact the number of 1's in its binary representation)

**Parameters:**

$\leftarrow x$

**Returns:**

Hamming weight of x

**2.2.1.14 mr\_small invers\* (mr\_small x, mr\_small y)**

Calculates the inverse of an integer modulus a co-prime integer.

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$

**Returns:**

$x^{-1} \pmod{y}$

**Warning:**

Result unpredictable if x and y not co-prime

**2.2.1.15 BOOL isprime (big x)**

Tests whether or not a big number is prime using a probabilistic primality test. The number is assumed to be prime if it passes this test miracl::NTRY times, where miracl::NTRY is an instance variable with a default initialisation in routine mirsys()

**Parameters:**

$\leftarrow x$

**Returns:**

TRUE if x is (almost certainly) prime, otherwise FALSE

**Note:**



This routine first test divides x by the list of small primes stored in the instance array `miracl::PRIMES`. The testing of larger primes will be significantly faster in many cases if this list is increased. See `gprime()`. By default only the small primes less than 1000 are used

**See also:**

`gprime()`

**2.2.1.16 int jac (mr\_small x, mr\_small n)**

Calculates the value of the Jacobi symbol. See [Reisel]

**Parameters:**

$\leftarrow x$   
 $\leftarrow n$

**Returns:**

The value of  $(x | n)$  as +1 or -1, or 0 if symbol undefined

**See also:**

`jack`

**2.2.1.17 int jack (big U, big V)**

Calculates the value of the Jacobi symbol. See [Reisel]

**Parameters:**

$\leftarrow U$   
 $\leftarrow V$

**Returns:**

The value of  $(U | V)$  as +1 or -1, or 0 if symbol undefined

**See also:**

`jac`

**2.2.1.18 int logb2 (big x)**

Calculates the approximate integer log to the base 2 of a big number (in fact the number of bits in it)

**Parameters:**

$\leftarrow x$

**Returns:**

Number of bits in x

**2.2.1.19 void lucas (big p, big r, big n, big vp, big v)**

Performs Lucas modular exponentiation. Uses Montgomery arithmetic internally. This function can be speeded up further for particular moduli, by invoking special assembly language routines to implement Montgomery arithmetic. See `powmod()`

**Parameters:**

$\leftarrow p$  The base  
 $\leftarrow r$  The exponent  
 $\leftarrow n$  The modulus  
 $\rightarrow vp = V_{r-1}(p) \pmod n$   
 $\rightarrow v = V_r(p) \pmod n$

**Note:**

Only v is returned if v and vp are not distinct  
 The “sister” Lucas function  $U_r(p)$  can, if required, be calculated as  $U_r(p) * [pV_r(p) - 2V_{r-1}(p)] / (p^2 - 4) \pmod n$

**Precondition:**

The value of n must be odd

**2.2.1.20 BOOL multi\_inverse (int m, big \* x, big n, big \* w)**

Finds the modular inverses of many numbers simultaneously, exploiting Montgomery's observation that  $x^{-1} = y(xy)^{-1}$ ,  $y^{-1} = x(xy)^{-1}$ . This will be quicker, as modular inverses are slow to calculate, and this way only one is required

**Parameters:**

←m The number of inverses required  
 ←x An array of m numbers whose inverses are required  
 ←n The modulus  
 →w The resulting array of inverses

**Returns:**

TRUE if successful, otherwise FALSE

**Precondition:**

The parameters x and w must be distinct

**2.2.1.21 BOOL nroot (big x, int n, big w)**

Extracts lower approximation to a root of a big number

**Parameters:**

←x A big number  
 ←n A positive integer  
 →w = [nx]

**Returns:**

TRUE if the root is exact, otherwise FALSE

**Precondition:**

The value of n must be positive. If x is negative, then n must be odd

**See also:**

sqroot, nres\_sqroot

**2.2.1.22 BOOL nxprime (big w, big x)**

Finds next prime number

**Parameters:**

←w  
 ←x The next prime number greater than w

**Returns:**

TRUE if successful, otherwise FALSE

**See also:**

nxsafeprime

**2.2.1.23 BOOL nxsafeprime (int type, int subset, big w, big p)**

Finds next safe prime number greater than w. A safe prime number p is defined here to be one for which  $q = (p - 1)/2$  (type=0) or  $q = (p + 1)/2$  (type=1) is also prime

**Parameters:**

←type The type of safe prime as above  
 ←subset If subset = 1, then the search is restricted so that the value of the prime q is congruent to 1 mod 4. If subset = 3, then the search is restricted so that the value of q is congruent to 3 mod 4. If subset = 0 then there is no condition on q: it can be either 1 or 3 mod 4  
 ←w  
 →p

**Returns:**

TRUE if successful, otherwise FALSE

**See also:**

nxprime

#### 2.2.1.24 void pow\_brick (brick \* b, big e, big w)

Carries out a modular exponentiation, using the precomputed values stored in the brick structure

##### Parameters:

←b A pointer to the current instance  
 ←e A big exponent  
 →w =  $g^e \pmod{n}$ , where g and n are specified in the initial call to brick\_init()

##### Precondition:

Must be preceded by a call to brick\_init()

#### 2.2.1.25 void power (big x, long n, big z, big w)

Raises a big number to an integer power

##### Parameters:

←x A big number  
 ←n A positive integer  
 ←z A big number  
 →w =  $x^n \pmod{z}$

##### Precondition:

The value of n must be positive

#### 2.2.1.26 int powltr (int x, big y, big n, big w)

Raises an int to the power of a big number modulus another big number. Uses Left-to-Right binary method, and will be somewhat faster than powmod() for small x. Uses Montgomery arithmetic internally if the modulus n is odd

##### Parameters:

←x  
 ←y  
 ←n  
 →w =  $xy \pmod{n}$

##### Returns:

The result expressed as an integer, if possible. Otherwise the value MR\_TOOBIG

##### Precondition:

The value of y must be positive. The parameters x and n must be distinct

#### 2.2.1.27 void powmod (big x, big y, big n, big w)

Raises a big number to a big power modulus another big. Uses a sophisticated 5-bit sliding window technique, which is close to optimal for popular modulus sizes (such as 512 or 1024 bits). Uses Montgomery arithmetic internally if the modulus n is odd.

This function can be speeded up further for particular moduli, by invoking special assembly language routines (if your compiler allows it). A KCM Modular Multiplier will be automatically invoked if MR\_KCM has been defined in mirdef.h and has been set to an appropriate size. Alternatively a Comba modular multiplier will be used if MR\_COMBA is so defined, and the modulus is of the specified size. Experimental coprocessor code will be called if MR\_PENTIUM is defined. Only one of these conditionals should be defined

##### Parameters:

←x  
 ←y  
 ←n  
 →w =  $xy \pmod{n}$

##### Precondition:

The value of y must be positive. The parameters x and n must be distinct

#### 2.2.1.28 void powmod2 (big x, big y, big a, big b, big n, big w)

Calculates the product of two modular exponentiations. This is quicker than doing two separate exponentiations, and is useful for certain cryptographic protocols. Uses 2-bit sliding window

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\leftarrow a$   
 $\leftarrow b$   
 $\leftarrow n$   
 $\rightarrow w = xy \text{ ab (mod } n)$

**Precondition:**

The values of  $y$  and  $b$  must be positive. The parameters  $n$  and  $w$  must be distinct. The modulus  $n$  must be odd

**2.2.1.29 void powmodn (int n, big \* x, big \* y, big p, big w)**

Calculates the product of  $n$  modular exponentiations. This is quicker than doing  $n$  separate exponentiations, and is useful for certain cryptographic protocols. Extra memory is allocated internally for this function

**Parameters:**

$\leftarrow n$   
 $\leftarrow x$   
 $\leftarrow y$   
 $\leftarrow p$   
 $\rightarrow w = x[0]y[0]x[1]y[1] \cdots x[n-1]y[n-1] \text{ (mod } p)$

**Precondition:**

The values of  $y[i]$  must be positive. The parameters  $p$  and  $w$  must be distinct. The modulus  $p$  must be odd. The underlying number base must be a power of 2

**2.2.1.30 void scrct (small\_chinese \* c, mr\_utype \* u, big x)**

Applies Chinese Remainder Theorem (for small prime moduli)

**Parameters:**

$\leftarrow c$  A pointer to the current instance of the Chinese Remainder Theorem  
 $\leftarrow u$  An array of remainders  
 $\rightarrow x$  The big number which yields the given integer remainders  $u[i]$  when it is divided by the integer moduli specified in a prior call to `scrct_init()`

**Precondition:**

The routine `scrct_init()` must be called first

**2.2.1.31 void scrct\_end\* (small\_chinese \* c)**

Cleans up after an application of the Chinese Remainder Theorem

**Parameters:**

$\leftarrow c$  A pointer to the current instance of the Chinese Remainder Theorem

**2.2.1.32 BOOL scrct\_init (small\_chinese \* c, int r, mr\_utype \* moduli)**

Initialises an instance of the Chinese Remainder Theorem. Some internal workspace is allocated

**Parameters:**

$\rightarrow c$  A pointer to the current instance  
 $\leftarrow r$  The number of co-prime moduli  
 $\leftarrow \text{moduli}$  An array of at least two integer moduli

**Returns:**

TRUE if successful, otherwise FALSE

**2.2.1.33 void sftbit (big x, int n, big z)**

Shifts a big integer left or right by a number of bits

**Parameters:**

$\leftarrow x$   
 $\leftarrow n$  If positive shifts to the left, if negative shifts to the right  
 $\rightarrow z = x$  shifted by  $n$  bits

#### 2.2.1.34 `mr_small smul*` (`mr_small x`, `mr_small y`, `mr_small n`)

Multiplies two integers mod a third

##### Parameters:

$\leftarrow x$   
 $\leftarrow y$   
 $\leftarrow n$

##### Returns:

$xy \pmod n$

#### 2.2.1.35 `mr_small spmd*` (`mr_small x`, `mr_small n`, `mr_small m`)

Raises an integer to an integer power modulo a third

##### Parameters:

$\leftarrow x$   
 $\leftarrow n$   
 $\leftarrow m$

##### Returns:

$xn \pmod m$

#### 2.2.1.36 `mr_small sqrpm*` (`mr_small x`, `mr_small m`)

Calculates the square root of an integer modulo an integer prime number

##### Parameters:

$\leftarrow x$   
 $\leftarrow m$  A prime number

##### Returns:

$x \pmod m$ , or 0 if root does not exist

##### Precondition:

$p$  must be prime, otherwise the result is unpredictable

##### See also:

`sqroot`

#### 2.2.1.37 `BOOL sqroot` (`big x`, `big p`, `big w`)

Calculates the square root of a big integer mod a big integer prime

##### Parameters:

$\leftarrow x$   
 $\leftarrow p$   
 $\rightarrow w = x \pmod p$  if the square root exists, otherwise  $w = 0$ . Note that the "other" square root may be found by subtracting  $w$  from  $p$

##### Returns:

TRUE if the square root exists, FALSE otherwise

##### Precondition:

The number  $p$  must be prime

##### Note:

This routine is particularly efficient if  $p \equiv 3 \pmod 4$

#### 2.2.1.38 `int trial_division` (`big x`, `big y`)

Dual purpose trial division routine. If  $x$  and  $y$  are the same big variable then trial division by the small

prime numbers in the instance array `miracl::PRIMES` is attempted to determine the primality status of the big number. If `x` and `y` are distinct then, after trial division, the unfactored part of `x` is returned in `y`

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$

**Returns:**

If `x` and `y` are the same, then a return value of 0 means that the big number is definitely not prime, a return value of 1 means that it definitely is prime, while a return value of 2 means that it is possibly prime (and that perhaps further testing should be carried out). If `x` and `y` are distinct, then a return value of 1 means that `x` is smooth, that it is completely factored by trial division (and `y` is the largest prime factor). A return value of 2 means that the unfactored part `y` is possibly prime

### 2.2.1.39 int xgcd (big x, big y, big xd, big yd, big z)

Calculates extended Greatest Common Divisor of two big numbers. Can be used to calculate modular inverses. Note that this routine is much slower than a `mad()` operation on numbers of similar size

**Parameters:**

$\leftarrow x$   
 $\rightarrow y$   
 $\rightarrow xd$   
 $\rightarrow yd$   
 $\rightarrow z = \text{gcd}(x, y) = (x * xd) + (y * yd)$

**Returns:**

GCD as integer, if possible, otherwise `MR_TOOBIG`

**Precondition:**

If `xd` and `yd` are not distinct, only `xd` is returned. The GCD is only returned if `z` distinct from both `xd` and `yd`

**Example:**

```
xgcd(x, p, x, x, x,); // x = 1/x mod p (p is prime)
```

## 2.3 Montgomery arithmetic routines

**Functions**

- void nres (big x, big y)
- void nres\_dotprod (int m, big \*x, big \*y, big w)
- void nres\_double\_modadd (big x, big y, big w)
- void nres\_double\_modsub (big x, big y, big w)
- void nres\_lazy (big a0, big a1, big b0, big b1, big r, big i)
- void nres\_lucas (big p, big r, big vp, big v)
- void nres\_modadd (big x, big y, big w)
- int nres\_moddiv (big x, big y, big w)
- void nres\_modmult (big x, big y, big w)
- void nres\_modsub (big x, big y, big w)
- BOOL nres\_multi\_inverse (int m, big \*x, big \*w)
- void nres\_negate (big x, big w)
- void nres\_powltr (int x, big y, big w)
- void nres\_powmod (big x, big y, big w)
- void nres\_powmod2 (big x, big y, big a, big b, big w)
- void nres\_powmodn (int n, big \*x, big \*y, big w)
- void nres\_premult (big x, int k, big w)
- BOOL nres\_sqroot (big x, big w)
- mr\_small prepare\_monty (big n)
- void redc (big x, big y)

### 2.3.1 Function Documentation

#### 2.3.1.1 void nres (big x, big y)

Converts a big number to n-residue form

**Parameters:**

$\leftarrow x$   
 $\rightarrow y$  the  $n$ -residue form of  $x$

**Precondition:**

Must be preceded by call to `prepare_monty()`

**See also:**

`redc`

**2.3.1.2 void nres\_dotprod (int m, big \* x, big \* y, big w)**

Finds the dot product of two arrays of  $n$ -residues. So-called “lazy” reduction is used, in that the sum of products is only reduced once with respect to the Montgomery modulus. This is quicker---nearly twice as fast

**Parameters:**

$\leftarrow m$   
 $\leftarrow x$  An array of  $m$   $n$ -residues  
 $\leftarrow y$  An array of  $m$   $n$ -residues  
 $\rightarrow w = \sum x_i y_i \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by call to `prepare_monty()`

**2.3.1.3 void nres\_double\_modadd (big x, big y, big w)**

Adds two double length bigs modulo  $pR$ , where  $R = Zn$  and  $n$  is the smallest multiple of the word-length of the underlying MIRACL type, such that  $R > p$ . This is required for lazy reduction.

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = a + b \pmod{pR}$

**2.3.1.4 void nres\_double\_modsub (big x, big y, big w)**

Subtracts two double length bigs modulo  $pR$ , where  $R = Zn$  and  $n$  is the smallest multiple of the wordlength of the underlying MIRACL type, such that  $R > p$ . This is required for lazy reduction

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = a - b \pmod{pR}$

**2.3.1.5 void nres\_lazy (big a0, big a1, big b0, big b1, big r, big i)**

Uses the method of lazy reduction combined with Karatsuba’s method to multiply two `zzn2` variables. Requires just 3 multiplications and two modular reductions.

**Parameters:**

$\leftarrow a0$   
 $\leftarrow a1$   
 $\leftarrow b0$   
 $\leftarrow b1$   
 $\rightarrow r$  = the “real part” of  $(a0 + a1i)(b0 + b1i)$   
 $\rightarrow i$  = the “imaginary part” of  $(a0 + a1i)(b0 + b1i)$

**2.3.1.6 void nres\_lucas (big p, big r, big vp, big v)**

Modular Lucas exponentiation of an  $n$ -residue

**Parameters:**

$\leftarrow p$  An  $n$ -residue  
 $\leftarrow r$  A big exponent  
 $\rightarrow vp = V_{r-1}(p) \pmod{n}$  where  $n$  is the current Montgomery modulus  
 $\rightarrow v = V_r(p) \pmod{n}$  where  $n$  is the current Montgomery modulus

**Note:**

Only  $v$  is returned if  $v$  and  $vp$  are the same big variable

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of the first parameter to n-residue form.  
Note that the exponent is not converted to n-residue form

**See also:**

lucas

**2.3.1.7 void nres\_modadd (big x, big y, big w)**

Modular addition of two n-residues

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = x + y \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by a call to `prepare_monty()`

**2.3.1.8 int nres\_moddiv (big x, big y, big w)**

Modular division of two n-residues

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = x/y \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Returns:**

GCD of  $y$  and  $n$  as an integer, if possible, or `MR_TOOBIG`. Should be 1 for a valid result

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of parameters to n-residue form. Parameters  $x$  and  $y$  must be distinct

**2.3.1.9 void nres\_modmult (big x, big y, big w)**

Modular multiplication of two n-residues. Note that this routine will invoke a KCM Modular Multiplier if `MR_KCM` has been defined in `mirdef.h` and set to an appropriate size for the current modulus, or a Comba fixed size modular multiplier if `MR_COMBA` is defined as exactly the size of the modulus

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = xy \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of parameters to n-residue form

**2.3.1.10 void nres\_modsub (big x, big y, big w)**

Modular subtraction of two n-residues

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = x - y \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by a call to `prepare_monty()`

**2.3.1.11 BOOL nres\_multi\_inverse (int m, big \* x, big \* w)**

Finds the modular inverses of many numbers simultaneously, exploiting Montgomery's observation that  $x^{-1} = y(xy)^{-1}$ ,  $y^{-1} = x(xy)^{-1}$ . This will be quicker, as modular inverses are slow to calculate, and this way only one is required



**Parameters:**

$\leftarrow m$  The number of inverses required  
 $\leftarrow x$  An array of  $m$   $n$ -residues whose inverses are wanted  
 $\rightarrow w$  An array with the inverses of  $z$   $x$

**Returns:**

TRUE if successful, otherwise FALSE

**Precondition:**

The parameters  $x$  and  $w$  must be distinct

**2.3.1.12 void nres\_negate (big x, big w)**

Modular negation

**Parameters:**

$\leftarrow x$  An  $n$ -residue number  
 $\rightarrow w = -x \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by a call to `prepare_monty()`

**2.3.1.13 void nres\_powltr (int x, big y, big w)**

Modular exponentiation of an  $n$ -residue

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = x^y \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by call to `prepare_monty()`. Note that the small integer  $x$  and the exponent are not converted to  $n$ -residue form

**2.3.1.14 void nres\_powmod (big x, big y, big w)**

Modular exponentiation of an  $n$ -residue

**Parameters:**

$\leftarrow x$  An  $n$ -residue number, the base  
 $\leftarrow y$  A big number, the exponent  
 $\rightarrow w = x^y \pmod{n}$ , where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of the first parameter to  $n$ -residue form. Note that the exponent is not converted to  $n$ -residue form

**See also:**

`nres_powltr`, `nres_powmod2`

**Example:**

```

prepare_monty(n);
...
nres(x, y); // convert to n-residue form
nres_powmod(y, e, z);
redc(z, w); // convert back to normal form

```

**2.3.1.15 void nres\_powmod2 (big x, big y, big a, big b, big w)**

Calculates the product of two modular exponentiations involving  $n$ -residues.

**Parameters:**

$\leftarrow x$  An  $n$ -residue number  
 $\leftarrow y$  A big integer  
 $\leftarrow a$  An  $n$ -residue number  
 $\leftarrow b$  A big integer

$\rightarrow w = xy \text{ ab (mod } n\text{)},$  where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of the appropriate parameters to  $n$ -residue form. Note that the exponents are not converted to  $n$ -residue form

**See also:**

`nres_powltr`, `nres_powmod`

**2.3.1.16 void nres\_powmodn (int n, big \* x, big \* y, big w)**

Calculates the product of  $n$  modular exponentiations involving  $n$ -residues. Extra memory is allocated internally by this function

**Parameters:**

$\leftarrow n$  The number of  $n$ -residue numbers  
 $\leftarrow x$  An array of  $n$   $n$ -residue numbers  
 $\leftarrow y$  An array of  $n$  big integers  
 $\rightarrow w = x[0]y[0]x[1]y[1] \cdots x[n-1]y[n-1] \text{ (mod } p\text{)},$  where  $p$  is the current Montgomery modulus

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of the appropriate parameters to  $n$ -residue form. Note that the exponents are not converted to  $n$ -residue forms

**2.3.1.17 void nres\_premult (big x, int k, big w)**

Multiplies an  $n$ -residue by a small integer

**Parameters:**

$\leftarrow x$   
 $\leftarrow k$   
 $\rightarrow w = kx \text{ (mod } n\text{)},$  where  $n$  is the current Montgomery modulus

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of the first parameter to  $n$ -residue form. Note that the small integer is not converted to  $n$ -residue form

**See also:**

`nres_modmult`

**2.3.1.18 BOOL nres\_sqroot (big x, big w)**

Calculates the square root of an  $n$ -residue mod a prime modulus

**Parameters:**

$\leftarrow x$   
 $\rightarrow w = x \text{ (mod } n\text{)},$  where  $n$  is the current Montgomery modulus

**Returns:**

TRUE if the square root exists, otherwise FALSE

**Precondition:**

Must be preceded by call to `prepare_monty()` and conversion of the first parameter to  $n$ -residue form

**2.3.1.19 mr\_small prepare\_monty (big n)**

Prepares a Montgomery modulus for use. Each call to this function replaces the previous modulus (if any)

**Parameters:**

$\leftarrow n$  A big number which is to be the Montgomery modulus

**Returns:**

???

**Precondition:**

The parameter  $n$  must be positive and odd. Allocated memory is freed when the current instance of MIRACL is terminated by a call to `mirexit()`

### 2.3.1.20 void redc (big x, big y)

Converts an  $n$ -residue back to normal form

#### Parameters:

$\leftarrow x$  an  $n$ -residue  
 $\rightarrow y$  the normal form of the  $n$ -residue  $x$

#### Precondition:

Must be preceded by call to `prepare_monty()`

#### See also:

`nres`

## 2.4 ZZn2 arithmetic routines

### Functions

- `void zzn2_add (zzn2 *x, zzn2 *y, zzn2 *w)`
- `BOOL zzn2_compare* (zzn2 *x, zzn2 *y)`
- `void zzn2_conj (zzn2 *x, zzn2 *w)`
- `void zzn2_copy* (zzn2 *x, zzn2 *w)`
- `void zzn2_from_big (big x, zzn2 *w)`
- `void zzn2_from_bigs (big x, big y, zzn2 *w)`
- `void zzn2_from_int (int i, zzn2 *w)`
- `void zzn2_from_ints (int i, int j, zzn2 *w)`
- `void zzn2_from_zzn (big x, zzn2 *w)`
- `void zzn2_from_zzns (big x, big y, zzn2 *w)`
- `void zzn2_imul (zzn2 *x, int y, zzn2 *w)`
- `void zzn2_inv (zzn2 *w)`
- `BOOL zzn2_issunity (zzn2 *x)`
- `BOOL zzn2_iszero* (zzn2 *x)`
- `void zzn2_mul (zzn2 *x, zzn2 *y, zzn2 *w)`
- `void zzn2_negate (zzn2 *x, zzn2 *w)`
- `void zzn2_sadd (zzn2 *x, big y, zzn2 *w)`
- `void zzn2_smul (zzn2 *x, big y, zzn2 *w)`
- `void zzn2_ssub (zzn2 *x, big y, zzn2 *w)`
- `void zzn2_sub (zzn2 *x, zzn2 *y, zzn2 *w)`
- `void zzn2_timesi (zzn2 *u)`
- `void zzn2_zero* (zzn2 *w)`

### 2.4.1 Function Documentation

#### 2.4.1.1 void zzn2\_add (zzn2 \* x, zzn2 \* y, zzn2 \* w)

Adds two `zzn2` variables

#### Parameters:

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow w = x + y$

#### 2.4.1.2 BOOL zzn2\_compare\* (zzn2 \* x, zzn2 \* y)

Compares two `zzn2` variables for equality

#### Parameters:

$\leftarrow x$   
 $\leftarrow y$

#### Returns:

TRUE if  $x = y$ , otherwise FALSE

#### 2.4.1.3 void zzn2\_conj (zzn2 \*x, zzn2 \*w)

Finds the conjugate of a zzn2

Parameters:

$\leftarrow X$   
 $\rightarrow W$  If  $x = a + bi$ , then  $w = a - bi$

#### 2.4.1.4 void zzn2\_copy\* (zzn2 \* x, zzn2 \* w)

Copies one zzn2 to another

Parameters:

$\leftarrow X$   
 $\rightarrow W = x$

#### 2.4.1.5 void zzn2\_from\_big (big x, zzn2 \* w)

Creates a zzn2 from a big integer. This is converted internally into n-residue format

Parameters:

$\leftarrow X$   
 $\rightarrow W = x$

#### 2.4.1.6 void zzn2\_from\_bigs (big x, big y, zzn2 \* w)

Creates a zzn2 from two big integers. These are converted internally into n-residue format

Parameters:

$\leftarrow X$   
 $\leftarrow Y$   
 $\rightarrow W = x + yi$

#### 2.4.1.7 void zzn2\_from\_int (int i, zzn2 \* w)

Converts an integer to zzn2 format

Parameters:

$\leftarrow i$   
 $\rightarrow W = i$

See also:

zzn2\_from\_ints

#### 2.4.1.8 void zzn2\_from\_ints (int i, int j, zzn2 \* w)

Creates a zzn2 from two integers

Parameters:

$\leftarrow i$   
 $\leftarrow j$   
 $\rightarrow W = i + j i$

See also:

zzn2\_from\_int

#### 2.4.1.9 void zzn2\_from\_zzn (big x, zzn2 \* w)

Creates a zzn2 from a big already in n-residue format

Parameters:

$\leftarrow X$   
 $\rightarrow W = x$

#### 2.4.1.10 void zzn2\_from\_zzns (big x, big y, zzn2 \* w)

Creates a zzn2 from two bigs already in n-residue format

Parameters:

$\leftarrow X$

```

←y
→W = x + y i

```

See also:

```

zzn2_from_zzn

```

#### 2.4.1.11 void zzn2\_imul (zzn2 \* x, int y, zzn2 \* w)

Multiplies a zzn2 variable by an integer

Parameters:

```

←x
←y
→W = xy

```

#### 2.4.1.12 void zzn2\_inv (zzn2 \* w)

In-place inversion of a zzn2 variable

Parameters:

```

←→W = 1 / w

```

#### 2.4.1.13 BOOL zzn2\_isunity (zzn2 \* x)

Tests a zzn2 value for equality to one

Parameters:

```

←x

```

Returns:

TRUE if x is one, otherwise FALSE

#### 2.4.1.14 BOOL zzn2\_iszero\* (zzn2 \* x)

Tests a zzn2 value for equality to zero

Parameters:

```

←x

```

Returns:

TRUE if x is zero, otherwise FALSE

#### 2.4.1.15 void zzn2\_mul (zzn2 \* x, zzn2 \* y, zzn2 \* w)

Multiplies two zzn2 variables. If x and y are the same variable, a faster squaring method is used

Parameters:

```

←x
←y
→W = xy

```

#### 2.4.1.16 void zzn2\_negate (zzn2 \* x, zzn2 \* w)

Negates a zzn2

Parameters:

```

←x
→W = -x

```

#### 2.4.1.17 void zzn2\_sadd (zzn2 \* x, big y, zzn2 \* w)

Adds a big in n-residue format to a zzn2

Parameters:

```

←x
←y
→W = x + y

```

**2.4.1.18 void zzn2\_smul (zzn2 \* x, big y, zzn2 \* w)**

Multiplies a zzn2 variable by a big in n-residue

Parameters:

```

←x
←y
→W = xy

```

**2.4.1.19 void zzn2\_ssub (zzn2 \* x, big y, zzn2 \* w)**

Subtracts a big in n-residue format from a zzn2

Parameters:

```

←x
←y
→W = x - y

```

**2.4.1.20 void zzn2\_sub (zzn2 \* x, zzn2 \* y, zzn2 \* w)**

Subtracts two zzn2 variables

Parameters:

```

←x
←y
→W = x - y

```

**2.4.1.21 void zzn2\_timesi (zzn2 \* u)**

In-place multiplication of a zzn2 by i, the imaginary square root of the quadratic non-residue

Parameters:

```

←→u If u = a + bi then on exit u = i2b + ai

```

**2.4.1.22 void zzn2\_zero\* (zzn2 \* w)**

Sets a zzn2 variable to zero

Parameters:

```

→w = 0

```

**2.5 Encryption routines****Functions**

- mr\_unsign32 aes\_decrypt\* (aes \*a, char \*buff)
- mr\_unsign32 aes\_encrypt\* (aes \*a, char \*buff)
- void aes\_end\* (aes \*a)
- void aes\_getreg\* (aes \*a, char \*ir)
- BOOL aes\_init\* (aes \*a, int mode, int nk, char \*key, char \*iv)
- void aes\_reset\* (aes \*a, int mode, char \*iv)
- void shs256\_hash\* (sha256 \*sh, char hash[32])
- void shs256\_init\* (sha256 \*sh)
- void shs256\_process\* (sha256 \*sh, int byte)
- void shs384\_hash\* (sha384 \*sh, char hash[48])
- void shs384\_init\* (sha384 \*sh)
- void shs384\_process\* (sha384 \*sh, int byte)
- void shs512\_hash\* (sha512 \*sh, char hash[64])
- void shs512\_init\* (sha512 \*sh)
- void shs512\_process\* (sha512 \*sh, int byte)
- void shs\_hash\* (sha \*sh, char hash[20])
- void shs\_init\* (sha \*sh)
- void shs\_process\* (sha \*sh, int byte)
- void strong\_bigdig (csprng \*rng, int n, int b, big x)
- void strong\_bigrand (csprng \*rng, big w, big x)
- void strong\_init\* (csprng \*rng, int rawlen, char \*raw, mr\_unsign32 tod)
- void strong\_kill\* (csprng \*rng)
- int strong\_rng\* (csprng \*rng)

**2.5.1 Function Documentation**

**2.5.1.1 mr\_unsign32 aes\_decrypt\* (aes \* a, char \* buff)**

Decrypts a 16 or n byte input buffer in situ. If the mode of operation is as a block cipher (MR\_ECB or MR\_CBC) then 16 bytes will be decrypted. If the mode of operation is as a stream cipher (MR\_CFBn, MR\_OFBn or MR\_PCFBn) then n bytes will be decrypted

**Parameters:**

←a Pointer to an initialised instance of an aes structured defined in miracl.h  
 ←→buff Pointer to the buffer of bytes to be decrypted

**Returns:**

If MR\_CFBn and MR\_PCFBn modes then n byte(s) that were shifted off the end of the input register as result of decrypting the n input byte(s), otherwise 0

**Precondition:**

Must be preceded by call to aes\_init()

**2.5.1.2 mr\_unsign32 aes\_encrypt\* (aes \* a, char \* buff)**

Encrypts a 16 or n byte input buffer in situ. If the mode of operation is as a block cipher (MR\_ECB or MR\_CBC) then 16 bytes will be encrypted. If the mode of operation is as a stream cipher (MR\_CFBn, MR\_OFBn or MR\_PCFBn) then n bytes will be encrypted

**Parameters:**

←a Pointer to an initialised instance of an aes structure defined in miracl.h  
 ←→buff Pointer to the buffer of bytes to be encrypted

**Returns:**

In MR\_CFBn and MR\_PCFBn modes the n byte(s) that were shifted off the end of the input register as result of encrypting the n input byte(s), otherwise 0

**Precondition:**

Must be preceded by a call to aes\_init()

**2.5.1.3 void aes\_end\* (aes \* a)**

Ends an AES encryption session, and de-allocates the memory associated with it. The internal session key data is destroyed

**Parameters:**

←→a Pointer to an initialised instance of an aes structured defined in miracl.h

**2.5.1.4 void aes\_getreg\* (aes \* a, char \* ir)**

Reads the current contents of the input chaining register associated with this instance of the AES. This is the register initialised by the IV in the calls to aes\_init() and aes\_reset()

**Parameters:**

←a Pointer to an instance of the aes structured, defined in miracl.h  
 →ir A character array to hold the extracted 16-byte data

**Precondition:**

Must be preceded by a call to aes\_init()

**2.5.1.5 BOOL aes\_init\* (aes \* a, int mode, int nk, char \* key, char \* iv)**

Initialises an Encryption/Decryption session using the Advanced Encryption Standard (AES). This is a block cipher system that encrypts data in 128-bit blocks using a key of 128, 192 or 256 bits. See [Stinson] for more background on block ciphers.

**Parameters:**

→a Pointer to an instance of the aes structure defined in miracl.h  
 ←mode The mode of operation to be used: MR\_ECB (Electronic Code Book), MR\_CBC (Cipher Block Chaining), MR\_CFBn (Cipher Feed-Back where n is 1, 2 or 4), MR\_PCFBn (error Propagating Cipher Feed-Back where n is 1, 2 or 4) or MR\_OFBn (Output Feed-Back where n is 1, 2, 4, 8 or 16). The value of n indicates the number of bytes to be processed in each application. For more information on Modes of Operation, see [Stinson]. MR\_PCFBn is an invention of our own [Scott93]

←nk The size of the key in bytes. It can be either 16, 24 or 32  
 ←key A pointer to the key  
 ←iv A pointer to the Initialisation Vector (IV). A 16-byte initialisation vector should be specified for all modes other than MR\_ECB, in which case it can be NULL

**Returns:**

TRUE if successful, otherwise FALSE

**2.5.1.6 void aes\_reset\* (aes \* a, int mode, char \* iv)**

Resets the AES structure

**Parameters:**

←a Pointer to an instance of the aes structure defined in miracl.h  
 ←mode An Indication of the new mode of operation  
 ←iv A pointer to a (possibly new) initialisation vector

**2.5.1.7 void shs256\_hash\* (sha256 \* sh, char hash[32])**

Generates a 32 byte (256 bit) hash value into the provided array

**Parameters:**

←sh Pointer to the current instance  
 →hash Pointer to array to be filled

**2.5.1.8 void shs256\_init\* (sha256 \* sh)**

Initialises an instance of the Secure Hash Algorithm (SHA-256). Must be called before new use

**Parameters:**

→sh Pointer to an instance of a structure defined in miracl.h

**2.5.1.9 void shs256\_process\* (sha256 \* sh, int byte)**

Processes a single byte. Typically called many times to provide input to the hashing process. The hash value of all the processed bytes can be retrieved by a subsequent call to shs256\_hash()

**Parameters:**

←sh Pointer to the current instance  
 ←byte Character to be processed

**2.5.1.10 void shs384\_hash\* (sha384 \* sh, char hash[48])**

Generates a 48 byte (384 bit) hash value into the provided array

**Parameters:**

←sh Pointer to the current instance  
 →hash Pointer to array to be filled

**2.5.1.11 void shs384\_init\* (sha384 \* sh)**

Initialises an instance of the Secure Hash Algorithm (SHA-384). Must be called before new use

**Parameters:**

! sh Pointer to an instance of a structure defined in miracl.h

**Precondition:**

The SHA-384 algorithm is only available if 64-bit data-type is defined

**2.5.1.12 void shs384\_process\* (sha384 \* sh, int byte)**

Processes a single byte. Typically called many times to provide input to the hashing process. The hash value of all the processed bytes can be retrieved by a subsequent call to shs384\_hash()

**Parameters:**

←sh Pointer to the current instance  
 ←byte Character to be processed

**2.5.1.13 void shs512\_hash\* (sha512 \* sh, char hash[64])**



Generates a 64 byte (512 bit) hash value into the provided array

**Parameters:**

←sh Pointer to the current instance  
→hash Pointer to array to be filled

#### 2.5.1.14 void shs512\_init\* (sha512 \* sh)

Initialises an instance of the Secure Hash Algorithm (SHA-512). Must be called before new use

**Parameters:**

→sh Pointer to an instance of a structure defined in miracl.h

**Precondition:**

The SHA-512 algorithm is only available if 64-bit data-type is defined

#### 2.5.1.15 void shs512\_process\* (sha512 \* sh, int byte)

Processes a single byte. Typically called many times to provide input to the hashing process. The hash value of all the processed bytes can be retrieved by a subsequent call to shs512\_hash()

**Parameters:**

←sh Pointer to the current instance  
←byte Character to be processed

#### 2.5.1.16 void shs\_hash\* (sha \* sh, char hash[20])

Generates a twenty byte (160 bit) hash value into the provided array

**Parameters:**

←sh Pointer to the current instance  
→hash Pointer to array to be filled

#### 2.5.1.17 void shs\_init\* (sha \* sh)

Initialises an instance of the Secure Hash Algorithm (SHA-1). Must be called before new use

**Parameters:**

→sh Pointer to an instance of a structure defined in miracl.h

#### 2.5.1.18 void shs\_process\* (sha \* sh, int byte)

Processes a single byte. Typically called many times to provide input to the hashing process. The hash value of all the processed bytes can be retrieved by a subsequent call to shs\_hash()

**Parameters:**

←sh Pointer to the current instance  
←byte Character to be processed

#### 2.5.1.19 void strong\_bigdig (csprng \* rng, int n, int b, big x)

Generates a big random number of given length from the cryptographically strong generator rng

**Parameters:**

←rng A pointer to the random number generator  
←n  
←b  
→x Big random number n digits long to base b

**Precondition:**

The base b must be printable, that is  $2 \leq b \leq 256$

#### 2.5.1.20 void strong\_bigrand (csprng \* rng, big w, big x)

Generates a cryptographically strong random big number x using the random number generator rng such that  $0 \leq x < w$

**Parameters:**

←rng A pointer to the current instance  
←w

→x

**2.5.1.21 void strong\_init\* (csprng \* rng, int rawlen, char \* raw, mr\_unsign32 tod)**

Initialises the cryptographically strong random number generator rng. The array raw (of length rawlen) and the time-of-day value tod are the two sources used together to seed the generator. The former might be provided from random keystrokes, the latter from an internal clock. Subsequent calls to strong\_rng() will provide random bytes

**Parameters:**

→rng  
 ←rawlen  
 ←raw An array of length rawlen  
 ←tod A 32-bit time-of-day value

**2.5.1.22 void strong\_kill\* (csprng \* rng)**

Kills the internal state of the random number generator rng

**Parameters:**

←rng A pointer to a random number generator

**2.5.1.23 int strong\_rng\* (csprng \* rng)**

Generates a sequence of cryptographically strong random bytes

**Parameters:**

←rng A pointer to a random number generator

**Returns:**

A random byte

**2.6 Elliptic curve routines****Functions**

- void ebrick2\_end\* (ebbrick2 \*B)
- BOOL ebrick2\_init (ebbrick2 \*B, big x, big y, big a2, big a6, int m, int a, int b, int c, int window, int nb)
- void ebrick\_end\* (ebbrick \*B)
- BOOL ebrick\_init (ebbrick \*B, big x, big y, big a, big b, big n, int window, int nb)
- big ecurve2\_add (epoint \*p, epoint \*pa)
- BOOL ecurve2\_init (int m, int a, int b, int c, big a2, big a6, BOOL check, int type)
- void ecurve2\_mult (big e, epoint \*pa, epoint \*pt)
- void ecurve2\_mult2 (big e, epoint \*p, big ea, epoint \*pa, epoint \*pt)
- void ecurve2\_multi\_add (int m, epoint \*\*x, epoint \*\*w)
- void ecurve2\_multn (int n, big \*y, epoint \*\*x, epoint \*w)
- big ecurve2\_sub (epoint \*p, epoint \*pa)
- big ecurve\_add (epoint \*p, epoint \*pa)
- void ecurve\_init (big a, big b, big p, int type)
- void ecurve\_mult (big e, epoint \*pa, epoint \*pt)
- void ecurve\_mult2 (big e, epoint \*p, big ea, epoint \*pa, epoint \*pt)
- void ecurve\_multi\_add (int m, epoint \*\*x, epoint \*\*w)
- void ecurve\_multn (int n, big \*y, epoint \*\*x, epoint \*w)
- big ecurve\_sub (epoint \*p, epoint \*pa)
- BOOL epoint2\_comp (epoint \*a, epoint \*b)
- void epoint2\_copy\* (epoint \*a, epoint \*b)
- int epoint2\_get (epoint \*p, big x, big y)
- void epoint2\_getxyz (epoint \*p, big x, big y, big z)
- BOOL epoint2\_norm (epoint \*p)
- BOOL epoint2\_set (big x, big y, int cb, epoint \*p)
- BOOL epoint\_comp (epoint \*a, epoint \*b)
- void epoint\_copy\* (epoint \*a, epoint \*b)
- void epoint\_free\* (epoint \*p)
- int epoint\_get (epoint \*p, big x, big y)
- void epoint\_getxyz (epoint \*p, big x, big y, big z)
- epoint epoint\_init (void)
- epoint epoint\_init\_mem (char \*mem, int index)
- BOOL epoint\_norm (epoint \*p)
- BOOL epoint\_set (big x, big y, int cb, epoint \*p)
- BOOL epoint\_x (big x)

- int mul2\_brick (ebrick2 \*B, big e, big x, big y)
- int mul\_brick (ebrick \*B, big e, big x, big y)
- BOOL point\_at\_infinity\* (epoint \*p)

## 2.6.1 Function Documentation

### 2.6.1.1 void ebrick2\_end\* (ebrick2 \* B)

Cleans up after an application of the Comb for GF(2m) elliptic curves

#### Parameters:

←→B A pointer to the current instance

### 2.6.1.2 BOOL ebrick2\_init (ebrick2 \* B, big x, big y, big a2, big a6, int m, int a, int b, int c, int window, int nb)

Initialises an instance of the Comb method for GF(2m) elliptic curve multiplication with precomputation.

The field is defined with respect to the trinomial basis  $tm+ta+1$  or the pentanomial basis  $tm+ta+tb+tc+1$ .

Internally memory is allocated for 2w elliptic curve points which will be precomputed and stored. For bigger w more space is required, but the exponentiation is quicker. Try  $w = 8$

#### Parameters:

←B A pointer to the current instance  
 ←x x coordinate of the fixed point  
 ←y y coordinate of the fixed point  
 ←a2 The a2 coefficient of the curve  $y^2 + xy = x^3 + a_2x^2 + a_6$   
 ←a6 the a6 coefficient of the curve  $y^2 + xy = x^3 + a_2x^2 + a_6$   
 ←m  
 ←a  
 ←b  
 ←c  
 ←window The size w of the window  
 ←nb The maximum number of bits to be used in the exponent

#### Returns:

TRUE if successful, otherwise FALSE

#### Note:

If MR\_STATIC is defined in mirdef.h, then the x and y parameters in this function are replaced by a single mr\_small \* pointer to a precomputed table. In this case the function returns a void

### 2.6.1.3 void ebrick\_end\* (ebrick \* B)

Cleans up after an application of the Comb for GF(p) elliptic curves

#### Parameters:

←→B A pointer to the current instance

### 2.6.1.4 BOOL ebrick\_init (ebrick \* B, big x, big y, big a, big b, big n, int window, int nb)

Initialises an instance of the Comb method for GF(p) elliptic curve multiplication with precomputation.

Internally memory is allocated for 2w elliptic curve points which will be precomputed and stored. For bigger w more space is required, but the exponentiation is quicker. Try  $w = 8$

#### Parameters:

→B A pointer to the current instance  
 ←x x coordinate of the fixed point  
 ←y y coordinate of the fixed point  
 ←a The a coefficient of the curve  $y^2 = x^3 + ax + b$   
 ←b The b coefficient of the curve  $y^2 = x^3 + ax + b$   
 ←n The modulus  
 ←window The size w of the window  
 ←nb The maximum number of bits to be used in the exponent

#### Returns:

TRUE if successful, otherwise FALSE

#### Note:

If MR\_STATIC is defined in mirdef.h, then the x and y parameters in this function are replaced by a

single `mr_small` \* pointer to a precomputed table. In this case the function returns a void.

#### 2.6.1.5 `big ecurve2_add (epoint * p, epoint * pa)`

Adds two points on a GF(2m) elliptic curve using the special rule for addition. Note that if `pa = p`, then a different duplication rule is used. Addition is quicker if `p` is normalised

##### Parameters:

←p  
←→pa = pa + p

##### Returns:

An ephemeral pointer to the slope if curve is super-singular

##### Precondition:

The input points must actually be on the current active curve

#### 2.6.1.6 `BOOL ecurve2_init (int m, int a, int b, int c, big a2, big a6, BOOL check, int type)`

Initialises the internal parameters of the current active GF(2m) elliptic curve. The curve is assumed to be of the form  $y^2 + xy = x^3 + Ax^2 + B$ . The field is defined with respect to the trinomial basis  $t^m + ta + 1$  or the pentanomial basis  $t^m + ta + tb + tc + 1$ . This routine can be called subsequently with the parameters of a different curve

##### Parameters:

←m  
←a  
←b  
←c  
←a2 The A coefficient on the elliptic curve equation  
←a6 The B coefficient on the elliptic curve equation  
←check If TRUE a check is made that the specified basis is irreducible. If FALSE, this basis validity check, which is time-consuming, is suppressed  
←type Either `MR_PROJECTIVE` or `MR_AFFINE`, specifying whether projective or affine coordinates should be used internally. Normally the former is faster

##### Returns:

TRUE if parameters make sense, otherwise FALSE

##### Warning:

Allocated memory will be freed when the current instance of MIRACL is terminated by a call to `mirexit()`. Only one elliptic curve, GF(p) or GF(2m) may be active within a single MIRACL instance

#### 2.6.1.7 `void ecurve2_mult (big e, epoint * pa, epoint * pt)`

Multiplies a point on a GF(2m) elliptic curve by an integer. Uses the addition/subtraction method

##### Parameters:

←e  
←pa  
→pt = e × pa

##### Precondition:

The point `pa` must be on the active curve

#### 2.6.1.8 `void ecurve2_mult2 (big e, epoint * p, big ea, epoint * pa, epoint * pt)`

Calculates the point  $e \times p + ea \times pa$  on a GF(2m) elliptic curve. This is quicker than doing two separate multiplications and an addition. Useful for certain cryptosystems

##### Parameters:

←e  
←p  
←ea  
←pa  
→pt = e × p + ea × pa

##### Precondition:

The points p and pa must be on the active curve

#### 2.6.1.9 void ecurve2\_multi\_add (int m, epoint \*\* x, epoint \*\* w)

Simultaneously adds pairs of points on the active GF(2m) curve. This is much quicker than adding them individually, but only when using affine coordinates

Parameters:

←m  
 ←x  
 →w w[i] = w[i] + x[i] for i = 0 to m - 1

Note:

Only useful when using affine coordinates

See also:

ecurve2\_init

#### 2.6.1.10 void ecurve2\_multn (int n, big \* y, epoint \*\* x, epoint \* w)

Calculates the point  $x[0]y[0] + x[1]y[1] + \dots + x[n-1]y[n-1]$  on a GF(2m) elliptic curve, for  $n \geq 2$

Parameters:

←n  
 ←y an array of n big numbers  
 ←x an array of n elliptic curve points  
 →w =  $x[0]y[0] + x[1]y[1] + \dots + x[n-1]y[n-1]$

Precondition:

The points must be on the active curve. The y[] values must all be positive. The underlying number base must be a power of 2

#### 2.6.1.11 big ecurve2\_sub (epoint \* p, epoint \* pa)

Subtracts two points on a GF(2m) elliptic curve. Actually negates p and adds it to pa. Subtraction is quicker if p is normalised.

Parameters:

←p  
 ←→ pa = pa - p

Returns:

An ephemeral pointer to the slope

Precondition:

The input points must actually be on the current active curve

#### 2.6.1.12 big ecurve\_add (epoint \* p, epoint \* pa)

Adds two points on a GF(p) elliptic curve using the special rule for addition. Note that if pa = p, then a different duplication rule is used. Addition is quicker if p is normalised

Parameters:

←p  
 ←→ pa = pa + p

Returns:

An ephemeral pointer to the slope

Precondition:

The input points must actually be on the current active curve

#### 2.6.1.13 void ecurve\_init (big a, big b, big p, int type)

Initialises the internal parameters of the current active GF(p) elliptic curve. The curve is assumed to be of the form  $y^2 = x^3 + Ax + B \pmod{p}$ , the so-called Weierstrass model. This routine can be called subsequently with the parameters of a different curve

**Parameters:**

$\leftarrow$ a The A coefficient of the elliptic curve  
 $\leftarrow$ b The B coefficient of the elliptic curve  
 $\leftarrow$ p The modulus  
 $\rightarrow$ type Either MR\_PROJECTIVE or MR\_AFFINE, specifying whether projective or affine coordinates should be used internally. Normally the former is faster

**Warning:**

Allocated memory will be freed when the current instance of MIRACL is terminated by a call to mirexit(). Only one elliptic curve, GF(p) or GF(2m) may be active within a single MIRACL instance

**2.6.1.14 void ecurve\_mult (big e, epoint \* pa, epoint \* pt)**

Multiplies a point on a GF(p) elliptic curve by an integer. Uses the addition/subtraction method

**Parameters:**

$\leftarrow$ e  
 $\leftarrow$ pa  
 $\rightarrow$ pt =  $e \times pa$

**Precondition:**

The point pa must be on the active curve

**2.6.1.15 void ecurve\_mult2 (big e, epoint \* p, big ea, epoint \* pa, epoint \* pt)**

Calculates the point  $e \times p + ea \times pa$  on a GF(p) elliptic curve. This is quicker than doing two separate multiplications and an addition. Useful for certain cryptosystems

**Parameters:**

$\leftarrow$ e  
 $\leftarrow$ p  
 $\leftarrow$ ea  
 $\leftarrow$ pa  
 $\rightarrow$ pt =  $e \times p + ea \times pa$

**Precondition:**

The points p and pa must be on the active curve

**2.6.1.16 void ecurve\_multi\_add (int m, epoint \*\* x, epoint \*\* w)**

Simultaneously adds pairs of points on the active GF(p) curve. This is much quicker than adding them individually, but only when using affine coordinates

**Parameters:**

$\leftarrow$ m  
 $\leftarrow$ x  
 $\rightarrow$ w  $w[i] = w[i] + x[i]$  for  $i = 0$  to  $m - 1$

**Note:**

Only useful when using affine coordinates

**See also:**

ecurve\_init, nres\_multi\_inverse

**2.6.1.17 void ecurve\_multn (int n, big \* y, epoint \*\* x, epoint \* w)**

Calculates the point  $x[0]y[0] + x[1]y[1] + \dots + x[n-1]y[n-1]$  on a GF(p) elliptic curve, for  $n \geq 2$

**Parameters:**

$\leftarrow$ n  
 $\leftarrow$ y An array of n big numbers  
 $\leftarrow$ x An array of n elliptic curve points  
 $\rightarrow$ w =  $x[0]y[0] + x[1]y[1] + \dots + x[n-1]y[n-1]$

**Precondition:**

The points must be on the active curve. The  $y[]$  values must all be positive. The underlying number base must be a power of 2

**2.6.1.18 big ecurve\_sub (epoint \* p, epoint \* pa)**

Subtracts two points on a GF(p) elliptic curve. Actually negates p and adds it to pa. Subtraction is quicker if p is normalised.

**Parameters:**

←p  
 ←→pa = pa - p

**Returns:**

An ephemeral pointer to the sline slope

**Precondition:**

The input points must actually be on the current active curve

**2.6.1.19 BOOL epoint2\_comp (epoint \* a, epoint \* b)**

Compares two points on the current active GF(2m) elliptic curve

**Parameters:**

←a  
 ←b

**Returns:**

TRUE if the points are the same, otherwise FALSE

**2.6.1.20 void epoint2\_copy\* (epoint \* a, epoint \* b)**

Copies one point to another on a GF(2m) elliptic curve

**Parameters:**

←a  
 ←b = a

**2.6.1.21 int epoint2\_get (epoint \* p, big x, big y)**

Normalises a point and extracts its (x,y) coordinates on the active GF(2m) elliptic curve

**Parameters:**

←p  
 →x  
 →y

**Returns:**

The least significant bit of y. Note that it is possible to reconstruct a point from its x coordinate and just the least significant bit of y. Often such a 'compressed' description of a point is useful

**Precondition:**

The point p must be on the active curve

**Note:**

If x and y are not distinct variables on entry then only the value of x is returned

**Example:**

```
i = epoint2_get(p, x, x); // extract x coordinate and lsb of y/x
```

**2.6.1.22 void epoint2\_getxyz (epoint \* p, big x, big y, big z)**

Extracts the raw (x,y,z) coordinates of a point on the active GF(2m) elliptic curve

**Parameters:**

←p  
 →x  
 →y  
 →z

**Precondition:**

The point `p` must be on the active curve

**Note:**

If any of `x`, `y`, `z` is `NULL` then that coordinate is not returned

**2.6.1.23 BOOL `epoint2_norm` (`epoint * p`)**

Normalises a point on the current active GF(2<sup>m</sup>) elliptic curve. This sets the `z` coordinate to 1. Point addition is quicker when adding a normalised point. This function does nothing if affine coordinates are being used (in which case there is no `z` coordinate)

**Parameters:**

←`p` A point on the current active elliptic curve

**Returns:**

TRUE if successful, otherwise FALSE

**2.6.1.24 BOOL `epoint2_set` (`big x`, `big y`, `int cb`, `epoint * p`)**

Sets a point on the current active GF(2<sup>m</sup>) elliptic curve (if possible)

**Parameters:**

←`x` The `x` coordinate of the point

←`y` The `y` coordinate of the point

**2.6 Elliptic curve routines 59**

←`cb` If `x` and `y` are not distinct variables then `x` only is passed to the function, and `cb` is taken as the least significant bit of `y`. In this case the full value of `y` is reconstructed internally. This is known as 'point decompression' (and is a bit time-consuming, requiring the extraction of a modular square root)

→`p` = (`x`,`y`)

**Returns:**

TRUE if the point exists on the current active elliptic curve, otherwise FALSE

**Example:**

```
p = epoint_init();
epoint2_set(x, x, 1, p); // decompress p
```

**2.6.1.25 BOOL `epoint_comp` (`epoint * a`, `epoint * b`)**

Compares two points on the current active GF(p) elliptic curve

**Parameters:**

←`a`

←`b`

**Returns:**

TRUE if the points are the same, otherwise FALSE

**2.6.1.26 void `epoint_copy`\* (`epoint * a`, `epoint * b`)**

Copies one point to another on a GF(p) elliptic curve

**Parameters:**

←`a`

←`b` = `a`

**2.6.1.27 void `epoint_free`\* (`epoint * p`)**

Frees memory associated with a point on a GF(p) elliptic curve

**Parameters:**

←`p`



**2.6.1.28 int epoint\_get (epoint \* p, big x, big y)**

Normalises a point and extracts its (x,y) coordinates on the active GF(p) elliptic curve

**Parameters:**

←p  
→x  
→y

**Returns:**

The least significant bit of y. Note that it is possible to reconstruct a point from its x coordinate and just the least significant bit of y. Often such a 'compressed' description of a point is useful

**Precondition:**

The point p must be on the active curve

**Note:**

If x and y are not distinct variables on entry then only the value of x is returned

**Example:**

```
i = epoint_get(p, x, x); // extract x coordinate and lsb of y
```

**2.6.1.29 void epoint\_getxyz (epoint \* p, big x, big y, big z)**

Extracts the raw (x,y,z) coordinates of a point on the active GF(p) elliptic curve

**Parameters:**

←p  
→x  
→y  
→z

**Precondition:**

The point p must be on the active curve

**Note:**

If any of x, y, z is NULL then that coordinate is not returned

**2.6.1.30 epoint\* epoint\_init (void)**

Assigns memory to a point on a GF(p) elliptic curve, and initialises it to the 'point at infinity'

**Returns:**

A pointer to an elliptic curve point (in fact a pointer to a structure allocated from the heap)

**Warning:**

It is the C programmer's responsibility to ensure that all elliptic curve points initialised by a call to this function are ultimately freed by a call to epoint\_free(). If not a memory leak will result

**2.6.1.31 epoint\* epoint\_init\_mem (char \* mem, int index)**

Initialises memory for an elliptic curve point from a pre-allocated byte array mem. This array may be created from the heap by a call to ecp\_memalloc(), or in some other way. This is quicker than multiple calls to epoint\_init()

**Parameters:**

←mem  
←index An index into mem. Each index should be unique

**Returns:**

An initialised elliptic curve point

**Precondition:**

Sufficient memory must have been allocated and pointed to by mem

**2.6.1.32 BOOL epoint\_norm (epoint \* p)**

Normalises a point on the current active GF(p) elliptic curve. This sets the z coordinate to 1. Point addition is quicker when adding a normalised point. This function does nothing if affine coordinates are being used (in which case there is no z coordinate)

**Parameters:**

←p A point on the current active elliptic curve

**Returns:**

TRUE if successful, otherwise FALSE

### 2.6.1.33 BOOL epoint\_set (big x, big y, int cb, epoint \* p)

Sets a point on the current active GF(p) elliptic curve (if possible)

**Parameters:**

←x The x coordinate of the point

←y The y coordinate of the point

←cb If x and y are not distinct variables then x only is passed to the function, and cb is taken as the least significant bit of y. In this case the full value of y is reconstructed internally. This is known as 'point decompression' (and is a bit time-consuming, requiring the extraction of a modular square root)

→p = (x,y)

**Returns:**

TRUE if the point exists on the current active elliptic curve, otherwise FALSE

**Example:**

```
p = epoint_init();
epoint_set(x, x, 1, p); // decompress p
```

### 2.6.1.34 BOOL epoint\_x (big x)

Tests to see if the parameter x is a valid coordinate of a point on the curve. It is faster to test an x coordinate first in this way, rather than trying to directly set it on the curve by calling epoint\_set(), as it avoids an expensive modular square root

**Parameters:**

←x The integer coordinate x

**Returns:**

TRUE if x is the coordinate of a curve point, otherwise FALSE

### 2.6.1.35 int mul2\_brick (ebrick2 \* B, big e, big x, big y)

Carries out a GF(2m) elliptic curve multiplication using the precomputed values stored in the ebrick structure

**Parameters:**

←B A pointer to the current instance

←e A big exponent

→x The x coordinate of  $e \times G$ , where G is specified in the initial call to ebrick2\_init()

→y The y coordinate of  $e \times G$ , where G is specified in the initial call to ebrick2\_init()

**Returns:**

The least significant bit of y

**Note:**

If x and y are not distinct variables, only x is returned

**Precondition:**

Must be preceded by a call to ebrick2\_init()

### 2.6.1.36 int mul\_brick (ebrick \* B, big e, big x, big y)

Carries out a GF(p) elliptic curve multiplication using the precomputed values stored in the ebrick structure

**Parameters:**

←B A pointer to the current instance

←e A big exponent  
 →x The x coordinate of  $eG \pmod n$ , where G and n are specified in the initial call to `ebrick_init()`  
 →y The y coordinate of  $eG \pmod n$ , where G and n are specified in the initial call to `ebrick_init()`

**Returns:**

The least significant bit of y

**Note:**

If x and y are not distinct variables, only x is returned

**Precondition:**

Must be preceded by a call to `ebrick_init()`

**2.6.1.37 BOOL point\_at\_infinity\* (epoint \* p)**

Tests if an elliptic curve point is the 'point at infinity'

**Parameters:**

←p An elliptic curve point

**Returns:**

TRUE if p is the point at infinity, otherwise FALSE

**Precondition:**

The point must be initialised

**2.7 Floating-slash routines****Functions**

- void build (flash x, int(\*gen)(\_MIPT\_ big, int))
- void dconv (double d, flash w)
- void denom (flash x, big y)
- void facos (flash x, flash y)
- void facosh (flash x, flash y)
- void fadd (flash x, flash y, flash z)
- void fasin (flash x, flash y)
- void fasinsh (flash x, flash y)
- void fatan (flash x, flash y)
- void fatanh (flash x, flash y)
- int fcomp (flash x, flash y)
- void fconv (int n, int d, flash x)
- void fcos (flash x, flash y)
- void fcosh (flash x, flash y)
- void fdiv (flash x, flash y, flash z)
- double fsize (flash w)
- void fexp (flash x, flash y)
- void fincr (flash x, int n, int d, flash y)
- void flog (flash x, flash y)
- void flop (flash x, flash y, int \*op, flash z)
- void fmodulo (flash x, flash y, flash z)
- void fmul (flash x, flash y, flash z)
- void fpack (big n, big d, flash x)
- void fpi (flash pi)
- void fpmul (flash x, int n, int d, flash y)
- void fpower (flash x, int n, flash w)
- void fpowf (flash x, flash y, flash z)
- void frand (flash x)
- void frecip (flash x, flash y)
- BOOL froot (flash x, int n, flash w)
- void fsin (flash x, flash y)
- void fsinh (flash x, flash y)
- void fsub (flash x, flash y, flash z)
- void ftan (flash x, flash y)
- void ftanh (flash x, flash y)
- void ftrunc (flash x, big y, flash z)
- void mround (big num, big den, flash z)
- void numer (flash x, big y)

## 2.7.1 Function Documentation

### 2.7.1.1 void build (flash x, int(\*)(\_MIPT\_ big, int) gen)

Uses supplied generator of regular continued fraction expansion to build up a flash number x, rounded if necessary

#### Parameters:

→x The flash number created  
←gen The generator function

#### Example:

```
int phi(flash w, int n)
{
    // rcf generator for golden ratio //
    return 1;
}
...
build(x, phi);
...
// This will calculate the golden ratio (1 + sqrt(5)) / 2 in x -- very quickly!
```

### 2.7.1.2 void dconv (double d, flash w)

Converts a double to flash format

#### Parameters:

←d  
→w The flash equivalent of d

### 2.7.1.3 void denom (flash x, big y)

Extracts the denominator of a flash number

#### Parameters:

←x  
→y The denominator of x

### 2.7.1.4 void facos (flash x, flash y)

Calculates arc-cosine of a flash number, using fasin()

#### Parameters:

←x  
→y = arccos(x)

#### Precondition:

|x| must be less than or equal to 1

### 2.7.1.5 void facosh (flash x, flash y)

Calculates hyperbolic arc-cosine of a flash number

#### Parameters:

←x  
→y = arccosh(x)

#### Precondition:

|x| must be greater than or equal to 1

### 2.7.1.6 void fadd (flash x, flash y, flash z)

Adds two flash numbers

#### Parameters:

←x  
←y  
→z = x + y

#### 2.7.1.7 void fasin (flash x, flash y)

Calculates arc-sin of a flash number, using fatan()

Parameters:

$\leftarrow x$   
 $\rightarrow y = \arcsin(x)$

Precondition:

$|x|$  must be less than or equal to 1

#### 2.7.1.8 void fasinH (flash x, flash y)

Calculates hyperbolic arc-sin of a flash number

Parameters:

$\leftarrow x$   
 $\rightarrow y = \operatorname{arcsinh}(x)$

#### 2.7.1.9 void fatan (flash x, flash y)

Calculates the arc-tangent of a flash number, using an  $O(n2.5)$  method based on Newton's iteration

Parameters:

$\leftarrow x$   
 $\rightarrow y = \arctan(x)$

#### 2.7.1.10 void fatanh (flash x, flash y)

Calculates the hyperbolic arc-tangent of a flash number

Parameters:

$\leftarrow x$   
 $\rightarrow y = \operatorname{arctanh}(x)$

Precondition:

$x^2$  must be less than 1

#### 2.7.1.11 int fcomp (flash x, flash y)

Compares two flash numbers

Parameters:

$\leftarrow x$   
 $\leftarrow y$

Returns:

-1 if  $y > x$ , +1 if  $x > y$  and 0 if  $x = y$

#### 2.7.1.12 void fconv (int n, int d, flash x)

Converts a simple fraction to flash format

Parameters:

$\leftarrow n$   
 $\leftarrow d$   
 $\rightarrow x = n/d$

#### 2.7.1.13 void fcos (flash x, flash y)

Calculates cosine of a given flash angle, using ftan()

Parameters:

$\leftarrow x$   
 $\rightarrow y = \cos(x)$

#### 2.7.1.14 void fcosh (flash x, flash y)

Calculates hyperbolic cosine of a given flash angle

Parameters:

```
←x
→y = cosh(x)
```

#### 2.7.1.15 void fdiv (flash x, flash y, flash z)

Divides two flash numbers

Parameters:

```
←x
←y
→z = x/y
```

#### 2.7.1.16 double fsize (flash w)

Converts a flash number to double format

Parameters:

```
←w
```

Returns:

The value of the parameter x as a double

Precondition:

The value of x must be representable as a double

#### 2.7.1.17 void fexp (flash x, flash y)

Calculates the exponential of a flash number using O(n2.5) method

Parameters:

```
←x
→y = ex
```

#### 2.7.1.18 void fincr (flash x, int n, int d, flash y)

Add a simple fraction to a flash number

Parameters:

```
←x
←n
←d
→y = x + n/d
```

Example:

```
// This subtracts two-thirds from the value of x
fincr(x, -2, 3, x);
```

#### 2.7.1.19 void flog (flash x, flash y)

Calculates the natural log of a flash number using O(n2.5) method

Parameters:

```
←x
→y = log(x)
```

#### 2.7.1.20 void flop (flash x, flash y, int \* op, flash z)

Performs primitive flash operation. Used internally. See source listing comments for more details

Parameters:

```
←x
←y
←op
→z = Fn(x,y), where the function performed depends on the parameter op
```

#### 2.7.1.21 void fmodulo (flash x, flash y, flash z)

Finds the remainder when one flash number is divided by another

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow z = x \pmod{y}$

**2.7.1.22 void fmul (flash x, flash y, flash z)**

Multiplies two flash numbers

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow z = xy$

**2.7.1.23 void fpack (big n, big d, flash x)**

Forms a flash number from big numerator and denominator

**Parameters:**

$\leftarrow n$   
 $\leftarrow d$   
 $\rightarrow x = n/d$

**Precondition:**

The denominator must be non-zero. Flash variable x and big variable d must be distinct. The resulting flash variable must not be too big for the representation

**2.7.1.24 void fpi (flash pi)**

Calculates  $\pi$  using Gauss-Legendre  $O(n^2 \log n)$  method. Note that on subsequent calls to this routine,  $\pi$  is immediately available, as it is stored internally. (This routine is disappointingly slow. There appears to be no simple way to calculate a rational approximation to  $\pi$  quickly)

**Parameters:**

$\rightarrow \text{pi} = \pi$

**Note:**

Internally allocated memory is freed when the current MIRACL instance is ended by a call to mirexit()

**2.7.1.25 void fpmul (flash x, int n, int d, flash y)**

Multiplies a flash number by a simple fraction

**Parameters:**

$\leftarrow x$   
 $\leftarrow n$   
 $\leftarrow d$   
 $\rightarrow y = xn/d$

**2.7.1.26 void fpower (flash x, int n, flash w)**

Raises a flash number to an integer power

**Parameters:**

$\leftarrow x$   
 $\leftarrow n$   
 $\rightarrow w = x^n$

**2.7.1.27 void fpowf (flash x, flash y, flash z)**

Raises a flash number to a flash power

**Parameters:**

$\leftarrow x$   
 $\leftarrow y$   
 $\rightarrow z = xy$

**2.7.1.28 void frand (flash x)**

Generates a random flash number

**Parameters:**

→x A flash random number in the range  $0 < x < 1$

**2.7.1.29 void frecip (flash x, flash y)**

Calculates reciprocal of a flash number

**Parameters:**

←x  
→y =  $1/x$

**2.7.1.30 BOOL froot (flash x, int n, flash w)**

Calculates n-th root of a flash number using Newton's  $O(n^2)$  method

**Parameters:**

←x  
←n  
→w =  $n x$

**Returns:**

TRUE for exact root, otherwise FALSE

**2.7.1.31 void fsin (flash x, flash y)**

Calculates sine of a given flash angle. Uses ftan()

**Parameters:**

←x  
→y =  $\sin(x)$

**2.7.1.32 void fsinh (flash x, flash y)**

Calculates hyperbolic sine of a given flash angle

**Parameters:**

←x  
→y =  $\sinh(x)$

**2.7.1.33 void fsub (flash x, flash y, flash z)**

Subtracts two flash numbers

**Parameters:**

←x  
←y  
→z =  $x - y$

**2.7.1.34 void ftan (flash x, flash y)**

Calculates the tan of a given flash angle, using an  $O(n^2.5)$  method

**Parameters:**

←x  
→y =  $\tan(x)$

**2.7.1.35 void ftanh (flash x, flash y)**

Calculates the hyperbolic tan of a given flash angle

**Parameters:**

←x  
→y =  $\tanh(x)$

**2.7.1.36 void ftrunc (flash x, big y, flash z)**

Separates a flash number to a big number and a flash remainder

**Parameters:**



```

←x
→y = int(x)
→z The fractional remainder. If y is the same as z, only int(x) is returned

```

#### 2.7.1.37 void mround (big num, big den, flash z)

Forms a rounded flash number from big numerator and denominator. If rounding takes place the instance variable EXACT is set to FALSE. EXACT is initialised to TRUE in routine mirsys(). This routine is used internally

##### Parameters:

```

←num
←den
→z = R(num/den)--- the flash number num/den is rounded if necessary to fit the representation

```

##### Precondition:

The denominator must be non-zero

#### 2.7.1.38 void numer (flash x, big y)

Extracts the numerator of a flash number

##### Parameters:

```

←x
→y the numerator of x

```

## 3 MIRACL Data Structure Documentation

### 3.1 MIRACL Structure Reference

MIRACL Instance Pointer.

```
#include <miracl.h>
```

#### Data Fields

- BOOL ERCON
- int ERNUM
- BOOL EXACT
- int INPLEN
- int IOBASE
- int IOBSIZ
- char \* IOBUFF
- int NTRY
- int \* PRIMES
- BOOL RPOINT
- BOOL TRACER

#### 3.1.1 Detailed Description

MIRACL Instance Pointer.

#### 3.1.2 Field Documentation

##### 3.1.2.1 BOOL ERCON

Errors by default generate an error message and immediately abort the program. Alternatively by setting mip->ERCON=TRUE error control is left to the user

##### 3.1.2.2 int ERNUM

Number of the last error that occurred

##### 3.1.2.3 BOOL EXACT

Initialised to TRUE. Set to FALSE if any rounding takes place during flash arithmetic

##### 3.1.2.4 int INPLEN

Length of input string. Must be used when inputting binary data

##### 3.1.2.5 int IOBASE

The 'printable' number base to be used for input and output. May be changed at will within a program. Must be greater than or equal to 2 and less than or equal to 256

**3.1.2.6 int IOBSIZ**

Size of I/O buffer

**3.1.2.7 char\* IOBUFF**

Input/Output buffer

**3.1.2.8 int NTRY**

Number of iterations used in probabilistic primality test by isprime(). Initialised to 6

**3.1.2.9 int\* PRIMES**

Pointer to a table of small prime numbers

**3.1.2.10 BOOL RPOINT**

If set to ON numbers are output with a radix point. Otherwise they are output as fractions (the default)

**3.1.2.11 BOOL TRACER**

If set to ON causes debug information to be printed out, tracing the progress of all subsequent calls to MIRACL routines. Initialised to OFF