M.I.R.A.C.L. Users Manual [1]

Shamus Software Ltd.
4 Foster Place North
Ballybough
Dublin 3
Ireland

August 2006

[1] This manual documents Version 5.2 of the MIRACL library

Abstract

The MIRACL library consists of well over 100 routines that cover all aspects of multi-precision arithmetic. Two new data-types are defined - *big* for large integers and *flash* (short for floating-slash) for large rational numbers. The large integer routines are based on Knuth's algorithms, described in Chapter 4 of his classic work 'The Art of Computer Programming'. Floating-slash arithmetic, which works with rounded fractions, was originally proposed by D. Matula and P. Kornerup. All routines have been thoroughly optimised for speed and efficiency, while at the same time remaining standard, portable C. However optional fast assembly language alternatives for certain time-critical routines are also included, particularly for the popular Intel 80x86 range of processors. A C++ interface is also provided. Full source code is included.

1. Introduction

Remember when as a naive young computer user, you received delivery of your brand new state-of-the-art micro; remember your anticipation at the prospect of the computer power now available at your fingertips; remember recalling all those articles which promised that 'today's microcomputers are as powerful as yesterdays mainframes'. Remember then slowly and laboriously typing in your first program, to calculate, say, 1000! (i.e. 1000 × 999 × 998... ×1) - a calculation unimaginable by hand.

```
10 LET X=1
20 FOR I=1 TO 1000
30 X=X*I
40 NEXT I
50 PRINT X
60 END

RUN
```

After a few seconds the result appeared:-

```
Too big at line 30
```

Remember your disappointment.

Now try the MIRACL approach. MIRACL is a portable C library which implements multiprecision integer and rational data-types, and provides the routines to perform basic arithmetic on them.

Run the program **fact** from the distribution media, and type in 1000. There is your answer - a 2568 digit number.

Now compile and run the program **roots**, and ask it to calculate the square root of 2. Virtually instantly your computer comes back with the value correct to 100+ decimal places. Now that's what I call computing!

Next run the Public Key Cryptography program **enciph**. When it asks the name of a file to be enciphered press return. When it asks for an output filename, type FRED followed by return. Now type in any message, finishing with CONTROL-Z. Your message has been thoroughly enciphered in the file FRED.BLG (type it out and see).

Now run ‘deciph’, and type in FRED. Press return for the requested output filename. Your original message appears on the screen.

This type of encipherment, based as it is on the difficulty of factoring large numbers, offers much greater security and flexibility than more traditional methods.

A useful demonstration of the power of MIRACL is given by the program **ratcalc**, a powerful scientific calculator - accurate to 36 decimal places and with the unusual ability to handle fractions directly.

It is assumed in this manual that the reader is familiar with the C language, and with his/her own computer. On a first reading Chapters 4, 5 and 6 may be safely skipped. Examination of the example programs' source code will be very rewarding.

2. Installation

The MIRACL library has been successfully installed on a VAX11/780, on a variety of UNIX workstations (Sun, SPARC, Next, IBM RS/6000), on an IBM PC using the Microsoft C and C++ compilers, Borland's Turbo C and Borland C++ compilers, the Watcom C compiler and the DJGPP GNU compiler; on ARM based computers, and on an Apple Macintosh. Recently it has been implemented on Itanium and AMD 64-bit processors.

The complete source code for each module in the MIRACL library, and for each of the example programs is provided on the distribution media. Most are written in Standard ANSI C, and should compile using any decent ANSI C compiler. Some modules contain extensive amounts of in-line assembly language, used to optimise performance for certain compiler/processor combinations. However these are invoked transparently by conditional compilation commands and will not interfere with other compilers. The batch files *xxdoit.xxx* contain the commands used for the creation of a library file and the example programs for several compilers. Print out and examine the appropriate file for your configuration.

Pre-compiled libraries for immediate use with certain popular compilers may be found on the distribution media: ready-to-run versions of only some of the example programs may be included, to conserve space.

To create a library you will need access to a compiler, a text editor, a linker, a librarian utility, and an assembler (optional). Read your compiler documentation for further details. The file *mrmuldv.any*, which contains special assembly language versions of the time-critical routines **muldiv**, **muldvd**, **muldvd2** and **muldvm** together with some portable C versions, which may need to be tailored for your configuration. These modules are particularly required if the compiler does not support a double length type which can hold the product of two word-length integers. Most modern compilers do provide this support (often the double length type is called **long long),** and in this case it is often adequate to use the standard C version of this module *mrmuldv.ccc* which can simply be copied to *mrmuldv.c*. Read this manual carefully, and the comments in *mrmuldv.any* for more details.

The hardware/compiler specific file *mirdef.h* needs to be specified. To assist with this, five example versions of the header are supplied: *mirdef.h16* for use with a 16-bit processor, *mirdef.h32* for 32-bit processors, *mirdef.haf* if using a 32-bit processor in a 16-bit mode, and *mirdef.hpc* for pseudo 32-bit working in a 16-bit environment. Note that the full 32-bit version is fastest, but only possible if using a true 32-bit compiler with a 32-bit processor. Try *mirdef.gcc* for use with **gcc** and **g++** in a Unix environment (no assembler).

To assist with the configuration process, a file *config.c* is provided. When compiled and run on the target processor it automatically generates a *mirdef.h* file and gives general advice on configuration. It also generates a *miracl.lst* file with a list of MIRACL modules to be included in the associated library build. Experimentation with this program is strongly encouraged. When compiling this program DO NOT use any compiler optimization.

The *mirdef.h* file contains some optional definitions: Define **MR_NOFULLWIDTH** if you are unable to supply versions of **muldvd**, **muldvd2** and **muldvm** in *mrmuldv.c*. Define **MR_FLASH** if you wish to use *flash* variables in your programs.
Either one of **MR_LITTLE_ENDIAN** or **MR_BIG_ENDIAN** must be defined. The *config.c* program automatically determines which is appropriate for your processor.

By omitting the **MR_FLASH** definition *big* variables can be made much larger, and the library produced will be much smaller, leading to more compact executables. Define **MR_STRIPPED_DOWN** to omit error messages, to save even more space in production code. Use with care!

If you don't want any assembler, define **MR_NOASM**. This generates standard C code for the four time-critical routines, and generates it in-line. This is faster - saves on function calling overhead - and also gives an optimising compiler something to chew on. Note that if **MR_NOASM** is defined, then the *mrmuldv* module is not required in the MIRACL library.

If using the Microsoft Visual C++ tool, some helpful advice can be found in the file *msvisual.txt*. If using the Linux operating system, check out *linux.txt*. Users of the Borland compiler should look at *borland.txt*.

In the majority of cases where pre-built libraries or specific advice in a *.txt* file is not available, the following procedure will result in a successful build of the MIRACL library:-

1. Compile and run *config.c* on the target processor.
2. Rename the generated file *mirdef.tst* to *mirdef.h*
3. If so advised by the **config** program, extract a suitable *mrmuldv.c* file from *mrmuldv.any* (or copy the standard C version *mrmuldv.ccc* to *mrmuldv.c* and use this). If it is pure assembly language it may be appropriate to name it *mrmuldv.s* or *mrmuldv.asm*.
4. If the fast **KCM** or **Comba** methods for modular multiplication were selected (see below), compile and run the *mex.c* utility on any workstation. Use it to automatically generate either the module *mrcomba.c* or *mrkcm.c*. This will require a processor/compiler-specific *xxx.mcs* file. The compiler must support inline assembly.
5. Make sure that all the MIRACL header files are accessible to the compiler. Typically the flag **–I.** or **/I.** allows these headers to be accessed from the current directory.
6. Compile the MIRACL modules listed in the generated file *miracl.lst* and create a library file, typically *miracl.a* or *miracl.lib*. This might be achieved by editing *miracl.lst* into a suitable batch or make file. On UNIX it might be as simple as:-

```
gcc -I. -c -O2 mr*.c
ar rc miracl.a mr*.o
```

7. If using the C++ MIRACL wrapper, compile the required modules, for example *zzn.cpp* and/or *big.cpp* etc.
8. Compile and link your application code to any C++ modules it requires and to the MIRACL library.

Remember that MIRACL is portable software. It may be ported to **any** computer which supports an ANSI C compiler.

Note that MIRACL is a C library, not C++. It should always be built as a C library otherwise you might get compiler errors. To include MIRACL routines in a C program, include the header *miracl.h* at the start of the program, after including the C standard header *stdio.h* . You may also call MIRACL routines directly from a C++ program by:-

```
extern "C"
{
     #include "miracl.h"
}
```

although in most cases it will be preferable to use the C++ wrapper classes described in Chapter 7.

2.1 Optimising

In the context of MIRACL this means speeding things up. A critical decision to be made when configuring MIRACL is to determine the optimal underlying type to use. Usually this will be the **`int`** type. In general try to define the maximum possible underlying type, as requested by *config*. If you have a 64-bit processor, you should be able to specify a 64-bit underlying type. In some circumstances it may be faster to use a floating-point **`double`** underlying type.

Obviously an all-C build of MIRACL will be slowest (but still pretty fast!). It is also the easiest to start with. This requires an integer data type twice the width of the underlying type. In this context note that these days most compilers support a `long long` integer type which is twice the width of the `int`. Sometimes it is called `__int64` instead of `long long`.

If your processor is of the extreme RISC variety and supports no integer multiplication/division instruction, or if using a very large modulus, then the Karatsuba-Montgomery-Comba technique for fast modular multiplication may well be faster for exponentiation cryptosystems. Again the *config* program will guide you through this.

It is sometimes faster to implement the *mrmuldv* module in assembly language. This does not require the double-width data type. If you are lucky your compiler will also be supported by automatically invoked inline assembly, which will speed things up even further. See *miracl.h* to see which compilers are supported in this way.

For the ultimate speed, use the extreme techniques implemented in *mrkcm.c*, *mrcomba.c*. See *kcmcomba.txt* for instructions on how to automatically generate these files using the supplied **mex** utility. See also Chapter 5 for more details.

2.2 Upgrading from Version 3

Version 4.0 introduces the Miracl Instance Pointer, or *mip*. Previous versions used a number of global and static variables to store internal status information. There are two problems with this. Firstly such globals have to be given obscure names to avoid clashes with other project globals. Secondly it makes multi-threaded applications much more difficult to develop. So from Version 4.0 all such variables, now referred to as instance variables, are members of a structure of type *miracl*, and must be accessed via a pointer to an instance of this structure. This global pointer is now the only static/global variable maintained by the MIRACL library. Its value is returned by the **mirsys** routine, which initialises the MIRACL library.

C++ programmers should note the change in the name of the instance class from `miracl` to `Miracl`. The *mip* can be found by taking the address of this instance.

```
Miracl precision=50;

.
mip=&precision;
.
etc
```

2.3 Multi-Threaded Programming

From version 4.4 MIRACL offers full support for Multi-threaded programming. This comes in various flavours.

The problem to be overcome is that MIRACL has to have access to a lot of instance specific status information via its *mip*. Ideally there should be no global variables, but MIRACL has this one pointer. Unfortunately every thread that uses MIRACL needs to have its own *mip*, pointing to its own independent status. This is a well-known issue that arises with threads.

The first solution is to modify MIRACL so that the *mip*, instead of being a global, is passed as a parameter to every MIRACL function. The MIRACL routines can be automatically modified to support this by defining **MR_GENERIC_MT** in *mirdef.h*. Now (almost all) MIRACL routines are changed such that the *mip* is the first parameter to each function. Some simple functions are exceptions and do not require the *mip* parameter – these are marked with an asterix in Chapter 9. For an example of a program modified to work with a MIRACL library built in this way, see the program *brent_mt.c*. Note however that this solution does NOT apply to programs written using the MIRACL C++ wrapper described in Chapter 7. It only applies to C programs that access the MIRACL routines directly.

An alternative solution is to use *Keys*, which are a type of thread specific "global" variable. These *Keys* are not part of the C/C++ standard, but are operating system specific extensions, implemented via special function calls. MIRACL provides support for both Microsoft Windows and Unix operating systems. In the former case these *Keys* are called *Thread-Local Storage*. See [Richter] for more information. For Unix MIRACL supports the POSIX standard interface for multithreading. A very useful reference for both Windows and Unix is [Walmsley]. This support for threads is implemented in the module *mrcore.c*, at the start of the file and in the initialisation routine **mirsys**.

For Windows, define **MR_WINDOWS_MT** in *mirdef.h*, and for Unix define **MR_UNIX_MT**. In either case there are some programming implications.

In the first place the *Key* that is to maintain the *mip* must be initialised and ultimately destroyed by the programs primary thread. These functions are carried out by calls to the special routines **mr_init_threading** and **mr_end_threading** respectively.

In C++ programs these functions might be associated with the constructor and destructor of a global variable [Walmsley] – this will ensure that they are called at the appropriate time before new threads are forked off from the main thread. They must

be called before any thread calls **mirsys** either explicitly, or implicitly by creating a thread-specific instance of the class `Miracl`.

It is strongly recommended that program development be carried out without support for threads. Only when a program is fully tested and debugged should it be converted into a thread.

Threaded programming may require other OS-specific measures, in terms of linking to special libraries, or access to special heap routines. In this regard it is worth pointing out that all MIRACL heap accesses are via the module *mralloc.c*.

See the example program *threadwn.cpp* for an example of Windows C++ multithreading. Read the comments in this program – it can be compiled and run from a Windows Command prompt. Similarly see *threadux.cpp* for an example of Unix multi-threading.

2.4 Constrained environments

In version 5 of MIRACL there is new support for implementations in very small and constrained environments. Using the *config* utility it is now possible to allow various time/space trade-offs, but the main innovation is the possibility of building and using MIRACL in an environment which does not support a heap. Normally space for big variables is obtained from the heap, but by specifying in the configuration header **MR_STATIC**, a version of the library is built which will always attempt to allocate space not from the heap, but from static memory or from the stack.

The main downside to this is that the maximum size of big variables must be set at compile time, when the library is being created. As always it is best to let the *config* utility guide you through the process of creating a suitable *mirdef.h* configuration header.

For the C programmer, the allocation of memory from the stack for big variables proceeds as follows.

```
big x,y,z;
char mem[MR_BIG_RESERVE(3)];
memset(mem,0, MR_BIG_RESERVE(3));
```

This allocates space for 3 big variables on the stack, and set that memory to zero. Each individual big variable is then initialised as

```
x=mirvar_mem(mem,0);
y=mirvar_mem(mem,1);
z=mirvar_mem(mem,2);
```

Allocating all the space for multiple big variables from a single chunk of memory makes sense, as it leads to a faster initialization, and also gives complete control over variable alignment, which compilers sometimes get wrong. Note that in this mode the

usual big number initialization function *mirvar* is no longer available, and allocation must be implemented as described above.

Finally this memory chunk may optionally be cleared before leaving a function by a final call to *memset(.)* – this may be important for security reasons. For an example see the program *brent.c*.

This mechanism may be particularly useful when trying to implement a very small program using elliptic curves, which anyway require much smaller big numbers than other cryptographic techniques. To allocate memory from the stack for an elliptic curve point

```
epoint *x,*y,*z;
char mem[MR_ECP_RESERVE(3)];
memset(mem,0, MR_ECP_RESERVE(3));
```

To initialize these points

```
x=epoint_init_mem(mem,0);
y=epoint_init_mem(mem,1);
z=epoint_init_mem(mem,2);
```

Again it may be advisable to clear the memory associated with these points before exiting the function.

This mechanism is fully supported for C++ programs as well, where it works in conjunction with the stack allocation method described in chapter 7. See *pk-demo.cpp* for an example of use.

In some extreme cases it may be desired to use only the stack for all memory allocation. This allows maximum use and re-use of memory, and avoids any fragmentation of precious RAM. This can be achieved for C programs by defining **MR_GENERIC_MT** in *mirdef.h*. See above for more details on this option.

A typical mirdef.h header in this case might look like:-

```
/*
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   Copyright (c) 1988-2005 Shamus Software Ltd.
 */

#define MR_LITTLE_ENDIAN
#define MIRACL 32
#define mr_utype int
#define MR_IBITS 32
#define MR_LBITS 32
#define mr_unsign32 unsigned int
#define mr_dltype __int64
#define mr_unsign64 unsigned __int64
#define MR_STATIC 7
#define MR_ALWAYS_BINARY
#define MR_NOASM
#define MAXBASE ((mr_small)1<<(MIRACL-1))
#define MR_BITSINCHAR 8
#define MR_SHORT_OF_MEMORY
#define MR_GENERIC_MT
#define MR_STRIPPED_DOWN
```

For examples of programs which use this kind of header, see *ecsgen_s.c*, *ecsign_s.c* and *ecsver_s.c*, and *ecsgen2s.c*, *ecsign2s.c* and *ecsver2s.* These programs implement very small and fast ECDSA key generation, digital signature, and verification on a Pentium using Microsoft C++. See *ecdh2m.c* and *ecdhp.c* for more nice examples, which use precomputation to speed up EC Diffie-Hellman implementations.

For small 8 and 16-bit processors, see example programs *ecdhp8.c*, *ecdhp16.c*, *ecdh2m8.c* and *ecdh2m16.c*

NOTE: Doing without a heap is a little problematical. Structures can no longer be of variable size, and so various features of MIRACL become unavailable in this mode. For example precomputations such as required for application of the Chinese remainder theorem are no longer supported. However in a constrained environment it could be reasonably assumed that such precomputations are carried out off-line, and made available to the constrained program fixed in ROM.

3. The MIRACL modules are carefully designed so that an application will only pull in the minimal number of modules from the library for any given task. This helps to keep the program size down to a minimum. However if program size is a big issue then extra savings can sometimes be made by manually deleting from the modules functions that are not needed by your particular program (the linker will complain if the function is in fact needed).

The User Interface

AN EXAMPLE

```
/*
 *   Program to calculate factorials.
 */

#include <stdio.h>
#include "miracl.h"   /* include MIRACL system */

void main()
{ /* calculate factorial of number */
    big nf;          /* declare "big" variable nf */
    int n;
    miracl *mip=mirsys(5000,10);
                     /* base 10, 5000 digits per big  */
    nf=mirvar(1);   /* initialise big variable nf=1  */
    printf("factorial program\n");
    printf("input number n= \n");
    scanf("%d",&n);
    getchar();
    while (n>1)
       premult(nf,n--,nf); /* nf=n!=n*(n-1)*...2*1  */
    printf("n!= \n");
    otnum(nf,stdout); /* output result */
}
```

This program can be used to quickly calculate and print out 1000! (a 2568 digit number) in less a second on a 60MHz Intel Pentium-based computer, a task first performed ‘by H.S. Uhler using a desk calculator and much patience over a period of several years’ [Knuth73]. Many other example programs are described in Chapter 8.

Any program that wishes to make use of the MIRACL system must have an `#include "miracl.h"` statement. This tells the compiler to include the C header file *miracl.h* with the main program source file before proceeding with the compilation. This file contains declarations of all the MIRACL routines available to the user. The small sub-header file *mirdef.h* contains hardware/compiler-specific details.

In the main program the MIRACL system must be initialised by a call to the routine **mirsys**, which sets the number base and the maximum size of the *big* and *flash* variables. It also initialises the random number system, and creates several workspace *big* variables for its own internal use. The return value is the Miracl Instance Pointer, or *mip*. This pointer can be used to access various internal parameters associated with the current instance of MIRACL. For example to set the **ERCON** flag, one might write

```
mip->ERCON=TRUE;
```

The initial call to **mirsys** also initialises the error tracing system which is integrated with the MIRACL package. Whenever an error is detected the sequence of routine calls down to the routine which generated the error is reported, as well as the error itself. A typical error message might be

```
MIRACL error from routine powltr
     called from isprime
     called from your program
Raising integer to a negative power
```

Such an error report facilitates debugging, and assisted us during the development of these routines. An associated instance variable **TRACER**, initialised to OFF, if set by the user to ON, will cause a trace of the program's progress through the MIRACL routines to be output to the computer screen.

An instance flag **ERNUM,** initialised to zero, records the number of the last internal MIRACL error to have occurred. If the flag **ERCON** is set to FALSE (the default), an error message is directed to *stdout* and the program aborts via a call to the system routine *exit(0)*. If your system does not supply such a routine, the programmer must provide one instead. If **ERCON** is set to TRUE no error message is emitted and instead the onus is on the programmer to detect and handle the error. In this case execution continues. The programmer may choose to deal with the error, and reset **ERNUM** to zero. However errors are usually fatal, and if **ERNUM** is non-zero all MIRACL routines called subsequently will “fall-through” and exit immediately. See *miracl.h* for a list of all possible errors.

Every *big* or *flash* variable in the users program must be initialised by a call to the routine **mirvar**, which also allows the variable to be given an initial small integer value.

The full set of arithmetic and number-theoretic routines declared in *miracl.h* may be used on these variables. Full flexibility is (almost always) allowed in parameter usage with these routines. For example the call **multiply(*x*,*y*,*z*),** multiplies the *big* variable ***x*** by the *big* variable ***y*** to give the result as *big* variable ***z***. Equally valid would be **multiply(*x*,*y*,*x*), multiply(*y*,*y*,*x*)**, or **multiply(*x*,*x*,*x*)**. This last simply squares ***x***. Note that the first parameters are by convention always (usually) the inputs to the routines. Routines are provided not only to allow arithmetic on *big* and *flash* numbers, but also to allow these variables to perform arithmetic with the built-in integer and double precision data-types.

Conversion routines are provided to convert from one type to another. For details of each routine see the relevant documentation in Chapter 9 and the example programs of Chapter 8.

Input and output to a file or I/O device is handled by the routines **innum**, **otnum** **cinnum** and **cotnum**. The first two use the fixed number base specified by the user in the initial call of **mirsys**. The latter pair work in conjunction with the instance variable **IOBASE** which can be assigned dynamically by the user. A simple rule is that if the program is CPU bound, or involves changes of base, then set the base

initially to MAXBASE (or 0 if a full-width base is possible - see Chapter 4) and use **cinnum** and **cotnum**. If on the other hand the program is I/O bound, or needs access to individual digits of numbers (using **getdig**, **putdig** and **numdig**), use **innum** and **otnum**.

Input and output to/from a character string is also supported in a similar fashion by the routines **instr**, **otstr**, **cinstr** and **cotstr**. The input routines can be used to set *big* or *flash* numbers to large constant values. By outputting to a string, formatting can take place prior to actual output to a file or I/O device.

Numbers to bases up to 256 can be represented. Numbers up to base 60 use as many of the symbols 0-9, A-Z, a-x as necessary.

A number base of 64 enforces standard base64 encoding. On output base64 numbers are padded with trailing = symbols if needed, but not otherwise formatted. On input white-space characters are skipped, and padding ignored. Do not use base64 with *flash* numbers. Do not use base64 for outputting negative numbers, as the sign is ignored.

If the base is greater than 60 (and not 64), the symbols used are the ASCII codes 0-255.

A base of 256 is useful when it is necessary to interpret a line of text as a large integer, as is the case for the Public Key Cryptography programs described in Chapter 8. The routines **big_to_bytes** and **bytes_to_big** allow for direct conversion from the internal *big* format to/from pure binary.

Strings are normally zero-terminated. However a problem arises when using a base of 256. In this case every digit from 0 - 255 can legitimately occur in a number. So a 0 does not necessarily indicate the end of the string. On input another method must be used to indicate the number of digits in the string.

By setting the instance variable **INPLEN** = 25 (for example), just prior to a call to **innum** or **instr**, input is terminated after 25 bytes are entered. **INPLEN** is initialised to 0, and reset to 0 by the relevant routine before it returns.

For example, initialise MIRACL to use *big*s of 400 bytes

```
miracl *mip=mirsys(400,256);
```

Internal calculations are very efficient using this base.

Input an ASCII string as a base 256 number. This will be zero-terminated, so no need for **INPLEN.**

```
innum(x,stdin);
```

Now it is required to input exactly 1024 random bits

```
mip->INPLEN=128;
```

```
innum(y,stdin);
```

But we want to see output in HEX

```
mip->IOBASE=16;
cotnum(w,stdout);
```

Now in base64

```
mip->IOBASE=64;
cotnum(w,stdout);
```

4. Rational numbers may be input using either a radix point (e.g 0.3333) or as a fraction (e.g. 1/3). Either form can be used on output by setting the instance variable RPOINT=ON or =OFF.

Internal Representation

Conventional computer arithmetic facilities as provided by most computer language compilers usually provide one or two floating-point data types (e.g. single and double precision) to represent all the real numbers, together with one or more integer types to represent whole numbers. These built-in data-types are closely related to the underlying computer architecture, which is sensibly designed to work quickly with large amounts of small numbers, rather than slowly with small amounts of large numbers (given a fixed memory allocation). Floating-point allows a relatively small binary number (e.g. 32 bits) to represent real numbers to an adequate precision (e.g. 7 decimal places) over a large dynamic range. Integer types allow small whole numbers to be represented directly by their binary equivalent, or in 2's complement form if negative. Nevertheless this conventional approach to computer arithmetic has several disadvantages.

- Floating-point and Integer data-types are incompatible. Note that the set of integers, although infinite, is a subset of the rationals (i.e. fractions), which is in turn a subset of the reals. Thus every integer has an equivalent floating-point representation. Unfortunately these two representations will in general be different. For example a small positive whole number will be represented by its binary equivalent as an integer, and as separated mantissa and exponent as a floating-point. This implies the need for conversion routines, to convert from one form to the other.

- Most rational numbers cannot be expressed exactly (e.g. 1/3). Indeed the floating-point system can only express exactly those rationals whose denominators are multiples of the factors of the underlying radix. For example our familiar decimal system can only represent exactly those rational numbers whose denominators are multiples of 2 and 5; 1/20 is 0.05 exactly, 1/21 is 0.0476190476190.....

- Rounding in floating-point is base-dependant and a source of obscure errors.

- The fact that the size of integer and floating-point data types are dictated by the computer architecture, defeats the efforts of language designers to keep their languages truly portable.

- Numbers can only be represented to a fixed machine-dependent precision. In many applications this can be a crippling disadvantage, for example in the new and growing field of Public-Key cryptography.

- Base-dependent phenomena cannot easily be studied. For example it would be difficult to access a particular digit of a decimal number, as represented by a traditional integer data-type.

Herein is described a set of standard C routines which manipulate multi-precision rational numbers directly, with multi-precision integers as a compatible subset. Approximate real arithmetic can also be performed.

The two new data-types are called *big* and *flash*. The former is used to store multi-precision integers, and the latter stores multi-precision fractions as numerator and denominator in 'floating-slash' form. Both take the form of a fixed length array of digits, with sign and length information encoded in a separate 32-bit integer. The data type defined as **mr_small** used to store the number digits will be one of the built in types, for example `int`, `long` or even `double`. This is referred to as the "underlying type".

Both new types can be introduced into the syntax of the C language by the C statements

```
struct bigtype
{
     mr_unsign32 L;
     mr_small *d;
};

typedef struct bigtype *big;
typedef struct bigtype *flash;
```

Now *big* and *flash* variables can be declared just like any built-in data type, e.g.

```
     big x,y[10],z[10][10];
```

Observe that a *big* is just a pointer. The memory needed for each *big* or *flash* number instance is taken from the heap (or from the stack). Therefore each *big* or *flash* number must be initialised before use, and the required memory assigned to it.

Note that the user of these data-types is not concerned with this internal representation; the library routines allow *big* and *flash* numbers to be manipulated directly.

The structure of *big* and *flash* numbers is illustrated in figure (4.1).

These structures combine ease of use with representational efficiency. A denominator of length zero (*d=0*), implies an actual denominator of one; and similarly a numerator of length zero (*n=0*) implies a numerator of one. Zero itself is uniquely defined as the number whose first element is zero (i.e. *n=d=0*).

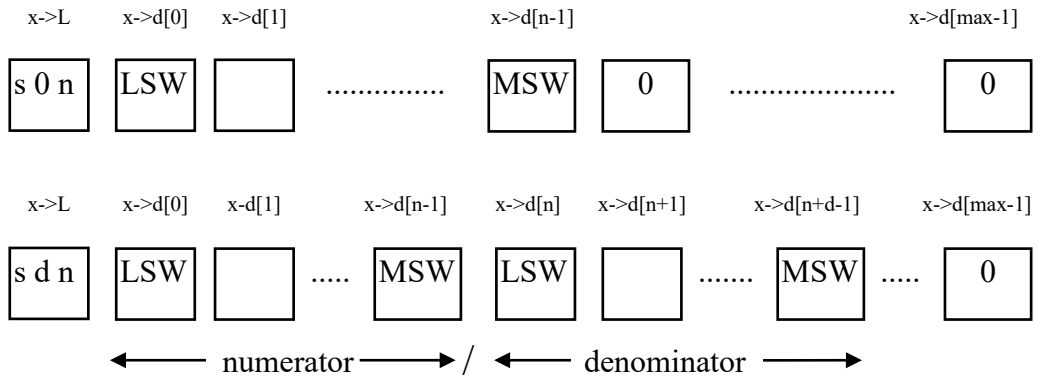


Figure 4.1: Structure of *big* and *flash* data-types where s is the sign of the number, *n* and *d* are the lengths of the numerator and denominator respectively, and LSW and MSW mean 'Least significant word and 'Most significant word' respectively

Note that the slash in the *flash* data-type is not in a fixed position, and may 'float' depending on the relative size of numerator and denominator.

A *flash* number is manipulated by splitting it up into separate *big* numerator and denominator components. A *big* number is manipulated by extracting and operating on each of its component integer elements. To avoid possible overflow, the numbers in each element are normally limited to a somewhat smaller range than that of the full word-length, e.g. 0 to 32767 (= 2^{15} - 1) on a 16-bit computer. However with careful programming a full-width base of 2^{16} can also be used, as the C language does not report a run-time error on integer overflow [Scott89b].

When the system is initialised the user specifies the fixed number of words (or bytes) to be assigned to all *big* or *flash* variables, and the number base to be used. Any base can be used, up to a maximum which is dependant on the wordlength of the computer used. If requested to use a small base *b*, the system will, for optimal efficiency, actually use base b^n, where *n* is the largest integer such that b^n fits in a single computer word. Programs will in general execute fastest if a full-width base is used (achieved by specifying a base of 0 in the initial call to **mirsys**). Note that this mode may be supported by extensive in-line assembly language for certain popular compiler/processor combinations, in certain time-critical routines, for example if using Borland/Turbo C with an 80x86 processor. Examine, for example, the source code in module *mrarth1.c*.

The encoding of the sign and numerator and denominator size information into a single word is possible, as the C language has standard constructs for bit manipulation.

5. Implementation

No great originality is claimed for the routines used to implement arithmetic on the *big* data-type. The algorithms used are faithful renditions of those described by Knuth [Knuth81]. However some effort was made to optimise the implementation for speed. At the heart of the time-consuming multiply and divide routines there is, typically, a need to multiply together a digit from each operand, add in a 'carry' from a previous operation, and then separate the total into a digit of the result, and a `carry' for the next operation. To illustrate consider this base 10 multiplication:

$$\begin{array}{r} \mathbf{8723536221} \\ \mathbf{\times\ 9} \\ \hline \mathbf{78511825989} \end{array}$$

To correctly process the column with the 5 in it, we multiply $5 \times 9 = 45$, add in the 'carry' from the previous column (a 3), to give 48, keep the 8 as the result for this column, and carry the 4 to the next column.

This basic primitive operation is essentially the calculation of the quotient *(a.b+c)/m* and its remainder. For the example above *a*=5, *b*=9, *c*=3 and *m*=10. This operation has surprisingly universal application, and since it lies at the innermost loop of the arithmetic algorithms, its efficient implementation is essential.

There are three main difficulties with a high-level language general base implementation of this MAD (Multiply, Add and Divide) operation.

- It will be slow.

- Quotient and remainder are not available simultaneously as a result of the divide operation. Therefore the calculation must be essentially done twice, once to get the quotient, and once for the remainder.

- Although the operation results in two single digit quantities, the intermediate product *(a.b+c)* may be double-length. Indeed such a Multiply-Add and Divide routine can be used on all occasions when a double-length quantity would be required by the basic arithmetic algorithms. Note that the C language is blessed with a 'long' integer data-type which may in fact be capable of temporarily storing this product.

For these reasons it is best to implement this critical operation in the assembly language of the computer used, although a portable C version is possible. At machine-code level a transitory double-length result can often be dealt with, even if the C long data-type is not itself double-length (as is the case for most C compilers as implemented on 32-bit computers, for which *ints* and *longs* are both 32-bit quantities). For further details see the documentation in the file *mrmuldv.any*.

A criticism of the MIRACL system might be its use of fixed length arrays for its *big* and *flash* data types. This was done to avoid the difficult and time-consuming problems of memory allocation and garbage collection, which would be needed by a variable-length representation. However it does mean that when doing a calculation on *big* integers that the results of all intermediate calculations must be less than or equal to the fixed size initially specified to **mirsys**.

In practise most numbers in a stable integer calculation are of more or less the same size, except when two are multiplied together in which case a double-length intermediate product is created. This is usually immediately reduced again by a subsequent divide operation. A classic example of this would be in the Pollard-Brent factoring program (Chapter 8).

Note that this is another manifestation, on a macro level, of the problem mentioned above. It would be a pity to have to specify each variable to be twice as large as necessary, just to cope with these occasional intermediate products. For this reason a special Multiply, Add and Divide routine **mad** has been included in the MIRACL library. It has proved very useful when implementing large programs (like the Pomerance-Silverman-Montgomery factoring program, Chapter 8) on computers with limited memory.

As well as the basic arithmetic operations, routines are also provided:

1. to generate and test *big* prime numbers, using a probabilistic primality test [Knuth81]
2. to generate *big* and *flash* random numbers, based on the subtract-with-borrow generator [Marsaglia]. Note however that the basic random number generator implemented internally is **not** cryptographicly secure. In a real cryptographic application it would not be adequate. A Cryptographicly strong generator is provided in the module *mrstrong.c*
3. to calculate powers and roots
4. to implement both the normal and extended Euclidean GCD (Greatest Common Divisor) algorithm [Knuth81]
5. to implement the 'Chinese Remainder Theorem' [Knuth81], and to calculate the Jacobi Symbol [Reisel].
6. to multiply extremely large numbers, using the Fast Fourier Transform method [Pollard71].

When performing extensive modular arithmetic, a time-critical operation is that of 'Modular Multiplication', that is multiplication of two numbers followed by reduction to the remainder when divided by a fixed *n*, the modulus. One obvious solution would be to use the **mad** routine described above. However Montgomery [Monty85] has proposed an alternative method. This requires that numbers are first converted to a special *n-residue* form. However once in this form modular multiplication is somewhat faster, using a special routine that requires no division whatsoever. When the calculation is complete, the answers can be converted back to normal form. Note that modular addition and subtraction of *n-residues* proceeds as usual, using the same routines as used for normal arithmetic. Given the requirement for conversion of variables to/from *n-residue* format, Montgomery's method should only be considered when a calculation requires an extensive amount of modular arithmetic using the same modulus. It is in fact much more convenient to use in a C++ environment, which hides these difficult details. See Chapter 7.

Montgomery arithmetic is used internally by many of the MIRACL library routines that require extensive modular arithmetic, such as the highly optimised modular

exponentiation function **powmod**, and those functions which implement GF(*p*) Elliptic Curve arithmetic. Details can be found in Chapter 9.2.

For the fastest possible modular arithmetic, one must alas resort to assembly language, and to methods optimised for a particular modulus, or moduli of a particular size. A number of different techniques are supported and can be used. The first two methods, the Comba and KCM methods, are implemented in the files *mrcomba.c* and *mrkcm.c* respectively. These files are created from template files *mrcomba.tpl* and *mrkcm.tpl* by inserting macros defined in a *.mcs* file. This is done automatically using the supplied **m**acro **ex**pansion utility **mex**. Compile and run *config.c* on your target system to automatically create a suitable *mirdef.h* and for advise on how to proceed. Also read *kcmcomba.txt*. To get the fastest possible performance for your embedded application it is recommended that you should develop your own *x.mcs* file, if one is not already provided for your processor/compiler.

Two other rather more experimental techniques are implemented in the files *mr87v.c* and *mr87f.c* for the Intel 80x86 family of processors only, using the Borland C++ compiler.

If conditions are right the appropriate code will be automatically invoked by calling for example **powmod**.

It is important to note that the four techniques described require a compiler that supports in-line assembly. Furthermore the latter two techniques have only been tested with the Borland C++ V4.5 compiler for the 80x86 family of processors.

The first idea is to completely unravel and reorganise the program loops implicit in the multiplication and reduction process, as first advocated by [Comba] and modified by [Scott96]. See *mrcomba.tpl*. A fixed length modulus must be used and specified at compile time by defining **MR_COMBA** to the modulus size (in words) in *mirdef.h*. This works well for small to medium size moduli, particularly as used in GF(*p*) elliptic curve cryptography. For even more speed, the modular reduction algorithm can be optimised for a modulus that has a particularly simple form. This can be done by manually inserting the appropriate code into *mrcomba.tpl*. Example code for the case of a modulus $p = 2^{192}-2^{64}-1$ is given there in the routine **comba_redc**. To invoke this special code **MR_SPECIAL** must be defined in *mirdef.h.*

This technique can be combined with Karatsuba's idea for fast multiplication [Knuth81] to speed up modular multiplication for larger moduli [WeiDai]. This Karatsuba-Comba-Montgomery (KCM) method is invoked by defining **MR_KCM** in *mirdef.h*. The modulus size in computer words is restricted to be equal to **MR_KCM*2^{n}** for any positive ***n*** (within reason). This is a consequence of using Karatsuba's algorithm. For example defining **MR_KCM** to be 8 on a 32-bit computer allows popular modulus sizes of 512, 1024, 2048 .... bits.

Another alternative is to exploit the floating point co-processor (if there is one), as its multiplication instruction is often faster than that of the integer unit [Rubin]. This is the case for the original Intel Pentium processor whose embedded co-processor takes only 3 cycles to perform a multiplication, compared with the 10 required for an integer multiply, although this is not true of the Pentium Pro, II, or III. Also the co-

processor has eight extra registers, and can manipulate 64-bit numbers directly. These features allow the programmer some extra flexibility, which can be used to advantage. Some experimental code has been written in the modules *mr87f.c* and *mr87v.c*, which may be exploited by defining **MR_PENTIUM** in *mirdef.h*. Use *config.c* to generate *mirdef.h* – this time the underlying type must be chosen as **`double`**. The module *mr87v.c* implements compact looping code, which will work with any modulus less than a certain maximum. The module *mr87f.c* unrolls the loops for more speed, but is bulkier and requires a fixed size modulus. Note that these modes of operation are incompatible with a full-width base, and work best with a number base of (usually) 2^{28} or 2^{29} – *config.c* will work it out for you. Note also that although this method will speed modular exponentiation on a Pentium, it may actually be slower for most other 80x86 processors, so use with care. In one test a 2048 bit number was raised to a 2048-bit power, *mod* a 2048 bit modulus. This took 2.4 seconds on a 60MHz Pentium.

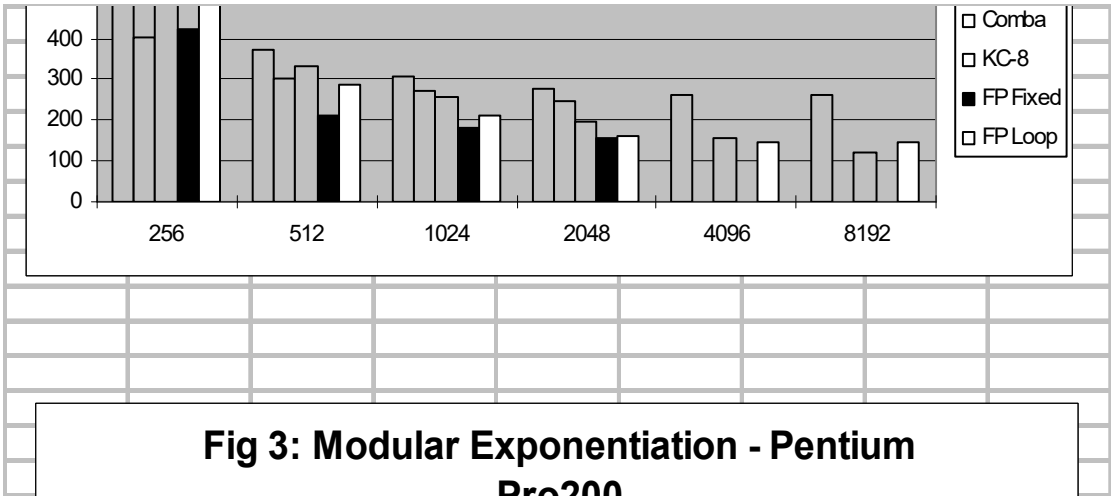


Fig 3: Modular Exponentiation - Pentium Pro200

This diagram illustrates the relative timings required by each method on a Pentium Pro 200MHz processor when compiled with the Borland C 32 bit compiler. The base line "Classic" method refers to the assembly language code implemented directly in *mrarth2.c* and *mrmonty.c*. The Comba and KCM implementations use assembly language from the *ms86.mcs* file. The modulus sizes are on the ***x*** axis, and the scaled time in seconds on the ***y*** axis. Note that in the calculation of $x^y \bmod n$ it is assumed that ***x***, ***y*** and ***n*** are randomly generated, all of same length in bits, and of no special form. It is assumed for example that the Comb optimisation technique (See [HAC] and *brick.c*) does not apply (that is ***x*** is a variable). The times shown are correct for the 8192 bit modulus. Times for smaller moduli are cumulatively scaled up by 8. So the times shown for a 4096 bit modulus should be divided by 8, for a 2048 bit modulus divided by 64, etc. Completely unrolled code is impracticable for the larger moduli, and hence timings for these methods are not given.

Note that the Comba method is optimal for moduli of 512 bits and less. This implies that it will be the optimal technique for fast GF(*p*) elliptic curve implementations, and for 1024-bit RSA decryption (which requires two 512-bit exponentiations and an application of the Chinese Remainder theorem). However these conclusions are processor-dependent, and may not be globally true. Also the Comba method can

generate a lot of code, and this may be an important consideration in some applications. In some circumstances (for example when the instruction cache is very small), it may in fact be advisable to take the working unrolled assembly language and carefully, manually, re-roll it.

From Version 5.20 of MIRACL, a new data type is supported directly in C. This is called a *zzn2* type, and basically it consists of two *bigs* in *n-residue* format

```
typedef struct
{
   big a;
   big b;
} zzn2;
```

where *a* and *b* can be considered as the real and imaginary parts respectively. The value of a *zzn2* is $a+ib$, where i is the imaginary square root of a quadratic non-residue. A *zzn2* variable is a representation of an element of a quadratic extension field with respect to a prime modulus p. For example if $p=3 \bmod 4$, then i can be taken as $\sqrt{-1}$, and the analogy to complex numbers with their real and imaginary parts becomes clear. They are particularly useful in implementations of cryptographic pairings. For an example of use, see the example program *cardona.cpp* which solves a cubic equation. A default value for the quadratic non-residue (which depends on the modulus) is stored in the instance variable *qnr*. Only the values -1 and -2 are currently supported.

6. To assist programmers generating code for a processor in a non-standard environment (e.g. an embedded controller), the code for dynamic memory allocation is always invoked from the module *mralloc.c*. By default this calls the standard C run-time functions calloc and free. However it can easily be modified to use an alternative user-defined memory allocation mechanism. For the same reason all screen/keyboard output and input is via the standard run-time functions fputc and fgetc. By intercepting calls to these functions, I/O can be redirected to non-standard devices.

Floating-Slash numbers

The straightforward way to represent rational numbers is as reduced fractions, as a numerator and denominator with all common factors cancelled out. These numbers can then be added, subtracted, multiplied and divided in the obvious way and the result reduced by dividing both numerator and denominator by their Greatest Common Divisor. An efficient GCD subroutine, using Lehmers modification of the classical Euclidean algorithm for multiprecision numbers [Knuth81], is included in the MIRACL package.

An alternative way to represent rationals would be as a finite continued fraction [Knuth81]. Every rational number p/q can be written as

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ldots}}}$$

or more elegantly as $p/q = [a_0/a_1/a_2/..../a_n]$ where the a_i are positive integers, usually quite small.

For example

$$\frac{277}{642} = [0/2/3/6/1/3/3]$$

Note that the a_i elements of the above continued fraction representation are easily found as the quotients generated as a by-product when the Euclidean GCD algorithm is applied to p and q.

As we are committed to fixed length representation of rationals, a problem arises when the result of some operation exceeds this fixed length. There is a necessity for some scheme of truncation, or rounding. While there is no obvious way to truncate a large fraction, it is a simple matter to truncate the continued fraction representation. The resulting, smaller, fraction is called a best rational approximation, or a convergent, to the original fraction.

Consider truncating 277/642 = [0/2/3/6/1/3/3]. Simply drop the last element from the CF representation, giving [0/2/3/6/1/3] = 85/197, which is a very close approximation to 277/642 (error = 0.0018%). Chopping more terms from the CF expansion gives the successive convergents as 22/51, 19/44, 3/7, 1/2, 0/1. As the fractions get smaller, the error increases. Obviously the truncation rule for a computer implementation should be to choose the biggest convergent that fits the computer representation.

The type of rounding described above is also called ‘Mediant rounding’. If p/q and r/s are two neighbouring representable slash numbers astride a gap, then their mediant is

the unrepresentable *(p+r)/(q+s)*. All larger fractions between *p/q* and the mediant will round to *p/q*, and those between *r/s* and the mediant will round to *r/s*. The mediant itself rounds to the 'simpler' of *p/q* and *r/s*.

This is theoretically a very good way to round, much better than the rather arbitrary and base-dependent methods used in floating-point arithmetic, and is the method used here. The full theoretical basis of floating-slash arithmetic is described in detail by Matula & Kornerup [Matula85]. It should be noted that our *flash* representation is in fact a cross between the fixed- and floating-slash systems analysed by Matula & Kornerup, as our slash can only float between words, and not between bits. However the characteristics of the *flash* data-type will tend to those of floating-slash, as the precision is increased.

The MIRACL routine **mround** implements mediant rounding. If the result of an arithmetic operation is the fraction *p/q*, then the Euclidean GCD algorithm is applied as before to *p* and *q*. However this time the objective is not to use the algorithm to calculate the GCD per se, but to use its quotients to build successive convergents to *p/q*. This process is stopped when the next convergent is too large to fit the *flash* representation. The complete algorithm is given below (Kornerup & Matula [Korn83])

Given $p \geq 0$ and $q \geq 1$

$$b_{-2}=p \qquad x_{-2}=0 \qquad y_{-2}=1$$
$$b_{-1}=q \qquad x_{-1}=1 \qquad y_{-1}=0$$

Now for $i=0,1,.....$ and for $b_{i-1}>0,$ find the quotient a_i and remainder b_i when b_{i-2} is divided by $b_{i-1},$ such that

$$b_i = -a_i.b_{i-1} + b_{i-2}$$

Then calculate

$$x_i = a_i.x_{i-1} + x_{i-2}$$
$$y_i = a_i.y_{i-1} + y_{i-2}$$

Stop when $\frac{x_i}{y_i}$ is to big to fit the *flash* representation, and take $\frac{x_{i-1}}{y_{i-1}}$ as the rounded result.

If applied to 277/642, this process will give the same sequence of convergents as stated earlier.

Since this rounding procedure must be applied to the result of each arithmetic operation, and since it is potentially rather slow, a lot of effort has been made to optimise its implementation. Lehmer's idea of operating only with the most significant piece of each number for as long as possible [Knuth81] is used, so that for most of the iterations only single-precision arithmetic is needed. Special care is taken to avoid the rounded result overshooting the limits of the *flash* representation [Scott89a]. The

application of the basic arithmetic routines to the calculation of elementary functions such as *log(x)*, *exp(x)*, *sin(x)*, *cos(x)*, *tan(x)* etc., uses the fast algorithms described by Brent [Brent76].

In many cases the result given by a program can be guaranteed to be exact. This can be checked by testing the instance variable **EXACT**, which is initialised to TRUE and is only set to FALSE if any rounding takes place.

A disadvantage of using a *flash* type of variable to approximate real arithmetic is the non-uniformity in gap-size between representable values (Matula & Kornerup [Matula85]).

To illustrate this consider a floating-slash system which is constrained to have the product of numerator and denominator less than 256. Observe that the first representable fraction less than 1/1 in such a system is 15/16, a gap of 1/16. The next fraction larger than 0/1 is 1/255, a gap of 1/255. In general, for a *k*-bit floating-slash system, the gap size varies from smaller than 2^{-k} to a worst case $2^{-k/2}$. In practise this means that a real value that falls into one of the larger gaps, will be represented by a fraction which will be accurate to only half its usual precision. Fortunately such large gaps are rare, and increasingly so for higher precision, occurring only near simple fractions. However it does mean that real results can only be completely trusted to half the given decimal places. A partial solution to this problem would be to represent rationals directly as continued fractions. This gives a much better uniformity of gap-size (Kornerup & Matula [Korn85]), but would be very difficult to implement using a high level language.

Arithmetic on *flash* data-types is undoubtedly slower than on an equivalent sized multiprecision floating-point type (e.g. [Brent78]). The advantages of the *flash* approach are its ability to exactly represent rational numbers, and do exact arithmetic on them. Even when rounding is needed, the result often works out correctly, due to the tendency of mediant-rounding to prefer a simple fraction over a complex one. For example the *roots* program (Chapter 8) when asked to find the square root of 2 and then square the result, comes back with the exact answer of 2, despite much internal rounding.

WARNING! Do **NOT** mix *flash* arithmetic with the built-in *double* arithmetic. They don't mix well. If you decide to use *flash* arithmetic, use it throughout, and convert all constants at the start to type *flash*. Even better specify such constants if possible as fractions. So (in C++) it is much preferable to write

```
x=Flash(5,8);  // x=5/8
```

7. rather than `x=.625;`

The C++ Interface

Many users of the MIRACL package would be disappointed that they have to calculate

$$t = x^2 + x + 1$$

for a flash variable *x* by the sequence

```
fmul(x,x,t);
fadd(t,x,t);
fincr(t,1,1,t);
```

rather than by simply

```
t=x*x+x+1;
```

Someone could of course use the MIRACL library to write a special purpose C compiler which could properly interpret such an instruction (see Cherry and Morris [Cherry] for an example of this approach). However such a drastic step is not necessary. A superset of C, called C++ has gained general acceptance as the natural successor to C. The enhancements to C are mainly aimed at making it an object-oriented language. By defining *big* and *flash* variables as 'classes' (in C++ terminology), it is possible to 'overload' the usual mathematical operators, so that the compiler will automatically substitute calls to the appropriate MIRACL routines when these operators are used in conjunction with *big* or *flash* variables. Furthermore C++ is able to look after the initialisation (and ultimate elimination) of these data-types automatically, using its constructor/destructor mechanism, which is included with the class definition. This relieves the programmer from the tedium of explicitly initialising each *big* and *flash* variable by repeated calls to **mirvar** . Indeed once the classes are properly defined and set up, it is as simple to work with the new data-types as with the built-in *double* and *int* types. Using C++ also helps shield the user from the internal workings of MIRACL.

The MIRACL library is interfaced to C++ via the header files *big.h*, *flash.h, zzn.h*, *gf2m.h, ecn.h* and *ec2.h*. Function implementation is in the associated files *big.cpp*, *flash.cpp*, *zzn.cpp, gf2m.cpp, ecn.cpp* and *ec2.cpp*, which must be linked into any application that requires them. The Chinese Remainder Theorem is also elegantly implemented as a class, in files *crt.h* and *crt.cpp*. See *decode.cpp* for an example of use. The Comb method for fast modular exponentiation with precomputation [HAC] is implemented in *brick.h*. See *brick.cpp* for an example of use. The GF(*p*) elliptic curve equivalents are in *ebrick.h* and *ebrick.cpp* and the GF(2^m) elliptic curve equivalents in *ebrick2.h* and *ebrick2.cpp* respectively._

EXAMPLE

```
/*
 *   Program to calculate factorials.
 */

#include <iostream>
#include "big.h"   /* include MIRACL system */

using namespace std;

Miracl precision(500,10); // This makes sure that MIRACL
                          // is initialised before main()
                          // is called
void main()
{ /* calculate factorial of number */
    Big nf=1;      /* declare "Big" variable nf */
    int n;
    cout << "factorial program\n";
    cout << "input number n= \n";
    cin >> n;
    while (n>1)
       nf*=(n--);  /* nf=n!=n*(n-1)*(n-2)*....3*2*1  */
    cout << "n!= \n" << nf << "\n";
}
```

Compare this with the C version of Chapter 3. Note the neat use of a dummy class *Miracl* used to set the precision of the *big* variables. Its declaration at global scope ensures that MIRACL is initialised before *main()* is called. (Note that this would not be appropriate in a multi-threaded environment.) When compiling and linking this program, don't forget to link in the *Big* class implementation file *big.cpp.*

Conversion to/from internal *Big* format is quite important:-

To convert a hex character string to a *Big*

```
Big x;
char c[100];
...
mip->IOBASE=16;
x=c;
```

To convert a *Big* to a hex character string

```
mip->IOBASE=16;
c << x;
```

To convert to/from pure binary, use the **from_binary**( ) and **to_binary**( ) friend functions.

```
int len;
char c[100];
...
Big x=from_binary(len,c);
        // creates Big x from len bytes of binary in c

len=to_binary(x,100,c,FALSE);
        // converts Big x to len bytes binary in c[100]
len=to_binary(x,100,c,TRUE);
        // converts Big x to len bytes binary in c[100]
        // (right justified with leading zeros)
```

In many of the example programs, particularly the factoring programs, all the arithmetic is done *mod n*. To avoid the tedious reduction *mod n* required after each operation, a new C++ class *ZZn* has been used, and defined in the file *zzn.h*. This class *ZZn* (for *ZZ(n)* or the ring of integers *mod n*) has its arithmetic operators defined to automatically perform the reduction. The function **modulo(n)** sets the modulus. In an analogous fashion the C++ class *GF2m* deals with elements of the field defined over GF(2^m). In this case the "modulus" is set via **modulo(m,a,b,c)**, which also specifies either a trinomial basis $t^m + t^a + 1$, (and set $b=c=0$), or a pentanomial basis $t^m + t^a + t^b + t^c + 1$. See the IEEE P1363 documentation for details.
http://grouper.ieee.org/groups/1363/draft.html

Internally the *ZZn* class uses Montgomery representation. See *zzn.h*. Note that the internal implementation of *ZZn* is hidden from the application programmer, a classic feature of C++. Thus the awkward internals of Montgomery representation need not concern the C++ programmer.

The class *ECn* defined in *ecn.h* makes manipulation of points on GF (*p*) elliptic curves a simple matter, again hiding all the grizzly details. The class *EC2* defined in *ec2.h* does the same for GF(2^m) elliptic curves.

Almost all of MIRACL's functionality is accessible from C++. Programming can often be done intuitively, without reference to this manual, using familiar C syntax as illustrated above. Other functions are accessed using the 'obvious' syntax - as in for example `x=gcd(x,y);`, or `y=sin(x);`. For more details examine the header files and example programs.

C++ versions of most of the example programs are included in the distribution media, with the file extensions *.cpp*

One problem with manipulating large objects in C++ is the tendency of the compiler to generate code to create/destroy/copy multiple temporary objects. By default MIRACL obtains memory for *Big* and *Flash* variables from the heap. This can be quite time-consuming, and all such objects need ultimately to be destroyed. It would be faster to assign memory instead from the stack, especially for relative small big numbers. This can now be achieved by defining **BIGS=m** at compilation time. For example if using the Microsoft C++ compiler from the command line:-

```
C:miracl>cl /O2 /GX /DBIGS=50 brent.cpp big.cpp zzn.cpp miracl.lib
```

Note that the value of **m** should be the same as or less than the value of **n** that is specified in the call to `mirsys(n,0);` or in `Miracl precision=n;` in the main program.

When using finite-field arithmetic, valid numbers are always less than a certain fixed modulus. For example in the finite field mod *n*, the class defined in *zzn.h* and *zzn.cpp* might handle numbers with respect to a 512-bit modulus *n*, which is set by **modulo(n)**. In this case one can define **ZZNS=16** so that all elements are of a size 16x32=512, and are created on the stack. (This works particularly well in combination with the Comba mechanism described in Chapter 5.)

In a similar fashion, when working over the field GF(2^{283}) , one can define **GF2MS=9**, so that all elements in the field are stored in a fixed memory allocation of 9 words taken from the stack.

In these latter two cases the precision **n** specified in the call to `mirsys(n,0);` or in `Miracl precision=n;` in the main program should be at least 2 greater than the **m** that specified in the **ZZNS=m** or **GF2MS=m** definition.

This is not recommended for program development, or if the objects are very large. It is only relevant with C++ programs. See the comments in the sample programs *ibe_dec.cpp* and *dl.cpp* for examples of the use of this mechanism. However the benefits can often be substantial – programs may be up to twice as fast.

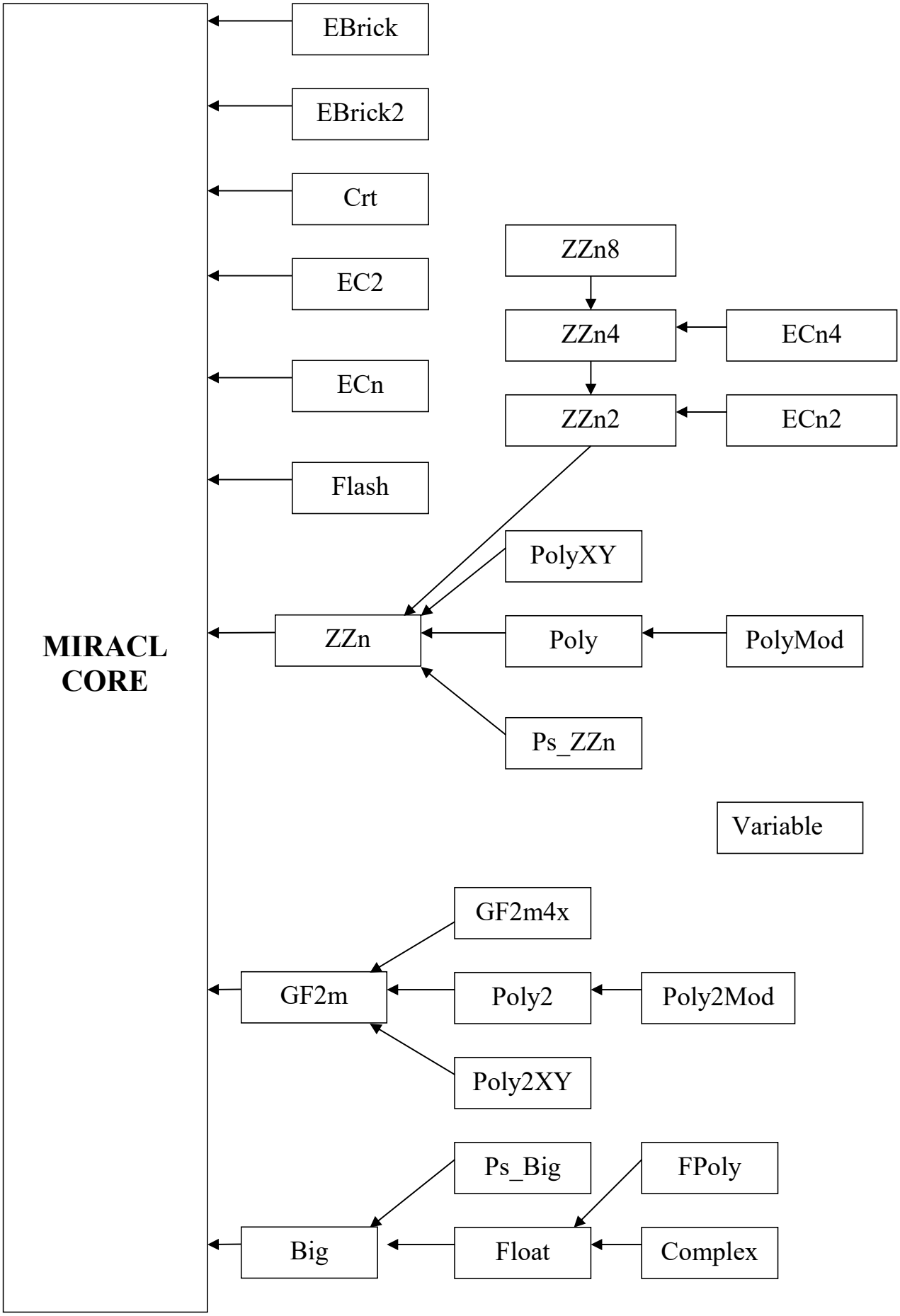
Finally here is a more elaborate C++ program to implement a relatively complex cryptographic protocol. Note the convention of using capitalised variables for field elements.

```
/*
 *   Gunthers's ID based key exchange - Finite field version
 *   See RFC 1824
 *   r^r variant (with Perfect Forward Security)
 */

#include <iostream>
#include <fstream>
#include "zzn.h"

using namespace std;

Miracl precision=100;

char *IDa="Identity 1";
char *IDb="Identity 2";

// Hash function
Big H(char *ID)
{ // hash character string to 160-bit big number
   int b;
   Big h;
   char s[20];
   sha sh;
```

```
    shs_init(&sh);
    while (*ID!=0) shs_process(&sh,*ID++);
    shs_hash(&sh,s);
    h=from_binary(20,s);
    return h;
}

int main()
{
    int bits;
    ifstream common("common.dss");    // construct file stream
    Big p,q,g,x,k,ra,rb,sa,sb,ta,tb,wa,wb;
    ZZn G,Y,Ra,Rb,Ua,Ub,Va,Vb,Key;
    ZZn A[4];
    Big b[4];
    long seed;
    miracl *mip=&precision;
    cout << "Enter 9 digit random number seed  = ";
    cin >> seed; irand(seed);

// get common data. Its in hex. G^q mod p = 1
    common >> bits;
    mip->IOBASE=16;
    common >> p >> q >> g;
    mip->IOBASE=10;
    modulo(p);    // set modulus

    G=(ZZn)g;

    cout << "Setting up Certification Authority ... " << endl;

// CA generates its secret and public keys

    x=rand(q);        // CA secret key, 0 < x < q
    Y=pow(G,x);       // CA public key, Y=G^x

    cout << "Visiting CA ...." << endl;

// Visit to CA - a
    k=rand(q);
    Ra=pow(G,k);
    ra=(Big)Ra%q;
    sa=(H(IDa)+(k*ra)%q);
    sa=(sa*inverse(x,q))%q;

// Visit to CA - b
    k=rand(q);
    Rb=pow(G,k);
    rb=(Big)Rb%q;
    sb=(H(IDb)+(k*rb)%q);
    sb=(sb*inverse(x,q))%q;

    cout << "Offline calculations .... " << endl;

// offline calculation - a
    wa=rand(q);
    Va=pow(G,wa);
    ta=rand(q);
    Ua=pow(Y,ta);

// offline calculation - b
```

```
    wb=rand(q);
    Vb=pow(G,wb);
    tb=rand(q);
    Ub=pow(Y,tb);

// Swap ID, R, U, V
    cout << "Calculate Key ... " << endl;

// calculate key - a
// Key = Vb^wa.Ub^sa.G^[(H(IDa)*tb)%q].Rb^[(rb*ta)%q]   mod p

    rb=(Big)Rb%q;
    A[0]=Vb; A[1]=Ub; A[2]=G; A[3]=Rb;
    b[0]=wa; b[1]=sa; b[2]=(H(IDb)*ta)%q; b[3]=(rb*ta)%q;

    Key=pow(4,A,b);    // extended exponentiation
    cout << "Key= \n" << Key << endl;

// calculate key - b
    ra=(Big)Ra%q;
    A[0]=Va; A[1]=Ua; A[2]=G; A[3]=Ra;
    b[0]=wb; b[1]=sb; b[2]=(H(IDa)*tb)%q; b[3]=(ra*tb)%q;

    Key=pow(4,A,b);    // extended exponentiation
    cout << "Key= \n" << Key << endl;
    return 0;
}
```

MIRACL has evolved quite a complex class hierarchy – see the diagram below. Where possible classes are built directly on top of the C/assembly core. Note the support for polynomials, power series and extension fields.


MIRACL
CORE
EBrick
EBrick2
Crt
EC2
ECn
Flash
ZZn8
ZZn4
ECn4
ZZn2
ECn2
PolyXY
ZZn
Poly
PolyMod
Ps_ZZn
Variable
GF2m4x
GF2m
Poly2
Poly2Mod
Poly2XY
Ps_Big
FPoly
Big
Float
Complex

8. Example Programs

Note: The programs described here are of an experimental nature, and in many cases are not completely 'finished off'. For further information read the comments associated with the appropriate source file.

8.1 Simple Programs

8.1.1 hail.c

This program allows you to investigate so-called hailstone numbers, as described by Gruenberger [Gruen]. The procedure is simple. Starting with any number apply the following rules:

(a) If it is odd, multiply it by 3 and add 1.

(b) If it is even, divide it by 2.

(c) Repeat the process, until the number becomes equal to 1, in which case stop.

It would appear that for any initial number this process always eventually terminates, although it has not been proved that this must happen, or that the process cannot get stuck in an infinite loop. What goes up, it seems, must come down. Try the program for an initial value of 27. Then try it using much bigger numbers, like 10709980568908647 (which has interesting behaviour).

8.1.2 palin.c

This programs allows one to investigate palindromic reversals [Gruen]. A palindromic number is one which reads the same in both directions. Start with any number and apply the following rules.

(a) Add the number to the number obtained by reversing the order of the digits. Make this the new number.

(b) Stop the process when the new number is palindromic.

It appears that for most initial numbers this process quickly terminates. Try it for 89. Then try it for 196.

8.1.3 mersenne.c

This program attempts to generate all prime numbers of the form 2^n-1. The largest known primes have always been of this form because of the efficiency of this Lucas-Lehmer test. The routine **fft_mult** is used, as it is faster for very large numbers.

8.2 Factoring Programs

Six different Integer Factorisation programs are included, covering all modern approaches to this classical problem. For more background and information on the algorithms used, see [Scott89c].

8.2.1 brute.c

This program attempts to factorise a number by brute force division, using a table of small prime numbers. When attempting a difficult factorisation it makes sense to try this approach first. Factorise 12345678901234567890 using this program. Then try it on bigger random numbers.

8.2.2 brent.c

This program attempts to factorise a number using the Brent-Pollard method. This method is faster at finding larger factors than the simple-minded brute force approach. However it will not always succeed, even for simple factorisations. Use it to factorise R17, that is 11111111111111111 (seventeen ones). Then try it on larger numbers that would not yield to the brute force approach.

8.2.3 pollard.c

Another factoring program, which implements Pollard's *(p-1)* method, specialises in quickly finding a factor *p* of a number *N* for which *(p-1)* has itself only small factors. Phase 1 of this method will work if all these small factors are less than LIMIT1. If Phase 1 fails then Phase 2 searches for just one final larger factor less than LIMIT2. The constants LIMIT1 and LIMIT2 are set inside the program.

8.2.4 williams.c

This program is similar to Pollards method, but can find a factor *p* of *N* for which *(p+1)* has only small factors. Again two phases are used. In fact this method is sometimes a *(p+1)* method, and sometimes a *(p-1)* method, so several attempts are made to hit on the *(p+1)* condition. The algorithm is rather more complex than that used in Pollards method, and is somewhat slower.

8.2.5 lenstra.c

Lenstra [Monty87] has discovered a new method of factorisation, generically similar to the Pollard and Williams methods, but potentially much more powerful. It works by randomly generating an Elliptic Curve, which can then be used to find a factor *p* of *N*, for which $p+1-\delta$ has only small factors, where δ depends on the particular curve chosen. If one curve fails then another can be tried, an option not possible with the Pollard/Williams methods. Again this is a two phase method, and although it has very good asymptotic behaviour, it is much slower than the Pollard/Williams methods for each iteration.

8.2.6 qsieve.c

This is a sophisticated Pomerance-Silverman-Montgomery [Pomerance], [Silverman] factoring program. which factors $F7 = 2^{128}+1$

340282366920938463463374607431768211457

in less than 30 seconds, running on a 60MHz Pentium-based computer. When this number was first factored, it took 90 minutes on an IBM 360 mainframe (Morrison & Brillhart [Morrison]), albeit using a somewhat inferior algorithm.

Its speciality is factoring all numbers (up to about sixty digits long), irrespective of the size of the factors. If the number to be factored is *N*, then the program actually works with a number *k.N*, where *k* is a small Knuth-Schroepel multiplier. The program itself works out the best value of *k* to use. Internally, the program uses a 'factor base' of small primes. The larger the number, the bigger will be this factor base. The program works by accumulating information from a number of simpler factorisations. As it progresses with these it prints out *working...n*. When it thinks it has enough information it prints out *trying*, but these tries may be premature and may not succeed. The program will always terminate before the number *n* in *working...n* reaches the size of the factor base.

This program uses much more memory than any of the other example programs, particularly when factoring bigger numbers. The amount of memory that the program can take is limited by the values defined for MEM, MLF and SSIZE at the beginning of the program. These limit the number of primes in the factor base, the number of 'larger' primes used by the so-called large-prime variation of the algorithm, and the sieve size respectively. They should be increased if possible, or reduced if your computer has insufficient memory. See [Silverman] for more details.

Use **qsieve** to factor 10000000000000000000000000000000009 (thirty-five digits).

8.2.7 factor.c

This program combines the above algorithms into a single general purpose program for factoring integers. Each method is used in turn in the attempt to extract factors. The number to be factored is given in the command line, as in **factor 11111111111**. The number can alternatively be specified as a formula, using the switch '-f', as in **factor -f (10#11-1)/9**. The symbol # here means 'to the power of' (# is used instead of ^ as the latter symbol has a special meaning for DOS on an IBM PC). Type **factor** on its own for a full description of this and other switches that can be used to control the input/output of this program.

8.3 Discrete Logarithm Programs

Two programs implement Pollards algorithms [Pollard78] for extracting discrete logarithms. The discrete logarithm problem is to find *x* given *y*, *r* and *n* in

$$y = r^x \bmod n$$

The above is a good example of a one-way function. It is easy to calculate *y* given *x*, but apparently extremely difficult to find *x* given *y*. Pollard's algorithms however perform quite well under certain circumstances, if *x* is known to be small or if *n* is a prime *p* for which *p-1* has only small factors.

8.3.1 kangaroo.c

This program finds *x* in the above, assuming that *x* is quite small. The value of *r* is fixed (at 16), and the modulus *n* is also fixed inside the program. Initially a 'trap' is set. Subsequently the discrete logarithm can be found (almost certainly) for any number, assuming its discrete logarithm is less than a certain upper limit. The number of steps required will be approximately the square root of this limit.

8.3.2 genprime.c

A prime number *p* with known factorisation of *p-1* is generated by this program, for use by the *index.c* and *identity.c* programs described below. The factors of *p-1* are output to a file *prime.dat*.

8.3.3 index.c

This program implements Pollard's rho algorithm for extracting discrete logarithms, when the modulus *n* in the above equation is a prime *p*, and when *p-1* has only relatively small factors. The number of steps required is a function of the square root of the largest of these factors.

8.4 Public-Key Cryptography

Public Key Cryptography is a two key cryptographic system with the very desirable feature that the encoding key can be made publicly available, without weakening the strength of the cipher. The first example program demonstrates many popular public-key techniques. Then two functional Public-Key cryptography systems, whose strength appears to depend on the difficulty of factorisation, are presented. The first is the classic RSA system (Rivest, Shamir & Adleman [RSA]). This is fast to encode a message, but painfully slow at decoding. A much faster technique has been invented by Blum and Goldwasser. This probabilistic Public Key system is also stronger than RSA in some senses. For more details see [Brassard], who describes it as 'the best that academia has had to offer thus far'. For both methods the keys are constructed from 'strong' primes to enhance security. Closely associated with PK Cryptography, is the concept of the Digital Signature. A group of example programs implement the Digital Signature Standard, using classic finite fields and elliptic curves over both the fields GF(p) and GF(2^m).

8.4.1 pk-demo.c

This program carries out a 1024-bit Diffie-Hellman key exchange, and then another Diffie-Hellman type key exchange, but this time based on a 160-bit prime and an elliptic curve. Next a test string is encrypted and decrypted using the El Gamal method. The program finishes with a 1024 bit RSA encryption/decryption of the same string. For a good description of all these techniques see [Stinson]. Anyone attempting to implement a PK system using MIRACL is strongly encouraged to examine this file, and its C++ counter-part **pk-demo.cpp**

8.4.2 bmark.c/imratio.c

The benchmarking program *bmark.c* allows the user to quickly determine the time that will be required to implement any of the popular public key methods. It can be compiled and linked with any of the variants of the MIRACL library, as specified in *mirdef.h*, to determine which gives the best performance on a particular platform for a particular PK method. The program *imratio.c* when compiled and run calculates the significant ratios S/M, I/M and J/M, where S is the time for a modular squaring, M the time for a modular multiplication, I the time for a modular inversion, and J the time for a Jacobi symbol calculation.

8.4.3 genkey.c

This program generates the ‘public’ encoding key and ‘private’ decoding keys that are necessary for both the original Rivest-Shamir-Adleman PK system and the superior Blum-Goldwasser method [Brassard]. These keys can take a long time to generate, as they are formed from very large prime numbers, which must be generated carefully for maximum security.

The size of each prime in bits is set inside the program by a `#define`. The security of the system depends on the difficulty of factoring the encoding ‘public’ key, which is formed from two such large primes. The largest numbers which can be routinely factored using hundreds of powerful computers are 430 bits long (1996). So a minimum size of 512 bits for each prime gives plenty of security (for now!)

After this program has run, the two keys are created in files PUBLIC.KEY and PRIVATE.KEY.

8.4.4 encode.c

Messages or files may be encoded with this program, which uses the ‘public’ encoding key from the file PUBLIC.KEY, generated by the program *genkey*, which must have been run prior to using this program. When run, the user is prompted for a file to encipher. Either supply the name of a text file, or press return to enter a message directly from the keyboard. In the former case the encoded output is sent to a file with the same name, but with the extension .RSA. In the latter case a prompt is issued for an output filename, which must be given. Text entered from the keyboard must be terminated by a CONTROL-Z (end-of-file character). Type out the encoded file and be impressed by how indecipherable it looks.

8.4.5 decode.c

Messages or files encoded using the RSA system may be decoded using this program, which uses the ‘private’ decoding key from the file PRIVATE.KEY generated by the program *genkey* which must have been run at some stage prior to using this program.

When run, the user is prompted for the name of the file to be decoded. Type in the filename (without an extension - the program will assume the extension .RSA) and press return. Then the user is asked for an output filename. Either supply a filename or press return, in which case the decoded output will be sent straight to the screen. A problem with the RSA system becomes immediately apparent - decoding takes quite a relatively long time! This is particularly true for larger key sizes and long messages.

8.4.6 enciph.c

This program works in an identical fashion to the program ‘encode’, except that it prompts for a random seed before encrypting the data. This random seed is then used internally to generate a larger random number. The encryption process depends on this random number, which means that the same data will not necessarily produce the same cipher-text, which is one of the strengths of this approach. As well as creating a file with a .BLG extension containing the encrypted data, a second small file (with the .KEY extension) is also produced.

8.4.7 deciph.c

This program works in an identical fashion to the program ‘decode’. However it has the advantage that it runs much more quickly. There will be a significant initial delay while a rather complex calculation is carried out. This uses the private key and the data in the .KEY file to recover the large random number used in the encryption process. Thereafter deciphering is as fast as encipherment.

8.4.8 dssetup.c

A standard method for digital signature has been proposed by the American National Institute of Standards and Technology (NIST), and fully described in the Digital Signature Standard [DSS]. This program generates a prime q, another much larger prime $p=2nq+1$, (where n is random) and a generator g. This information is made common to all. This program generates the common information $\{p,q,g\}$ into a file *common.dss*

8.4.9 limlee.c

It has been shown by Lim & Lee [LimLee] that for certain Discrete Logarithm based protocols (but not for the Digital Signature Standard) there is a weakness associated with primes of the kind generated by the *dssetup.c* program described above. To avoid these problems they recommend that p is of the form $p=2.p_1.p_2.p_3...q + 1$, where the p_i are primes greater than q. This program generates the values (p,q,g) into a file *common.dss*, and can be used in place of *dssetup.c*. It is a little slower.

8.4.10 dssgen.c

Each individual user who wishes to digitally sign a computer file randomly generates their own private key $x<q$ and makes available a public key $y=g^x \ mod \ p.$ The security of the system depends on the sizes of p and q (at least 512 bits and 160 bits respectively). This program generates a single public/private key pair in the files *public.dss* and *private.dss* respectively.

8.4.11 dssign.c

This program uses the private key from *private.dss* to 'sign' a document stored in a file. First the file data is 'hashed' down to a 160 bit number using SHA, the Standard Hash Algorithm. This is also specified by the NIST and is implemented in the provided module *mrshs.c*. The 160-bit hash is duly 'signed' as described in [DSS], and the signature, in the form of two 160-bit numbers, written out to a file. This file has the same name as the document file, but with the extension *.dss*.

8.4.12 dssver.c

This program uses the public key from *public.dss* to verify the signature associated with a file, as described in [DSS].

8.4.13 ecsgen.c, ecsign.c, ecsver.c

The Digital Signature technique can also be implemented using Elliptic Curves over the field GF(p)[Jurisic]. Common domain information in the order $\{p,A,B,q,X,Y\}$ is extracted from the file *common.ecs* created using one of the point-counting algorithms described below. These values specify an initial point (X,Y) on an elliptic curve $y^2=x^3+Ax+B$ mod p which has q points on it. The advantages are a much smaller public key for the same level of security. Smaller numbers can be used as the discrete logarithm problem is apparently much more difficult in the context of an elliptic curve. This in turn implies that elliptic curve arithmetic is also potentially faster. However the use of smaller numbers is somewhat offset by the more complex calculations involved.

This set of programs has the same functionality as those described above for the standard DSS. Note however that the file extension *.ecs* is used for all the generated files. Read the comments in the source files for more information.

8.4.14 ecsgen2.c, ecsign2.c, ecsver2.cpp

These programs provide the same functionality as those provided above, but use elliptic curves defined over the field GF(2^m). Domain information in this case is extracted from the file *common2.ecs* in the order $\{m,A,B,q,X,Y,a,b,c\}$, where (X,Y) specifies an initial point on the elliptic curve $y^2=x^3+Ax^2+B$ defined over GF(2^m). The parameters of a trinomial or pentanomial basis are also specified, t^m+t^a+1 or $t^m+t^a+t^b+t^c+1$ respectively. In the former case b and c are zero. Finally $cf.q$ specifies the number of points on the curve, the product of a large prime factor q and a small cofactor cf. The latter is normally 2 or 4. The file *common2.ecs* can be created by the **schoof2** program described below.

8.4.15 cm.cpp, schoof.cpp, mueller.cpp, process.cpp, sea.cpp, schoof2.cpp

A problem with Elliptic curve cryptography is the construction of suitable curves. This is actually much more difficult than the equivalent problem in the integer finite field as implemented by the program *dssetup.c/dssetup.cpp*. One approach is the Complex Multiplication method, as described in the Annex to the IEEE P1363 Standard Specifications for Public Key Cryptography (available from the Web). This is implemented here by the C++ program *cm.cpp* and its supporting modules *float.cpp*, *complex.cpp, flpoly.cpp, poly.cpp*, and associated header files.

The program when run uses command line arguments. Type **cm** on its own to get instructions. For example

```
cm -f 2#224-2#96+1 -o common.ecs
```

generates the common information needed to implement elliptic curve cryptography into the file *common.ecs*.

As an alternative to the CM method, a random curve can be generated, and the points on the curve directly counted. This is more time-consuming than complex multiplication, but may lead to more secure, less structured curves. The basic algorithm is due to Schoof [Sch],[Blake] and is only practical due to the use of Fast Fourier Transform methods [Shoup] for the multiplication/division of large degree polynomials. See *mrfast.c*. Its still very slow, much slower than **cm**. Type **schoof** on its own to get instructions. For example

```
schoof -f 2#192-2#64-1 -3 35317045537
```

counts the points on the curve $y^2 = x^3 - 3x + 35317045537$ mod $2^{192}-2^{64}-1$.

This curve is randomly selected (actually 35317045537 is my international phone number). The answer is the prime number

```
6277101735386680763835789423127240467907482257771524603027
```

Be prepared to wait, or….
Use the suite of programs, **mueller**, **process**, and **sea**, which together implement the superior, but more complex, Schoof-Elkies-Atkin method for point counting. See [Blake] for details.

First of all the **mueller** program should be run, to generate the required Modular Polynomials. This needs to be done just once – ever. The greater your collection of Modular Polynomials, the greater the size of prime modulus that can be used for the elliptic curves of interest. Note that this program is particularly hard on memory resources, as well as taking a long time to run. However after an hour at most you should have enough Modular Polynomials to start experimenting. As with all these programs, simply typing the program name without parameters generates instructions

for use. Also be sure to read the comments at the start of the source file, in this case *mueller.cpp.*

Next run the **process** application, which processes the file of raw modular polynomials output by **mueller**, for use with a specified prime modulus.

Finally run **sea** to count the points on the curve, and optionally to create a *.ecs* file as described above.

For example:-

```
mueller 0 120 -o mueller.raw
process -f 65112*2#144-1 -i mueller.raw -o test160.pol
sea -3 49 -i test160.pol
```

generates all the modular polynomials for primes from 0 to 120, and outputs them to the file *mueller.raw*. Then these polynomials are processed with respect to the prime $p = 65112.2^{144}-1$, to create the file *test160.pol*. Finally the main **sea** application counts the points on the curve $y^2=x^3-3x+49 \bmod p$

This may be more complicated to use, but its much faster than **schoof**.

Read the comments at the start of *sea.cpp* for more information.

For elliptic curves over GF(2^m), the program **schoof2** can be used, which is quite similar to **schoof**. It is even slower, but just about usable on contemporary hardware. For example

```
schoof2 1 52 191 9 -o common2.ecs
```

counts the points on the curve $y^2+xy=x^3+x^2+52$, over the field GF(2^{191}). A suitable irreducible basis must also be specified, in this case $t^{191}+t^9+1$. Tables of suitable bases can be found in many documents, for example in Appendix A of the IEEE P1363 standard. See [Menezes] for a description of the method.

For more information on building these applications see the files *cm.txt, schoof.txt, schoof2.txt* and *sea.txt.*

8.4.16 crsetup.cpp, crgen.cpp, crencode.cpp, crdecode.cpp

Public key schemes should ideally be immune from *adaptive chosen cipher-text* attacks, whereby an attacker is able to obtain decryptions of any presented cipher-texts other than the particular one they are interested in. Recently Cramer & Shoup [CS] have come up with a Public Key encryption method that is provably immune to such powerful attacks. The program **crsetup** creates various global parameters, and **crgen** generates one set of public and private keys in the files *public.crs* and *private.crs* respectively. To encrypt an ASCII file called for example *fred.txt*, run the **crencode** program that generates a random session key, and uses it to encrypt the file. This session key is in turn encrypted by the public key and stored in the file *fred.key*.

The binary encrypted file itself is stored as *fred.crs*. To decrypt the file, run the **crdecode** program, which uses the private key to recover the session key, and hence decode the text to the screen.

A couple of points are worth highlighting. First of all the bulk encryption is carried out using a block cipher method. Such hybrid systems are standard practise, as block ciphers are much faster than public key methods. The block cipher scheme used is the new Advanced Encryption Standard block cipher, which is implemented in *mraes.c*.

Examination of the source code *crdecode.cpp* reveals that decryption is a two-pass process. On the first pass the program determines the validity of the cipher-text, and only after that is known to be valid does the program go on to decrypt the file. So the decryption procedure will not respond at all to arbitrary bit strings concocted by an attacker.

8.4.17 brick.c, ebrick.c, ebrick2.c

Certain Cryptographic protocols require the exponentiation of a fixed number *g*, that is the calculation of $g^x \bmod n$, where *g* and *n* are known in advance. In this case the calculation can be substantially speeded up by a precomputation which generates a small table of *big* numbers. The method was first described by Brickell et al [Brick]. The example program *brick.c* illustrates the method. The GF(*p*) elliptic curve equivalent is provided in *ebrick.c* and the GF(2^m) equivalent in *ebrick2.c*. In a typical application the precomputed tables might be generated using one of these programs (see commented-out code in *ebrick2.c*), which then might be transferred to ROM in an embedded program. The embedded program might use a static build of MIRACL to make use of these tables.

8.4.18 identity.c

This is a program that allows individuals, issued with certain secret information, to establish mutual keys by performing a calculation involving only the other correspondents publicly known identity. No interchange of data is required [Maurer], and so this is called Non-Interactive Key Exchange. Note that the 'publicly known identity' might, for example, be simply an email address. For a full description see [Scott92]. This example program generates the secret data from the proffered Identity. However before this program is run, the program *genprime.c* must be run twice, to generate a pair of suitable trap-door primes. Copy the output of the program, *prime.dat,* first to *trap1.dat* and then to *trap2.dat*. The product of these primes will be used as the composite modulus used for subsequent calculations.

8.4.19 Pairing based Cryptography

A number of experimental programs are provided to implement cryptographic protocols based on *pairings*. Notably there are examples of Identity-Based Encryption (IBE) and authenticated key exchange. Read the files *pairings.txt*, *ake.txt* and *ibe.txt* for details.

8.5 'flash' Programs

Several programs demonstrate the use of *flash* variables. One gives an implementation of Gaussian elimination to solve a set of linear equations, involving notoriously ill-conditioned Hilbert matrices. Others show how rational arithmetic can be used to approximate real arithmetic, in, for example the calculation of roots and π. The former program detected an error in the value for the square root of 5 given in Knuth's appendix A [Knuth81]. The correct value is

2.23606 79774 99789 69640 91736 68731 **2**7623 54406

The error is in the tenth last digit, which is a 2, and not a 1.

The *roots* program runs particularly fast when calculating the square roots of single precision integers, as a simple form of continued fraction generator can be used. In one test the golden ratio $(1+\sqrt{5})/2$ was calculated to 100,000 decimal places in 3 hours of CPU time on a VAX11/780.

The 'sample' program was used to calculate π correct to 1000 decimal places, taking less than a minute on a 25MHz 80386-based IBM PC to do so.

8.5.1 roots.c

This program calculates the square root of an input number, using Newtons method. Try using it to calculate the square root of two. The accuracy obtained depends on the size of the flash variables, specified in the initial call to **mirsys**. The tendency of flash arithmetic to prefer simple numbers can be illustrated by requesting, say, the square root of 7. The program calculates this value and then squares it, to give 7 again exactly. On your pocket calculator the same result will only be obtained if clever use is made of extra (hidden) guard digits.

8.5.2 hilbert.c

Traditionally the inversion of 'Hilbert' matrices is regarded as a tough test for any system of arithmetic. This programs solves the set of linear equations $H.x = b$, where H is a Hilbert matrix and b is the vector [1,1,1,1,....1], using the classical Gaussian Elimination method.

8.5.3 sample.c

This program is the same as that used by Brent [Brent78] to demonstrate some of the capabilities of his Fortran Multiprecision arithmetic package. It calculates π, $\exp(\pi.\sqrt{163/9})$, and $\exp(\pi.\sqrt{163})$.

8.5.4 ratcalc.c

As a comprehensive and useful demonstration of flash arithmetic this program simulates a standard full-function scientific calculator. Its unique feature (besides its 36-digit accuracy) is its ability to work directly with fractions, and to handle mixed calculations involving both fractions and decimals. By using this program the user will quickly get a feel for flash arithmetic and its capabilities. Note that this program contains some non-portable code (screen handling routines) that must be tailored to each individual computer/terminal combination. The version supplied works only on standard PCs using DOS, or a command prompt window in Windows ‘NT/’98.

9. The MIRACL routines

Note: In these routines a *big* parameter can also be used wherever a *flash* is specified, but not visa-versa. Further information may be gleaned from the (lightly) commented source code. An asterix * after the name indicates that the function does not take a *mip* parameter if **MR_GENERIC_MT** is defined in *mirdef.h*. See Section 2.3 for more details.

9.1 Low level routines

9.1.1 absol *

Function:
```
void absol(x,y)
flash x,y;
```

Module: mrcore.c

Description: Gives absolute value of a big or flash number.

Parameters: Two big/flash variables *x* and *y*. On exit $y=|x|$.

Return value: None

Restrictions: None

9.1.2 add

Function:
```
void add(x,y,z)
big x,y,z;
```

Module: mrarth0.c

Description: Adds two big numbers.

Parameters: Three big numbers *x*, *y* and *z*. On exit $z=x+y$.

Return value: None

Restrictions: None

Example:
```
add(x,x,x);  /* This doubles the value of x. */
```

9.1.3 brand

Function: `int` **`brand`**`()`

Module: mrcore.c

Description: Generates random integer number

Parameters: None

Return Value: A random integer number

Restrictions: First use must be preceded by an initial call to **irand**.
NOTE: This generator is not cryptographicly strong. For cryptographic applications, use the **strong_rng** routine.

9.1.4 bigbits

Function:
```
void bigbits(n,x)
int n;
big x;
```

Module: mrbits.c

Description: Generates a big random number of given length. Uses the built-in simple random number generator initialised by **irand.**

Parameters A big number *x* and an integers *n*. On exit *x* contains a big random number *n* bits long.

Return value: None

Restrictions: None

Example:
```
bigbits(100,x);
```
This generates a 100 bit random number

9.1.5 big_to_bytes

Function:
```
int big_to_bytes(max,x,ptr,justify)
int max;
big x;
char *ptr;
BOOL justify
```

Module: mrarth1.c

Description: Converts a positive big number *x* into a binary octet string

Parameters: A big number *x* and a byte array *ptr* of length *max*. 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.

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

Restrictions: *max* must be greater than 0 if *justify*=TRUE

9.1.6 bytes_to_big

Function: `void bytes_to_big(len,ptr,x)`
`int len;`
`char *ptr;`
`big x;`

Module: mrarth1.c

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

Parameters: A pointer to a byte array *ptr* of length *len*, and a big result *x*.

Return value: None

Restrictions: None

Example:

```
/*
 *  test program to exercise big_to_bytes() and bytes_to_big()
 */

#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");

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

   return 0;
}
```

9.1.7 cinnum

Function:
```
int cinnum(x,f)
flash x;
FILE *f;
```

Module: mrio2.c

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

Parameters: A big/flash number *x* and a file descriptor *f*. For input from the keyboard specify *f* as *stdin*, otherwise as the descriptor of some other opened file. To force input of a fixed number of bytes, set the instance variable INPLEN to the required number, just before calling **cinnum**.

Return value: The number of input characters.

Restrictions: None

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

9.1.8 cinstr

Function:
```
int cinstr(x,s)
flash x;
char *s;
```

Module; mrio2.c

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

Parameters: A big/flash number *x* and a string *s*.

Return value: The number of input characters.

Restrictions: None

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

9.1.9 compare *

Function:
```
int compare(x,y)
big x,y;
```

Module: mrcore.c

Description: Compares two big numbers.

Parameters: Two big numbers *x* and *y*.

Return value: Returns +1 if $x>y$, returns 0 if $x=y$, returns -1 if $x<y$.

9.1.10 Restrictions: None

convert

Function:
```
void convert(n,x)
int n;
big x;
```

Module: mrcore.c

Description: Convert an integer number to big number format.

Parameters: An integer *n* and a big number *x*.

Return value: None

Restrictions: None

9.1.11 copy *

Function:
```
void copy(x,y)
flash x,y;
```
Module: mrcore.c

Description: Copies a big or flash number to another.

Parameters: Two big or flash numbers *x* and *y*. On exit *y*=*x*. Note that if *x* and *y* are the same variable, no operation is performed.

Return value: None

9.1.12 Restrictions: None

cotnum

Function:
```
int cotnum(x,f)
flash x;
FILE *f;
```

Module: mrio2.c

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

Parameters: A big/flash number *x* and a file descriptor *f*. If *f* is *stdout* then output will be to the screen, otherwise to the file opened with descriptor *f*.

Return value: Number of output characters.

Restrictions: None

Example:
```
mip->IOBASE=16;
cotnum(x,fp);
```

9.1.13 This outputs *x* in hex, to the file associated with *fp*.

cotstr

Function: `int cotstr(x,s)`
`flash x;`
`char *s;`

Module: mrio2.c

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

Parameters A big/flash number *x* and a string *s*. On exit *s* will contain a representation of the number *x*.

Return value: Number of output characters.

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

9.1.14 decr

Function: `void decr(x,n,z)`
`big x,z;`
`int n;`

Module: mrarth0.c

Description: Decrement a big number by an integer amount.

Parameters: Big numbers *x* and *z*, and integer *n*.
On exit *z*=*x*-*n*.

Return value: None

Restrictions: None

9.1.15 divide

Function: `void` **`divide`**`(x,y,z)`
`big x,y,z;`

Module: mrarth2.c

Description: Divides one big number by another.

Parameters: Three big numbers *x*, *y* and *z*. On exit *z=x/y*; *x=x mod y*. The quotient only is returned if *x* and *z* are the same, the remainder only if *y* and *z* are the same.

Return value: None

Restrictions: Parameters *x* and *y* must be different, and *y* must be non-zero.

Example: `divide(x,y,y);`

This sets *x* equal to the remainder when *x* is divided by *y*. The quotient is not returned.

9.1.16 divisible

Function: `BOOL` **`divisible`**`(x,y)`
`big x,y;`

Module: mrarth2.c

Description: Tests a big number for divisibility by another

Parameters: Two big numbers *x* and *y*.

Return value: TRUE if *y* divides *x* exactly, otherwise FALSE

9.1.17 Restrictions: The parameter *y* must be non-zero.

ecp_memalloc

Function: `void *ecp_memalloc(n)`
`int n;`

Module: mrcore.c

Description: Reserves space for *n* elliptic curve points in one heap access. Individual points can subsequently be initialised from this memory by calling **epoint_init_mem**.

Parameters: The number *n* of elliptic curve points to reserve space for.

Return value: A pointer to the allocated memory.

Restrictions: None.

9.1.18 ecp_memkill

Function: `void ecp_memkill(mem,n)`
`char *mem;`
`int n;`

Module: mrcore.c

Description: Deletes and sets to zero the memory previously allocated by **ecp_memalloc**

Parameters: A pointer to the memory to be erased and deleted, and the size of that memory in elliptic curve points.

Return value: None

Restrictions: Must be preceded by a call to **ecp_memalloc**

9.1.19 exsign *

Function:
```
int exsign(x)
flash x;
```

Module: mrcore.c

Description: Extracts the sign of a big/flash number.

Parameters: A big/flash number *x*.

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

Restrictions: None

9.1.20 getdig

Function:
```
int getdig(x,i)
big x;
int i;
```

Module: mrcore.c

Description: Extracts a digit from a big number.

Parameters: A big number *x*, and the required digit *i*.

Return value: The value of the requested digit.

Restrictions: Returns rubbish if required digit does not exist.

9.1.21 get_mip

Function:
```
miracl *get_mip(void)
```

Module: mrcore.c

Description: Get the current Miracl Instance Pointer

Parameters: None

Return value: The *mip* - Miracl Instance Pointer – for the current thread.

Restrictions: This function does not exist if **MR_GENERIC_MT** is defined.

9.1.22 igcd *

Function:
```
int igcd(x,y)
int x,y;
```

Module: mrcore.c

Description: Calculates the Greatest Common Divisor of two integers using Euclids Method.

Parameters: Two integers *x* and *y*

Return value: The GCD of *x* and *y*

9.1.23 incr

Function:
```
void incr(x,n,z)
big x,z;
int n;
```

Module: mrarth0.c

Description: Increment a big variable.

Parameters: Big numbers *x* and *z*, and an integer *n*. On exit $z=x+n$.

Return value: None

Restrictions: None

Example:
```
incr(x,2,x);  /* This increments x by 2. */
```

9.1.24 init_big_from_rom

Function:
```
BOOL init_big_from_rom(big,int,const mr_small*,int,int*)
big x;
int len;
const mr_small *rom;
int romsize;
int *romptr;
```

Module: mrcore.c

Description: Initialises a big variable from ROM memory.

Parameters: A big number *x* and its length in computer words. The address of ROM memory which stores up to *romsize* computer words, and a pointer into the ROM. This pointer is incremented internally as ROM memory is accessed to fill *x*.

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

9.1.25 init_point_from_rom

Function:
```
BOOL init_point_from_rom(epoint *,int,const mr_small*,int,int*)
epoint *P;
int len;
const mr_small *rom;
int romsize;
int *romptr;
```

Module: mrcore.c

Description: Initialises an elliptic curve point from ROM memory.

Parameters: An elliptic curve point *P* and its length of its two big coordinates in computer words. The address of ROM memory which stores up to *romsize* computer words, and a pointer into the ROM. This pointer is incremented internally as ROM memory is accessed to fill *P*.

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

9.1.26 innum

Function:
```
int innum(x,f)
flash x;
FILE *f;
```

Module: mrio1.c

Description: Inputs a big or 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: A big/flash number *x* and a file descriptor *f*. For input from the keyboard specify *f* as *stdin*, otherwise as the descriptor of some other opened file.

Return value: The number of characters input.

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

Hint: 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**.

9.1.27 insign *

Function:
```
void insign(s,x)
int s;
flash x;
```

Module: mrcore.c

Description: Forces a big/flash number to a particular sign.

Parameters: A big/flash number *x*, and the sign *s* that it is to take. On exit $x=s.|x|$.

Return value: None

Restrictions: None

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

9.1.28 instr

Function:
```
int instr(x,s)
flash x;
char *s;
```

Module: mrio1.c

Description 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 A big/flash number *x* and a character string *s*.

Return value: The number of characters input.

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

9.1.29 irand

Function:
```
void irand(seed)
long seed;
```

Module: mrcore.c

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

Parameters: A long integer seed, which is used to start off the random number generator.

Return value: None

Restrictions: None

9.1.30 lgconv

Function:
```
void lgconv(ln,x)
long ln;
big x;
```

Module: mrcore.c

Description: Converts a long integer to big number format

Parameters: A long integer *ln* and a big number *x*

Return value: None

Restrictions: None

9.1.31 mad

Function: `void mad(x,y,z,w,q,r)`
`big x,y,z,w,q,r;`

Module: mrarth2.c

Description: Multiply add and divide big numbers. The initial product is stored in a double-length internal variable to avoid the possibility of overflow at this stage.

Parameters: Six big numbers *x,y,z,w,q* and *r*. On exit *q=(x.y+z)/w* and *r* contains the remainder. 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.

Return value: None

Restrictions: 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 */`

9.1.32 memalloc

Function: `void *memalloc(n)`
`int n;`

Module: mrcore.c

Description: Reserves space for *n* big variables in one heap access. Individual big/flash variables can subsequently be initialised from this memory by calling **mirvar_mem**.

Parameters: The number *n* of big/flash variables to reserve space for.

Return value: A pointer to the allocated memory.

Restrictions: None.

9.1.33 memkill

Function: `void memkill(mem,n)`
`char *mem;`
`int n;`

Module: mrcore.c

Description: Deletes and sets to zero the memory previously allocated by **memalloc**

Parameters: A pointer to the memory to be erased and deleted, and the size of that memory in bigs.

Return value: None

Restrictions: Must be preceded by a call to **memalloc**

9.1.34 mirexit

Function: `void mirexit()`

Module: mrcore.c

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

Parameters: None

Return value: None

9.1.35 Restrictions: Must be called after mirsys.

mirkill *

Function: `void` **`mirkill`**`(x)`
`big x;`

Module: mrcore.c

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

Parameters: A big/flash number *x*.

Return Value: None

9.1.36 mirsys

Function: `miracl *mirsys(nd,nb)`
`int nd,nb;`

Module: mrcore.c

Description: Initialise 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: The number of digits *nd* to use for each big/flash variable and the number base *nb*. If *nd* is negative it is taken as indicating the size of big/flash numbers in 8-bit bytes.

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

Restrictions: 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. (See *mirdef.h*)

Example: `miracl *mip=mirsys(500,10);`

9.1.37 This initialises the MIRACL system to use 500 decimal digits for each big or flash number.

mirvar

Function:
```
flash mirvar(iv)
int iv;
```

Module: mrcore.c

Description: 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: An integer initial value for the big/flash number.

Return value: A pointer to the reserved memory.

Restrictions: None

Example:
```
flash x;
x=mirvar(8);
```

Creates a flash variable *x=8*.

9.1.38 mirvar_mem

Function:
```
flash mirvar_mem(mem,index)
char *mem;
int index;
```

Module: mrcore.c

Description: 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: A pointer to the pre-allocated array *mem*, and an index into that array. Each index should be unique.

Return value: An initialised big/flash variable

Restrictions: Sufficient memory must have been allocated and pointed to by *mem*.

Example: See *brent.c* for an example of use.

9.1.39 multiply

Function: `void multiply(x,y,z)`
`big x,y,z;`

Module: mrarth2.c

Description: Multiplies two big numbers

Parameters: Three big numbers *x,y* and *z*. On exit *z=x.y*

Return value: None

Restrictions: None

9.1.40 negify *

Function: `void negify(x,y)`
`flash x,y;`

Module: mrcore.c

Description: Negates a big/flash number.

Parameters: Two big/flash numbers *x* and *y*. On exit *y*=-*x*.

Return value: None

Restrictions: None. Note that `negify(x,x)` is valid and sets *x*=-*x*

9.1.41 normalise

Function: `int normalise(x,y)`
`big x,y;`

Module: mrarth2.c

Description 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: Two big numbers *x* and *y*. On exit *y*=*n.x*.

Return value: Returns *n*, the normalising multiplier.

Restrictions: Use with care. Used internally.

9.1.42 numdig

Function:
```
int numdig(x)
big x;
```

Module: mrcore.c

Description: Determines the number of digits in a big number.

Parameters: A big number *x*.

Return value: The number of digits in *x*.

Restrictions: None

9.1.43 otnum

Function:
```
int otnum(x,f)
flash x;
FILE *f;
```

Module: mrio1.c

Description: Output a big or 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 RPOINT=ON. Otherwise it will be output as a fraction.

Parameters: A big/flash number *x* and a file descriptor *f*. If *f* is *stdout* then output will be to the screen, otherwise to the file opened with descriptor *f*.

Return value: Number of output characters.

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

9.1.44 otstr

Function:
```
int otstr(x,s)
flash x;
char *s;
```

Module: mrio1.c

Description Output 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 RPOINT=ON. Otherwise it will be output as a fraction.

Parameters A big/flash number *x* and a character string *s*. On exit *s* will contain a representation of *x*.

Return value: Number of output characters.

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

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

9.1.45 premult

Function:
```
void premult(x,n,z)
int n
big x,z;
```

Module: mrarth1.c

Description: Multiplies a big number by an integer

Parameters: Two big numbers *x* and *z*, and an integer *n*.
On exit *z=n.x*

Return value: None

Restrictions: None

9.1.46 putdig

Function:
```
void putdig(n,x,i)
big x;
int i,n;
```

Module: mrcore.c

Description: Set a digit of a big number to a given value

Parameters: A big number *x*, a digit number *i*, and its new value *n*.

Return value: None

Restrictions: The digit indicated must exist.

9.1.47 remain

Function:
```
int remain(x,n)
big x;
int n;
```

Module: mrarth1.c

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

Parameters: A big number *x*, and an integer *n*.

Return value: The integer remainder

9.1.48 set_io_buffer_size

Function:
```
void set_io_buffer_size(len)
int len;
```

Module: mrcore.c

Description: 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: The size of I/O buffer required.

Return value: None

Restrictions: Destroys the current contents of the I/O buffer

9.1.49 set_user_function

Function: `void set_user_function(func)`
`BOOL (*user)(void);`

Module: mrcore.c

Description: 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: A pointer to a user-supplied function, or NULL if not required.

Return value: None

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);
```

9.1.50 size *

Function: `int size(x)`
`big x;`

Module: mrcore.c

Description: 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: A big number *x*.

Return value: 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.**

Restrictions: None

9.1.51 subdiv

Function: `int subdiv(x,n,z)`
`int n;`
`big x,z;`

Module: mrarth1.c

Description: Divide a big number by an integer.

Parameters: Two big numbers *x* and *z*, and an integer *n*.
On exit *z=x/n*.

Return value: The integer remainder.

Restrictions: The value of *n* must not be zero.

9.1.52 subdivisible

Function:
```
BOOL subdivisible(x,n)
big x;
int n;
```

Module: mrarth1.c

Description: Tests a big number for divisibility by an integer.

Parameters: A big number *x* and an integer *n*.

Return value: TRUE is *n* divides *x* exactly, otherwise FALSE.

Restrictions: The value of *n* must not be zero.

9.1.53 subtract

Function:
```
void subtract(x,y,z)
big x,y,z;
```

Module: mrarth0.c

Description: Subtracts two big numbers.

Parameters: Three big numbers *x*, *y* and *z*. On exit *z=x-y*.

Return value: None

Restrictions: None

9.1.54 zero *

Function:
```
void zero(x)
flash x;
```

Module: mrcore.c

Description: Sets a big or flash number to zero

Parameters: A big or flash number *x*.

Return value: None

9.2 Advanced Arithmetic Routines

9.2.1 bigdig

Function:
```
void bigdig(n,b,x)
int n,b;
big x;
```

Module: mrrand.c

Description: Generates a big random number of given length. Uses the built-in simple random number generator initialised by **irand.**

Parameters: A big number *x* and two integers *n* and *b*. On exit *x* contains a big random number *n* digits long to base *b*.

Return value: None

Restrictions: The base *b* must be printable, that is $2 \le b \le 256$.

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

9.2.2 bigrand

Function:
```
void bigrand(w,x)
big w,x;
```

Module: mrrand.c

Description: Generates a big random number. Uses the built-in simple random number generator initialised by **irand.**

Parameters: Two big numbers *w* and *x*. On exit *x* is a big random number in the range $0 \le x < w$.

Return value: None

Restrictions: None

9.2.3 brick_init

Function:
```
BOOL brick_init(binst,g,n,w,nb)
brick *binst;
big g,n;
int w,nb;
```

Module: mrbrick.c

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

Parameters: A pointer to the current instance *binst*, the fixed generator *g*, the modulus *n*, the window size *w*, and the maximum number of bits to be used in the exponent *nb*.

Return value: TRUE if all went well, FALSE if there was a problem.

Restrictions: 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`.

9.2.4 brick_end *

Function:
```
void brick_end(binst)
brick *binst
```

Module: mrbrick.c

Description: Cleans up after an application of the Comb method.

Parameters: A pointer to the current instance

Return value: None

Restrictions: None

9.2.5 crt

Function:
```
void crt(pbc,rem,x)
big_chinese *pbc;
big *rem;
big x;
```

Module: mrcrt.c

Description Applies Chinese Remainder Theorem.

Parameters: A pointer *pbc* to the current instance. On exit *x* contains the big number which yields the given big remainders *rem[.]* when it is divided by the big moduli specified in a prior call to **crt_init**.

Return value: None

Restrictions: The routine **crt_init** must be called first.

9.2.6 crt_end *

Function:
```
void crt_end(pbc)
big_chinese *pbc;
```

Module: mrcrt.c

Description: Cleans up after an application of the Chinese Remainder Theorem.

Parameters: A pointer to the current instance of the Chinese Remainder Theorem.

Return value: None

9.2.7 Restrictions: None

crt_init

Function:
```
BOOL crt_init(pbc,np,m)
big_chinese *pbc;
int np;
big *m;
```

Module: mrcrt.c

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

Parameters: A pointer to the current instance *pbc*, the number of co-prime moduli *np*, and an array of at least two big moduli *m[.]*

Return value: TRUE if all went well, FALSE if there was a problem.

Restrictions: None

9.2.8 egcd

Function:
```
int egcd(x,y,z)
big x,y,z;
```

Module: mrgcd.c

Description: Calculates the Greatest Common Divisor of two big numbers.

Parameters: Three big numbers *x, y* and *z*.
On exit *z=gcd(x,y)*

Return value: GCD as integer, if possible, otherwise **MR_TOOBIG**

Restrictions: None

9.2.9 expb2

Function:
```
void expb2(n,x)
int n;
big x;
```

Module: mrbits.c

Description: Calculates 2 to the power of an integer as a big

Parameters: An integer *n*, and a big result *x*.
On exit $x=2^n$.

Return value: None

Restrictions: None

Example:
```
expb2(1398269,x);
decr(x,1,x);
mip->IOBASE=10;
cotnum(x,stdout);
```

This calculates and prints out the largest known prime number (on a true 32-bit computer with lots of memory!)

9.2.10 expint

Function:
```
void expint(b,n,x)
int b,n;
big x;
```

Module: mrarth3.c

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

Parameters: An integer *b*, an integer *n*, and a big result *x*.
On exit $x=b^n$.

Return value: None

Restrictions: None

9.2.11 fft_mult

Function: `void` **`fft_mult`**`(x,y,z)`
`big x,y,z;`

Module: mrfast.c

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

Parameters: Three big numbers *x, y* and *z*. On exit *z=x.y*

Return value: None

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

9.2.12 Example: See *mersenne.c*

gprime

Function: `void gprime(n)`
`int n;`

Module: mrprime.c

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

Parameters: A positive integer *n* indicating the maximum prime number to be generated. If *n=0* the PRIMES array is deleted.

Return value: None

9.2.13 hamming

Function: `int hamming(n)`
`big n;`

Module: mrarth1.c

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

Parameters: A big number *x*.

Return value: Hamming weight of *x*

9.2.14 invers *

Function: `unsigned int invers(x,y)`
`unsigned int x,y;`

Module: mrsmall.c

Description: Calculates the inverse of an integer modulus a co-prime integer

Parameters: An integer *x* and a co-prime integer *y*.

Return value: $x^{-1} \mod y$

Restrictions: Result unpredictable if *x* and *y* not co-prime

9.2.15 isprime

Function: `BOOL` **`isprime`**`(x)`
`big x;`

Module: mrprime.c

Description: 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 **NTRY** times, where **NTRY** is an instance variable with a default initialisation in routine **mirsys**.

NOTE: This routine first test divides *x* by the list of small primes stored in the instance array **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.

Parameters: A big number *x*.

Return value: Returns the boolean value TRUE if *x* is (almost certainly) prime, otherwise FALSE.

Restrictions: None

9.2.16 jac

Function: `int` **`jac`**`(x,n)`
`unsigned int x,n;`

Module: mrsmall.c

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

Parameters: Two unsigned numbers *x* and *n*

Return value: The value of *(x/n)* as +1 or -1, or 0 if symbol undefined

Restrictions: None

9.2.17 jack

Function:
```
int jack(x,n)
big x,n;
```

Module: mrjack.c

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

Parameters: Two big numbers *x* and *n*

Return value: The value of *(x/n)* as +1 or -1, or 0 if symbol undefined

Restrictions: None

9.2.18 logb2

Function: `int logb2(x)`
`big x;`

Module: mrbits.c

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

Parameters: A big number *x*.

Return value: Number of bits in *x*

Restrictions: None

9.2.19 lucas

Function: `void lucas(x,e,n,vp,v)`
`big x,e,n,vp,v`

Module: mrlucas.c

Description: 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: Five big numbers *x, e, n, vp* and *v*.
On exit $v=V_e(x) \bmod n$ and $vp=V_{e-1}(x) \bmod n$ where *n* is the current Montgomery modulus. Only *v* is returned if *v* and *vp* are not distinct.

Return value: None

Restrictions: The value of *n* must be odd.

Note: The "sister" Lucas function $U_e(x)$ can, if required, be calculated as

$$U_e(x) \equiv [x.V_e(x) - 2.V_{e-1}(x)]/(x^2 - 4) \bmod n$$

9.2.20 multi_inverse

Function:
```
BOOL multi_inverse(m,x,n,w)
int m;
big n;
big *x,*w;
```

Module: mrxgcd.c

Description: 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: The number of inverses required *m*, an array *x*[.] of *m* numbers whose inverses are wanted, the modulus *n*, and the resulting array of inverses *w*[.].

Return value: TRUE if successful, otherwise FALSE.

Restrictions: The parameters *x* and *w* must be distinct.

9.2.21 nres

Function:
```
void nres(x,y)
big x,y;
```

Module: mrmonty.c

Description: Converts a big number to *n-residue* form.

Parameters: Two big numbers *x* and *y*.
On exit *y* is the *n-residue* form of *x*.

Return value: None

Restrictions: Must be preceded by call to **prepare_monty**.

9.2.22 nres_dotprod

Function
```
void nres_dotprod(m,x,y,w)
int m;
big x[],y[],w;
```

Module: mrmonty.c

Description: 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: Two arrays *x* and *y* each of *m n-residues.*
On exit $w=\Sigma x_i y_i \mod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: Must be preceded by call to **prepare_monty**.

9.2.23 nres_double_modadd

Function:
```
void nres_double_modadd(x,y,w)
big x,y,w;
```

Module: mrmonty.c

Description: Adds two double length bigs modulo *p.R*, where *R* is 2^n 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: Three big numbers *x*, *y* and *z*. On exit $z=a+b \mod pR$

Return value: None

9.2.24 nres_double_modsub

Function: `void` **`nres_double_modsub`**`(x,y,w)`
`big x,y,w;`

Module: mrmonty.c

Description: Subtracts two double length bigs modulo $p.R$, where R is 2^n 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: Three big numbers x, y and z. On exit $z=a-b \bmod pR$

Return value: None

9.2.25 nres_lazy

Function: `void` **`nres_lazy`**`(a,b,c,d,x,y)`
`big a,b,c,d,x,y;`

Module: mrmonty.c

Description: 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: Six big numbers. On exit $(x+iy)=(a+ib)(c+id)$, where i is imaginary square root of the quadratic non-residue.

Return value: None

9.2.26 nres_lucas

Function: `void` **`nres_lucas`**`(x,e,vp,v)`
`big x,e,vp,v;`

Module: mrlucas.c

Description: Modular Lucas exponentiation of an *n-residue*

Parameters: An *n-residue x*, a big exponent *e*, and two *n-residue* outputs *vp* and *v*. On exit $v=V_e(x) \bmod n$ and $vp=V_{e-1}(x) \bmod n$ where *n* is the current Montgomery modulus. Only *v* is returned if *v* and *vp* are the same big variable.

Return value: None

Restrictions: 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.

9.2.27 nres_modadd

Function: `void` **`nres_modadd(x,y,z)`**
`big x,y,z;`

Module: mrmonty.c

Description: Modular addition of two *n-residues*

Parameters: Three *n-residue* numbers *x*, *y*, and *z*.
On exit $z=x+y \bmod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: Must be preceded by a call to **prepare_monty.**

9.2.28 nres_moddiv

Function: `int nres_moddiv(x,y,z)`
`big x,y,z;`

Module: mrmonty.c

Description: Modular division of two *n-residues*.

Parameters: Three *n-residue* numbers *x, y* and *z*.
On exit *z =x/y mod n*, where *n* is the current Montgomery modulus.

Return value: GCD of *y* and *n* as an integer, if possible, or **MR_TOOBIG**. Should be 1 for a valid result.

Restrictions: Must be preceded by call to **prepare_monty** and conversion of parameters to *n-residue* form. Parameters *x* and *y* must be distinct.

9.2.29 nres_modmult

Function: `void nres_modmult(x,y,z)`
`big x,y,z;`

Module: mrmonty.c

Description: 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: Three *n-residue* numbers *x*, *y* and *z*
On exit *z =xy mod n*, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: Must be preceded by call to **prepare_monty** and conversion of parameters to *n-residue* form.

9.2.30 nres_modsub

Function: `void` **`nres_modsub(x,y,z)`**
`big x,y,z;`

Module: mrmonty.c

Description: Modular subtraction of two *n-residues*

Parameters: Three *n-residue* numbers *x*, *y*, and *z*.
On exit $z=x-y \bmod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: Must be preceded by a call to **prepare_monty.**

9.2.31 nres_multi_inverse

Function: `BOOL` **`nres_multi_inverse`**`(m,x,w)`
`int m;`
`big *x,*w;`

Module: mrmonty.c

Description: 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: The number of inverses required *m*, an array *x*[.] of *m n-residues* whose inverses are wanted, and an array of their inverses *w*[.].

Return value: TRUE if successful, otherwise FALSE.

Restrictions: The parameters *x* and *w* must be distinct.

9.2.32 nres_negate

Function: `void` **`nres_negate`**`(x,w)`
`big x,w;`

Module: mrmonty.c

Description: Modular negation.

Parameters: Two *n-residue* numbers *x* and *w*.
On exit *w= -x mod n*, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: Must be preceded by a call to **prepare_monty.**

9.2.33 nres_powltr

Function: `void` **`powltr`**`(x,e,w)`
`int x;`
`big e,w;`

Module: mrpower.c

Description: Modular exponentiation of an *n-residue*

Parameters: An ordinary small integer *x*, a big number *e* and an *n-residue* result *w*.
On exit $w=x^e \bmod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: Must be preceded by call to **prepare_monty**. Note that the small integer *x* and the exponent are **not** converted to *n-residue* form.

9.2.34 nres_powmod

Function: `void` **`nres_powmod`**`(x,y,z)`
`big x,y,z;`

Module: mrpower.c

Description: Modular exponentiation of an *n-residue*.

Parameters: An *n-residue* number *x*, a big number *y* and an *n-residue* result *z*. On exit $z = x^y \bmod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: 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.

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 */
```

9.2.35 nres_powmod2

Function: `void` **`nres_powmod2`**`(x,y,a,b,w)`
`big x,y,a,b,w;`

Module: mrpower.c

Description: Calculate the product of two modular exponentiations involving *n-residues*.

Parameters: Three *n-residue* numbers *x*, *a* and *w*, and two big integers *y* and *b*. On exit $w = x^y . a^b \bmod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: 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.

9.2.36 nres_powmodn

Function:
```
void nres_powmodn(m,x,y,w)
int m;
big *x,*y,w;
```

Module: mrpower.c

Description: Calculate the product of *m* modular exponentiations involving *n-residues*. Extra memory is allocated internally by this function.

Parameters: The integer *m,* an array of *m n-residue* numbers *x*, an array of *m* big integers *y*, and an *n-residue w*.
On exit $w = x[0]^{y[0]} \cdot x[1]^{y[1]} \ldots \cdot x[m-1]^{y[m-1]} \bmod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: 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.

9.2.37 nres_premult

Function:
```
void nres_premult(x,k,w)
int k;
big x,w;
```

Module: mrmonty.c

Description: Multiplies an *n-residue* by a small integer.

Parameters: Two *n-residues x* and *w*, and a small integer *k*.
On exit $w = kx \bmod n$, where *n* is the current Montgomery modulus.

Return value: None

Restrictions: 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.

9.2.38 nres_sqroot

Function:
```
BOOL nres_sqroot(x,w)
big x,w;
```

Module: mrsroot.c

Description: Calculates the square root of an *n-residue* mod a prime modulus

Parameters: Two *n-residues x* and *w*.
On exit $w = \sqrt{x} \bmod n$ where *n* is the current Montgomery modulus.

Return value: TRUE if the square root exists, otherwise FALSE

Restrictions: Must be preceded by call to **prepare_monty** and conversion of the first parameter to *n-residue* form.

9.2.39 nroot

Function:
```
BOOL nroot(x,n,z)
big x,z;
int n;
```

Module: mrarth3.c

Description: Extracts lower approximation to a root of a big number.

Parameters: Two big numbers *x* and *z*, and an integer *n*.
On exit $z = \lfloor x^{1/n} \rfloor$.

Return value: Returns the boolean value TRUE if the root found is exact, otherwise returns FALSE.

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

9.2.40 nxprime

Function: `BOOL` **`nxprime`**`(w,x)`
`big w,x;`

Module: mrprime.c

Description: Find next prime number.

Parameters: Two big numbers *w* and *x*.
On exit *x* contains the next prime number greater than *w*.

Return value: TRUE if successful, FALSE otherwise.

Restrictions: None

9.2.41 nxsafeprime

Function: `BOOL n`**`xsafeprime`**`(type,subset,w,p)`
`int type,subset;`
`big w,p;`

Module: mrprime.c

Description: Find 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: The integer parameter *type* determines the type of safe prime as above. If the parameter *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.

Return value: TRUE if successful, FALSE otherwise

9.2.42 pow_brick

Function:
```
void pow_brick(binst,e,w)
brick *binst;
big e,w;
```

Module: mrbrick.c

Description: Carries out a modular exponentiation, using the precomputed values stored in the *brick* structure.

Parameters: A pointer to the current instance, a big exponent *e* and a big number *w*. On exit $w=g^e \bmod n$, where *g* and *n* are specified in the initial call to **brick_init.**

Return value: None

Restrictions: Must be preceded by a call to **brick_init**.

9.2.43 power

Function:
```
void power(x,n,z,w)
long n;
big x,z,w;
```

Module: mrarth3.c

Description: Raise a big number to an integer power.

Parameters: Two big numbers *x* and *z*, and an integer *n*. On exit $w=x^n$. If *w* and *z* are distinct, then $w=x^n \bmod z$

Return value: None

Restrictions: The value of *n* must be positive

9.2.44 powltr

Function:
```
int powltr(x,y,z,w)
int x;
big y,z,w;
```

Module: mrpower.c

Description: Raise 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 *z* is odd.

Parameters: An integer *x* and three bigs *y*, *z* and *w*.
On exit $w=x^y \bmod z$

Return value: The result expressed as an integer, if possible. Otherwise the value **MR_TOOBIG**.

Restrictions: The value of *y* must be positive. The parameters *w* and *z* must be distinct.

9.2.45 powmod

Function: `void powmod(x,y,z,w)`
`big x,y,z,w;`

Module: mrpower.c

Description: Raise 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 *z* 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: Four big numbers *x, y, z* and *w*.
On exit $w=x^y \bmod z$.

Return value: None

Restrictions: The value of *y* must be positive. The parameters *w* and *z* must be distinct.

9.2.46 powmod2

Function: `void powmod2(a,b,c,d,z,w)`
`big a,b,c,d,z,w;`

Module: mrpower.c

Description: Calculate 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: Six big numbers *a ,b, c, d, z* and *w*.
On exit $w=a^b.c^d \bmod z$.

Return value: None

Restrictions: The values of *b* and *d* must be positive. The parameters *w* and *z* must be distinct. The modulus *z* must be odd.

Example: See *dssver.c*

9.2.47 powmodn

Function: `void powmodn(m,a,b,z,w)`
`int m;`
`big *a,*b,z,w;`

Module: mrpower.c

Description: Calculate the product of *m* modular exponentiations. This is quicker than doing *m* separate exponentiations, and is useful for certain Cryptographic protocols. Extra memory is allocated internally for this function

Parameters: An integer *m*, two big number arrays *a[]* and *b[]*, and two big numbers *z* and *w*. On exit $w=a[0]^{b[0]}.a[1]^{b[1]} \ldots . a[m-1]^{b[m-1]} \bmod z$.

Return value: None

Restrictions: The values of *b[]* must be positive. The parameters *w* and *z* must be distinct. The modulus *z* must be odd. The underlying number base must be a power of 2.

9.2.48 prepare_monty

Function: `void prepare_monty(n)`
`big n;`

Module: mrmonty.c

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

Parameters: A big number *n*, which is to be the Montgomery modulus.

Return value: None

Restrictions: 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**.

9.2.49 redc

Function: `void redc(x,y)`
`big x,y;`

Module: mrmonty.c

Description: Converts an *n-residue* back to normal form.

Parameters: Two big numbers *x* and *y*.
On exit *y* is the normal form of the *n-residue x*.

Return value: None

Restrictions: Use must be preceded by call to **prepare_monty**.

9.2.50 scrt

Function:
```
void scrt(psc,rem,x)
small_chinese *psc;
int *rem;
big x;
```

Module: mrscrt.c

Description: Applies Chinese Remainder Theorem (for small prime moduli).

Parameters: A pointer *psc* to the current instance of the Chinese Remainder Theorem. On exit *x* contains the big number which yields the given integer remainders *rem[.]* when it is divided by the integer moduli specified in a prior call to **scrt_init**.

Return value: None

Restrictions: The routine **scrt_init** must be called first.

9.2.51 scrt_end *

Function:
```
void scrt_end(psc)
small_chinese *psc;
```

Module: mrscrt.c

Description: Cleans up after an application of the Chinese Remainder Theorem.

Parameters: A pointer to the current instance of the Chinese Remainder Theorem..

Return value: None

9.2.52 Restrictions: None

scrt_init

Function:
```
BOOL scrt_init(psc,np,m)
small_chinese *psc;
int np;
int *m;
```

Module: mrscrt.c

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

Parameters: A pointer to the current instance *psc*. The number of co-prime moduli *np*, and an array of at least two integer moduli *m[.]*.

Return value: TRUE if all went well, FALSE if there was a problem.

Restrictions: None

9.2.53 sftbit

Function:
```
void sftbit(x,n,z)
big x,z;
int n;
```

Module: mrbits.c

Description: Shifts a big integer left or right by a number of bits.

Parameters: The big parameter *x* is shifted by *n* bits, to give *z*. Positive *n* shifts to the left, negative to the right.

Return value: None

Restrictions: None

9.2.54 smul *

Function:
```
unsigned int smul(x,y,z)
Unsigned int x,y,z;
```

Module: mrsmall.c

Description: Multiplies two integers mod a third

Parameters: Integers *x, y* and *z*

Return value: *x.y mod z*

9.2.55 spmd *

Function:
```
unsigned int spmd(x,y,z)
Unsigned int x,y,z;
```

Module: mrsmall.c

Description: Raises an integer to an integer power modulus a third

Parameters: Integers *x, y*, and *z*

Return value: $x^y \bmod z$

Restrictions: None

9.2.56 sqrmp *

Function:
```
unsigned int sqrmp(x,p)
Unsigned int x,p;
```

Module: mrsmall.c

Description: Calculates the square root of an integer mod an integer prime number

Parameters: An integer *x* and a prime number *p*

Return value: $\sqrt{x} \bmod p$, or 0 if root does not exist

Restrictions: Return value unpredictable if *p* is not prime

9.2.57 sqroot

Function: `BOOL sqroot(x,p,w)`
`big x,p;`

Module: mrsroot.c

Description: Calculates the square root of a big integer mod a big integer prime.

Parameters: Two big integers *x* and *w*, and a big prime number *p*.
On exit $w = \sqrt{x} \bmod p$ if the square root exists, otherwise *w=0*. Note that the "other" square root may be found by subtracting *w* from *p*.

Return value: TRUE if the square root exists, FALSE otherwise.

Restrictions: The number *p* must be prime.

9.2.58 trial_division

Function: `int trial_division(x,y)`
`big x,y;`

Module: mrprime.c

Description: 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 **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: Two big integers *x* and *y*.

Return value: 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 is 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.

9.2.59 xgcd

Function: `int xgcd(x,y,xd,yd,z)`
`big x,y,xd,yd,z;`

Module: mrxgcd.c

Description: 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: Five big numbers *x,y,xd,yd* and *z*.
On exit *z=gcd(x,y)=x.xd+y.yd*

Return value: GCD as integer, if possible, otherwise **MR_TOOBIG**

Restrictions: 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) */
```

9.2.60 zzn2_add

Function: `void zzn2_add(x,y,z)`
`zzn2 *x,*y,*z;`

Module: mrzzn2.c

Description: Adds two zzn2 variables.

Parameters: Three zzn2 variables *x*, *y* and *z*. On exit *z=x+y*

Return value: None

9.2.61 zzn2_compare *

Function: `BOOL zzn2_compare(x,y)`
`zzn2 *x,*y;`

Module: mrzzn2.c

Description: Compares two zzn2 variables for equality

Parameters: Two zzn2 values *x* and *y*

Return value: TRUE if *x=y*, otherwise FALSE

9.2.62 zzn2_conj

Function: `void zzn2_conj(x,y)`
`zzn2 *x,*y;`

Module: mrzzn2.c

Description: Finds the conjugate of a zzn2

Parameters: Two zzn2 variables *x* and *y*. If *x=a+ib*, then on exit *y=a-ib*

Return value: None

9.2.63 zzn2_copy *

Function: `void zzn2_copy(x,y)`
`zzn2 *x,*y;`

Module: mrzzn2.c

Description: Copies one zzn2 to another

Parameters: Two zzn2 variables *x* and *y*. On exit *y=x*

Return value: None

9.2.64 zzn2_from_big

Function: `void zzn2_from_big(a,x)`
`big a;`
`zzn2 *x;`

Module: mrzzn2.c

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

Parameters: A big integer *a* and a zzn2 *x*. On exit *x=a*

Return value: None

9.2.65 zzn2_from_bigs

Function: `void zzn2_from_bigs(a,b,x)`
`big a,b;`
`zzn2 *x;`

Module: mrzzn2.c

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

Parameters: Two big integers *a* and *b* and a zzn2 *x*. On exit *x=a+ib*

Return value: None

9.2.66 zzn2_from_int

Function:
```
void zzn2_from_int(a,x)
int a;
zzn2 *x;
```

Module: mrzzn2.c

Description: Converts an integer to zzn2 format

Parameters: An integer *a* and a zzn2 *x*. On exit *x=a*

Return value: None

9.2.67 zzn2_from_ints

Function:
```
void zzn2_from_ints(a,b,x)
int a,b;
zzn2 *x;
```

Module: mrzzn2.c

Description: Creates a zzn2 from two integers

Parameters: Two integers *a* and *b* and a zzn2 *x*. On exit *x=a+ib*

Return value: None

9.2.68 zzn2_from_zzn

Function:
```
void zzn2_from_zzn(a,x)
big a;
zzn2 *x;
```

Module: mrzzn2.c

Description: Creates a zzn2 from a big already in *n-residue* format.

Parameters: A big *a* and a zzn2 *x*. On exit *x=a*

Return value: None

9.2.69 zzn2_from_zzns

Function:
```
void zzn2_from_zzns(a,b,x)
big a,b;
zzn2 *x;
```

Module: mrzzn2.c

Description: Creates a zzn2 from two bigs already in *n-residue* format.

Parameters: Two bigs *a* and *b* and a zzn2 *x*. On exit *x=a+ib*

Return value: None

9.2.70 zzn2_imul

Function:
```
void zzn2_simul(x,y,z)
zzn2 *x,*z;
int y;
```

Module: mrzzn2.c

Description: Multiplies a zzn2 variable by an integer.

Parameters: Two zzn2 variables *x* and *z*, and an integer *y*. On exit *z=x.y*

Return value: None

9.2.71 zzn2_inv

Function:
```
BOOL zzn2_inv(x)
zzn2 *x;
```

Module: mrzzn2.c

Description: In-place inversion of a zzn2 variable

Parameters: A single zzn2 variable *x*. On exit *x=1/x*.

Return value: None

9.2.72 zzn2_isunity

Function:
```
BOOL zzn2_isunity(x)
zzn2 *x;
```

Module: mrzzn2.c

Description: Tests a zzn2 value for equality to one

Parameters: A single zzn2 variable *x*

Return value: TRUE if *x* is one, otherwise FALSE.

9.2.73 zzn2_iszero *

Function:
```
BOOL zzn2_iszero(x)
zzn2 *x;
```

Module: mrzzn2.c

Description: Tests a zzn2 variable for equality to zero

Parameters: A single zzn2 value *x*

Return value: TRUE if *x* is zero, otherwise FALSE.

9.2.74 zzn2_mul

Function:
```
void zzn2_mul(x,y,z)
zzn2 *x,*y,*z;
```

Module: mrzzn2.c

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

Parameters: Three zzn2 variables *x, y* and *z*. On exit *z=x.y*

Return value: None

9.2.75 zzn2_negate

Function: `void zzn2_negate(x,y)`
`zzn2 *x,*y;`

Module: mrzzn2.c

Description: Negate a zzn2.

Parameters: Two zzn2 variables *x* and *y*. On exit *y=-x*

Return value: None

9.2.76 zzn2_sadd

Function:
```
void zzn2_sadd(x,y,z)
zzn2 *x,*z;
big y;
```

Module: mrzzn2.c

Description: Adds a big in *n-residue* format to a zzn2 .

Parameters: Two zzn2 variables *x* and *z*, and a big variable *y*. On exit *z=x+y*

Return value: None

9.2.77 zzn2_smul

Function:
```
void zzn2_smul(x,y,z)
zzn2 *x,*z;
big y;
```

Module: mrzzn2.c

Description: Multiplies a zzn2 variable by a big in *n-residue*.

Parameters: Two zzn2 variables *x* and *z*, and a big variable *y*. On exit *z=x.y*

Return value: None

9.2.78 zzn2_ssub

Function:
```
void zzn2_ssub(x,y,z)
zzn2 *x,*z;
big y;
```

Module: mrzzn2.c

Description: Subtracts a big in *n-residue* format from a zzn2 .

Parameters: Two zzn2 variables *x* and *z*, and a big variable *y*. On exit *z=x-y*

Return value: None

9.2.79 zzn2_sub

Function:
```
void zzn2_sub(x,y,z)
zzn2 *x,*y,*z;
```

Module: mrzzn2.c

Description: Subtracts two zzn2 variables .

Parameters: Three zzn2 variables *x, y* and *z*. On exit *z=x-y*

Return value: None

9.2.80 zzn2_timesi

Function:
```
BOOL zzn2_timesi(x)
zzn2 *x;
```

Module: mrzzn2.c

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

Parameters: A single zzn2 variable *x*. If $x=a+ib$ then on exit $x=i^2b+ia$.

Return value: None

9.2.81 zzn2_zero *

Function: `void zzn2_iszero(x)`
`zzn2 *x;`

Module: mrzzn2.c

Description: Sets a zzn2 variable to zero

Parameters: A single zzn2 variable *x*. On exit *x=0*

Return value: None

9.3 Elliptic curve routines

9.3.1 ebrick_init

Function:
```
BOOL ebrick_init(binst,x,y,a,b,n,w,nb)
ebrick *binst;
big x,y;
big a,b,n;
int w,nb;
```

Module: mrebrick.c

Description: Initialises an instance of the Comb method for GF(p) elliptic curve multiplication with precomputation. Internally memory is allocated for 2^w 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: A pointer to the current instance *binst*, the fixed point G=(x,y) on the curve $y^2 = x^3 + ax + b$, the modulus n, and the maximum number of bits to be used in the exponent *nb*.

Return value: TRUE if all went well, FALSE if there was a problem.

Restrictions: 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`.

9.3.1 ebrick2_init

Function:
```
BOOL ebrick2_init(binst,x,y,A,B,m,a,b,c,nb)
ebrick2 *binst;
big x,y;
big A,B;
int m,a,b,c,nb;
```

Module: mrec2m.c

Description: Initialises an instance of the Comb method for GF(2^m) elliptic curve multiplication with precomputation. The field is defined with respect to the trinomial basis t^m+t^a+1 or the pentanomial basis $t^m+t^a+t^b+t^c+1$. Internally memory is allocated for 2^w 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: A pointer to the current instance *binst*, the fixed point $G=(x,y)$ on the curve $y^2 + xy = x^3 + Ax^2 + B$, the field parameters *m, a, b, c*, and the maximum number of bits to be used in the exponent *nb*. Set $b = 0$ for a trinomial basis.

Return value: TRUE if all went well, FALSE if there was a problem.

Restrictions: 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`.

9.3.2 ebrick_end *

Function:
```
void ebrick_end(binst)
ebrick *binst
```

Module: mrebrick.c

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

Parameters: A pointer to the current instance

Return value: None

Restrictions: None

9.3.3 ebrick2_end *

Function: `void ebrick2_end(binst)`
`ebrick2 *binst`

Module: mrec2m.c

Description: Cleans up after an application of the Comb method for GF(2^m) elliptic curves.

Parameters: A pointer to the current instance

Return value: None

Restrictions: None

9.3.4 ecurve_add

Function: `void ecurve_add(p,pa)`
`epoint *p,*pa;`

Module: mrcurve.c

Description: 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: Two points on the current active curve*, pa* and *p*. On exit *pa=pa+p.*

Return value: None

Restrictions: The input points must actually be on the current active curve.

9.3.5 ecurve2_add

Function:
```
void ecurve2_add(p,pa)
epoint *p,*pa;
```

Module: mrec2m.c

Description: Adds two points on a GF(2^m) 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: Two points on the current active curve*, pa* and *p*. On exit *pa=pa+p.*

Return value: None

Restrictions: The input points must actually be on the current active curve.

9.3.6 ecurve_init

Function:
```
void ecurve_init(A,B,p,type)
big A,B,p;
int type;
```

Module: mrcurve.c

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

Parameters: Three big numbers *A*, *B* and *p*. The *type* parameter must be either **MR_PROJECTIVE** or **MR_AFFINE**, and specifies whether projective or affine co-ordinates should be used internally. Normally the former is faster.

Return value: None

Restrictions: Allocated memory will be freed when the current instance of MIRACL is terminated by a call to **mirexit**. However only one elliptic curve, GF(*p*) or GF(2^m) may be active within a single MIRACL instance. In addition, a call to a function like **powmod** will overwrite the stored modulus. This can be restored by a repeat call to **ecurve_init**

9.3.7 ecurve2_init

Function:
```
BOOL ecurve2_init(m,a,b,c,A,B,check,type)
big A,B;
int m,a,b,c,type;
BOOL check;
```

Module: mrec2m.c

Description: Initialises the internal parameters of the current active 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+t^a+1 or the pentanomial basis $t^m+t^a+t^b+t^c+1$. This routine can be called subsequently with the parameters of a different curve.

Parameters: The fixed point $G=(x,y)$ on the curve $y^2 + xy = x^3 + Ax^2 + B$, the field parameters *m, a, b, c.* Set $b = 0$ for a trinomial basis. The *type* parameter must be either **MR_PROJECTIVE** or **MR_AFFINE**, and specifies whether projective or affine co-ordinates should be used internally. Normally the former is faster. If *check* is TRUE a check is made that the specified basis is irreducible. If FALSE, this basis validity check, which is time-consuming, is suppressed.

Return value: TRUE if parameters make sense, otherwise FALSE.

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

9.3.8 ecurve_mult

Function:
```
void ecurve_mult(k,p,pa)
big k;
epoint *p,*pa;
```

Module: mrcurve.c

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

Parameters: A big number *k*, and two points *p* and *pa*. On exit *pa=k*p.*

Return value: None

Restrictions: The point *p* must be on the active curve.

9.3.9 ecurve2_mult

Function:
```
void ecurve2_mult(k,p,pa)
big k;
epoint *p,*pa;
```

Module: mrec2m.c

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

Parameters: A big number *k*, and two points *p* and *pa*. On exit *pa=k*p.*

Return value: None

Restrictions: The point *p* must be on the active curve.

9.3.10 ecurve_mult2

Function:
```
void ecurve_mult2(k1,p1,k2,p2,pa)
big k1,k2;
epoint *p1,*p2,*pa;
```

Module: mrcurve.c

Description: Calculates the point *k1.p1+k2.p2* on a GF(*p*) elliptic curve. This is quicker than doing two separate multiplications and an addition. Useful for certain cryptosystems. (See *ecsver.c* for example)

Parameters: Two big integers *k1* and *k2*, and three points *p1, p2* and *pa*.
On exit *pa = k1.p1+k2.p2*

Return value: None

Restrictions: The points *p1* and *p2* must be on the active curve.

9.3.11 ecurve2_mult2

Function:
```
void ecurve2_mult2(k1,p1,k2,p2,pa)
big k1,k2;
epoint *p1,*p2,*pa;
```

Module: mrec2m.c

Description: Calculates the point *k1.p1+k2.p2* on a GF(2^m) elliptic curve. This is quicker than doing two separate multiplications and an addition. Useful for certain cryptosystems. (See *ecsver2.c* for example)

Parameters: Two big integers *k1* and *k2*, and three points *p1, p2* and *pa*.
On exit *pa = k1.p1+k2.p2*

Return value: None

Restrictions: The points *p1* and *p2* must be on the active curve.

9.3.12 ecurve_multi_add

Function:
```
void ecurve_multi_add(m,x,w)
int m;
epoint **x,**w;
```

Module: mrcurve.c

Description: Simultaneously adds pairs of points on the active GF(*p*) curve. This is much quicker than adding them individually, but only when using Affine co-ordinates.

Parameters: An integer *m* and two arrays of points *w* and *x*. On exit *w[i]=w[i]+x[i]* for *i* =0 to *m*-1

Return value: None

Restrictions: Only useful when using Affine co-ordinates.

See also: **ecurve_init** and **nres_multi_inverse**, which is used internally.

9.3.13 ecurve2_multi_add

Function:
```
void ecurve2_multi_add(m,x,w)
int m;
epoint **x,**w;
```

Module: mrec2m.c

Description: Simultaneously adds pairs of points on the active GF(2^m) curve. This is much quicker than adding them individually, but only when using Affine co-ordinates.

Parameters: An integer *m* and two arrays of points *w* and *x*. On exit *w[i]=w[i]+x[i]* for *i* =0 to *m*-1

Return value: None

Restrictions: Only useful when using Affine co-ordinates.

See also: **ecurve2_init**

9.3.14 ecurve_multn

Function:
```
void ecurve_multn(n,k,p,pa)
int n;
big *k;
epoint **p;
```

Module: mrcurve.c

Description: Calculates the point *k[0].p[0] + k[1].p[1] + ... + k[n-1].p[n-1]* on a GF(*p*) elliptic curve, for *n>2*.

Parameters: An integer *n*, an array of *n* big numbers *k[]*, and an array of *n* points. The result is returned in *pa*.

Return value: None

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

9.3.15 ecurve2_multn

Function:
```
void ecurve2_multn(n,k,p,pa)
int n;
big *k;
epoint **p;
```

Module: mrec2m.c

Description: Calculates the point *k[0].p[0] + k[1].p[1] + ... + k[n-1].p[n-1]* on a GF(2^m) elliptic curve, for *n>2*.

Parameters: An integer *n*, an array of *n* big numbers *k[]*, and an array of *n* points. The result is returned in *pa*.

Return value: None

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

9.3.16 ecurve_sub

Function:
```
void ecurve_sub(p,pa)
epoint *p,*pa;
```

Module: mrcurve.c

Description: 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: Two points on the current active curve, *pa* and *p*. On exit *pa = pa-p.*

Return value: None

Restrictions: The input points must actually be on the current active curve.

9.3.17 ecurve2_sub

Function: `void` **`ecurve2_sub`**`(p,pa)`
`epoint *p,*pa;`

Module: mrec2m.c

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

Parameters: Two points on the current active curve, *pa* and *p*. On exit *pa* = *pa-p.*

Return value: None

Restrictions: The input points must actually be on the current active curve.

9.3.18 epoint_comp

Function: `BOOL` **`epoint_comp`**`(p1,p2)`
`epoint *p1,*p2;`

Module: mrcurve.c

Description: Compares two points on the current active GF(*p*) elliptic curve.

Parameters: Two points *p1* and *p2*.

Return Value: TRUE if the points are the same, otherwise FALSE.

Restrictions: None

9.3.19 epoint2_comp

Function: `BOOL` **`epoint2_comp`**`(p1,p2)`
`epoint *p1,*p2;`

Module: mrec2m.c

Description: Compares two points on the current active GF(2^m) elliptic curve.

Parameters: Two points *p1* and *p2*.

Return Value: TRUE if the points are the same, otherwise FALSE.

Restrictions: None

9.3.20 epoint_copy *

Function: `void` **`epoint_copy`**`(p1,p2)`
`epoint *p1,*p2;`

Module: mrcurve.c

Description: Copies one point to another on a GF(*p*) elliptic curve.

Parameters: Two points *p1* and *p2*. On exit *p2=p1*.

Return value: None

Restrictions: None

9.3.21 epoint2_copy *

Function: `void epoint2_copy(p1,p2)`
`epoint *p1,*p2;`

Module: mrec2m.c

Description: Copies one point to another on a GF(2^m) elliptic curve.

Parameters: Two points *p1* and *p2*. On exit *p2=p1*.

Return value: None

Restrictions: None

9.3.22 epoint_free *

Function: `void epoint_free(p)`
`epoint *p;`

Module: mrcore.c

Description: Frees memory associated with a point on a GF(*p*) elliptic curve.

Parameters: A point *p*.

Return value: None

Restrictions: None

9.3.23 epoint_get

Function:
```
int epoint_get(p,x,y)
epoint *p;
big x,y;
```

Module: mrcurve.c

Description: Normalises a point and extracts its *(x,y)* co-ordinates on the active GF(*p*) elliptic curve.

Parameters: A point *p*, and two big integers *x* and *y*. If *x* and *y* are not distinct variables on entry then only the value of *x* is returned.

Return value: The least significant bit of *y*. Note that it is possible to reconstruct a point from its *x* co-ordinate and just the least significant bit of *y*. Often such a "compressed" description of a point is useful.

Restrictions: The point *p* must be on the active curve.

Example:
```
i=epoint_get(p,x,x);
     /* extract x co-ordinate and lsb of y */
```

9.3.24 epoint_getxyz

Function:
```
void epoint_getxyz(p,x,y,z)
epoint *p;
big x,y,z;
```

Module: mrcurve.c

Description: Extracts the raw (*x,y,z*) co-ordinates of a point on the active GF(*p*) elliptic curve.

Parameters: A point *p*, and three big integers *x, y* and *z*. If any of these is NULL that coordinate is not returned.

Return value: None

Restrictions: The point *p* must be on the active curve.

9.3.25 epoint2_get

Function:
```
int epoint2_get(p,x,y)
epoint *p;
big x,y;
```

Module: mrec2m.c

Description: Normalises a point and extracts its *(x,y)* co-ordinates on the active GF(2^m) elliptic curve.

Parameters: A point *p*, and two big integers *x* and *y*. If *x* and *y* are not distinct variables on entry then only the value of *x* is returned.

Return value: The least significant bit of *y/x*. Note that it is possible to reconstruct a point from its *x* co-ordinate and just the least significant bit of *y/x*. Often such a "compressed" description of a point is useful.

Restrictions: The point *p* must be on the active curve.

Example:
```
i=epoint_get(p,x,x);
     /* extract x co-ordinate and lsb of y/x */
```

9.3.26 epoint2_getxyz

Function: `void` **`epoint2_getxyz`**`(p,x,y,z)`
`epoint *p;`
`big x,y,z;`

Module: mrcurve.c

Description: Extracts the raw (*x,y,z*) co-ordinates of a point on the active GF(2^m) elliptic curve.

Parameters: A point *p*, and three big integers *x, y* and *z*. If any of these is NULL that coordinate is not returned.

Return value: None

Restrictions: The point *p* must be on the active curve.

9.3.27 epoint_init

Function: `epoint*` **`epoint_init`**`()`

Module: mrcore.c

Description: Assigns memory to a point on a GF(*p*) elliptic curve, and initialises it to the "point at infinity".

Parameters: None.

Return value: A point *p* (in fact a pointer to a structure allocated from the heap).

Restrictions: It is the C programmers 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.

9.3.28 epoint_init_mem

Function:
```
epoint* epoint_init_mem(mem,index)
char *mem;
int index;
```

Module: mrcore.c

Description: 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: A pointer to the pre-allocated array *mem*, and an index into that array. Each index should be unique.

Return value: An initialised elliptic curve point.

Restrictions: Sufficient memory must have been allocated and pointed to by *mem*.

9.3.29 epoint_norm

Function:
```
BOOL epoint_norm(p)
epoint *p;
```

Module: mrcurve.c

Description: 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* co-ordinate)

Parameters: A point on the current active elliptic curve.

Return value: TRUE if successful.

9.3.30 epoint2_norm

Function:
```
BOOL epoint2_norm(p)
epoint *p;
```

Module: mrec2m.c

Description: Normalises a point on the current active GF(2^m) 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* co-ordinate)

Parameters: A point on the current active elliptic curve.

Return value: TRUE if successful.

9.3.31 epoint_set

Function:
```
BOOL epoint_set(x,y,lsb,p)
big x,y;
int lsb;
epoint *p;
```

Module: mrcurve.c

Description: Sets a point on the current active GF(*p*) elliptic curve (if possible).

Parameters: The integer co-ordinates *x* and *y* of the point *p*. If *x* and *y* are not distinct variables then *x* only is passed to the function, and *lsb* 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). On exit *p=(x,y)*.

Return value: TRUE if the point exists on the current active point, otherwise FALSE.

Restrictions: None

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

9.3.32 epoint2_set

Function:
```
BOOL epoint2_set(x,y,lsb,p)
big x,y;
int lsb;
epoint *p;
```

Module: mrec2m.c

Description: Sets a point on the current active GF(2^m) elliptic curve (if possible).

Parameters: The integer co-ordinates *x* and *y* of the point *p*. If *x* and *y* are not distinct variables then *x* only is passed to the function, and *lsb* is taken as the least significant bit of *y/x.* 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 field square root). On exit *p=(x,y)*.

Return value: TRUE if the point exists on the current active point, otherwise FALSE.

Restrictions: None

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

9.3.33 epoint_x

Function:
```
BOOL epoint_x(x)
big x;
```

Module: mrcurve.c

Description: Tests to see if the parameter *x* is a valid co-ordinate of a point on the curve. It is faster to test an *x* co-ordinate 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: The integer coordinate *x*.

Return value: TRUE if *x* is the coordinate of a curve point, otherwise FALSE

Restrictions: None

9.3.34 mul_brick

Function:
```
int mul_brick(binst,e,x,y)
ebrick *binst;
big e,x,y;
```

Module: mrebrick.c

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

Parameters: A pointer to the current instance, a big exponent *e* and a big number *w*. On exit (*x,y*) = *e.G mod n*, where *G* and *n* are specified in the initial call to **ebrick_init.** If *x* and *y* are not distinct variables, only *x* is returned.

Return value: The least significant bit of *y*.

Restrictions: Must be preceded by a call to **ebrick_init**.

9.3.35 mul2_brick

Function:
```
int mul2_brick(binst,e,x,y)
ebrick2 *binst;
big e,x,y;
```

Module: mrec2m.c

Description: Carries out a GF(2^m) elliptic curve multiplication using the precomputed values stored in the *ebrick2* structure.

Parameters: A pointer to the current instance, a big exponent *e* and a big number *w*. On exit (*x,y*) = *e.G*, where *G* is specified in the initial call to **ebrick2_init.** If *x* and *y* are not distinct variables, only *x* is returned.

Return value: The least significant bit of *y/x*.

Restrictions: Must be preceded by a call to **ebrick2_init**.

9.3.36 point_at_infinity *

Function:
```
BOOL point_at_infinity(p)
epoint *p;
```

Module: mrcore.c

Description: Tests if an elliptic curve point is the "point at infinity".

Parameters: An elliptic curve point *p*.

Return value: TRUE if *p* is the point-at-infinity, otherwise FALSE.

Restrictions: The point must be initialised.

9.4 Encryption Routines

9.4.1 aes_decrypt *

Function: `mr_unsign32 aes_decrypt(a,buff)`
`aes *a;`
`char *buff;`

Module: mraes.c

Description: Decrypts a 16 or *n* byte input buffer in situ.

Parameters: Pointer to an initialised instance of an *aes* structure defined in *miracl.h*, and to the buffer of bytes to be decrypted. 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_CFB***n*, **MR_OFB***n* or **MR_PCFB***n*) then *n* bytes will be decrypted.

Return value: In **MR_CFB***n* and **MR_PCFB***n* modes the *n* byte(s) that were shifted off the end of the input register as result of decrypting the *n* input byte(s), otherwise 0.

9.4.2 aes_encrypt *

Function: `mr_unsign32 aes_encrypt(a,buff)`
`aes *a;`
`char *buff;`

Module: mraes.c

Description: Encrypts a 16 or *n* byte input buffer in situ.

Parameters: Pointer to an initialised instance of an *aes* structure defined in *miracl.h*, and to the buffer of bytes to be encrypted. 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_CFB***n*, **MR_OFB***n* or **MR_PCFB***n*) then a *n* bytes will be encrypted.

Return value: In **MR_CFB***n* and **MR_PCFB***n* 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.

9.4.3 aes_end *

Function:
```
void aes_end(a)
aes *a;
```

Module: mraes.c

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

Parameters: Pointer to an initialised instance of an *aes* structure defined in *miracl.h*

Return value: None

9.4.4 aes_getreg *

Function:
```
void aes_getreg(a,ir)
aes *a;
char *ir;
```

Module: mraes.c

Description: 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: Pointer to an instance of the *aes* structure, defined in *miracl.h*, and a character array to hold the extracted 16-byte data.

Return value: None

9.4.5 aes_init *

Function:
```
BOOL aes_init(a,mode,nk,key,iv)
aes *a;
int mode,nk;
char key,iv;
```

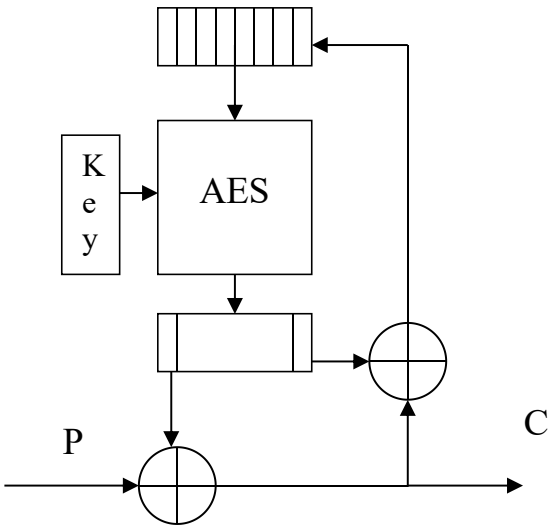
Module: mraes.c

Description: 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: Pointer to an instance of the *aes* structure defined in *miracl.h,* the *mode* of operation to be used, an integer *nk* which specifies the size of the Key in bytes, and pointers to the key itself and the optional Initialisation Vector (IV). The mode can be one of **MR_ECB** (Electronic Code Book), **MR_CBC** (Cipher Block Chaining), **MR_CFB***n* (Cipher Feed-back where *n* is 1, 2 or 4), **MR_PCFB***n* (error Propagating Cipher Feed-back where *n* is 1, 2 or 4) or **MR_OFB***n* (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_PCFB***n* is an invention of our own [Scott93]. See below.

The value of *nk* can be 16, 24 or 32. A 16 bytes initialisation vector *iv* should be specified for all modes other than MR_ECB, in which case it can be NULL.

Return value: TRUE if initialisation succeeded, otherwise FALSE.



MR_PCFB1 Mode

9.4.6 aes_reset *

Function:
```
void aes_reset(a,mode,iv)
aes *a;
int mode;
char *iv;
```

Module: mraes.c

Description: Resets the AES structure

Parameters: Pointer to an instance of the *aes* structure defined in *miracl.h*, an indication of the new *mode* of operation, and a pointer to a (possibly new) initialisation vector *iv*. See above for the modes allowed.

Return value: None

9.4.7 shs_init *

Function:
```
void shs_init(psh)
sha *psh;
```

Module: mrshs.c

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

Parameters: Pointer to an instance of a structure defined in *miracl.h*

Return value: None

9.4.8 shs_hash *

Function:
```
void shs_hash(psh,hash)
sha *psh;
char hash[20];
```

Module: mrshs.c

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

Parameters: Pointer to the current instance, and pointer to array to be filled.

Return value: None

9.4.9 shs_process *

Function:
```
void shs_process(psh,ch)
sha *psh;
int ch;
```

Module: mrshs.c

Description: 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: Pointer to the current instance, and character to be processed.

Return value: None

9.4.10 shs256_init *

Function:
```
void shs256_init(psh)
sha256 *psh;
```

Module: mrshs256.c

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

Parameters: Pointer to an instance of a structure defined in *miracl.h*

Return value: None

9.4.11 shs256_hash *

Function:
```
void shs256_hash(psh,hash)
sha256 *psh;
char hash[32];
```

Module: mrshs256.c

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

Parameters: Pointer to the current instance, and pointer to array to be filled.

Return value: None

9.4.12 shs256_process *

Function:
```
void shs256_process(psh,ch)
sha256 *psh;
int ch;
```

Module: mrshs256.c

Description: 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: Pointer to the current instance, and character to be processed.

Return value: None

9.4.13 shs384_init *

Function:
```
void shs384_init(psh)
sha384 *psh;
```

Module: mrshs512.c

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

Parameters: Pointer to an instance of a structure defined in *miracl.h*

Return value: None

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

9.4.14 shs384_hash *

Function:
```
void shs384_hash(psh,hash)
sha384 *psh;
char hash[48];
```

Module: mrshs512.c

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

Parameters: Pointer to the current instance, and pointer to array to be filled.

Return value: None

9.4.15 shs384_process *

Function:
```
void shs512_process(psh,ch)
sha384 *psh;
int ch;
```

Module: mrshs512.c

Description: 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: Pointer to the current instance, and character to be processed.

Return value: None

9.4.16 shs512_init *

Function:
```
void shs512_init(psh)
sha512 *psh;
```

Module: mrshs512.c

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

Parameters: Pointer to an instance of a structure defined in *miracl.h*

Return value: None

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

9.4.17 shs512_hash *

Function:
```
void shs512_hash(psh,hash)
sha512 *psh;
char hash[64];
```

Module: mrshs512.c

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

Parameters: Pointer to the current instance, and pointer to array to be filled.

Return value: None

9.4.18 shs512_process *

Function:
```
void shs512_process(psh,ch)
sha512 *psh;
int ch;
```

Module: mrshs512.c

Description: 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: Pointer to the current instance, and character to be processed.

Return value: None

9.4.19 strong_bigdig

Function:
```
void strong_bigdig(rng,n,b,x)
csprng *rng;
int n,b;
big x;
```

Module: mrstrong.c

Description: Generates a big random number of given length from the cryptographically strong generator *rng*.

Parameters A pointer to the random number generator *rng*. A big number *x* and two integers *n* and *b*. On exit *x* contains a big random number *n* digits long to base *b*.

Return value: None

Restrictions: The base *b* must be printable, that is $2 \le b \le 256$.

9.4.20 strong_bigrand

Function:
```
void strong_bigrand(rng,w,x)
csprng *rng;
big w,x;
```

Module: mrstrong.c

Description: Generates a cryptographically strong random big number *x* using the random number generator *rng* such that $0 \le x < w$

Parameters: Two big numbers *w* and *x*, and a random number generator *rng*

Return value: None

9.4.21 strong_init *

Function:
```
void strong_init(rng,rawlen,raw,tod)
csprng *rng;
int rawlen;
char *raw;
long tod;
```

Module: mrstrong.c

Description: Initialize the cryptographically strong random number generator *rng*.

Parameters: A pointer to the random number generator *rng*. An array *raw* of length *rawlen* and a 32-bit time-of-day value *tod*. These two sources are 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.

Return value: None

Example: See *test1363.c* and *p1363.c* for an example of use.

9.4.22 strong_kill *

Function: `void strong_kill(rng)`
`csprng *rng;`

Module: mrstrong.c

Description: Kills the internal state of the random number generator *rng*

Parameters: A pointer to a random number generator

Return value: None

9.4.23 strong_rng *

Function: `int strong_rng(rng)`
`csprng *rng;`

Module: mrstrong.c

Description: Generates a sequence of cryptographically strong random bytes.

Parameters: A pointer to a random number

Return value: A random byte

9.5 Floating-Slash Routines

9.5.1 build

Function:
```
void build(x,gener)
flash x;
int (*gener)();
```

Module: mrbuild.c

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

Parameters: The flash number created, and the generator function.

Return value: None

Example:
```
int phi(w,n)
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!

9.5.2 dconv

Function:
```
void dconv(d,x)
double d;
flash x;
```

Module: mrflash.c

Description: Converts a double to flash format.

Parameters: A double *d* and a flash variable *x*. On exit *x* will contain the flash equivalent of *d*.

Return value: None

9.5.3 denom

Function:
```
void denom(x,y)
flash x;
big y;
```

Module: mrcore.c

Description: Extract the denominator of a flash number

Parameters: A flash number *x* and a big number *y*. On exit *y* will contain the denominator of *x*.

Return value: None

Restrictions: None

9.5.4 facos

Function:
```
void facos(x,y)
flash x,y;
```

Module: mrflsh3.c

Description: Calculates arc-cosine of a flash number, using **fasin**.

Parameters: Two flash numbers *x* and *y*. On exit *y=arccos(x)*.

Return value: None

Restrictions: |*x*| must be less than or equal to 1.

9.5.5 facosh

Function: `void facosh(x,y)`
`flash x,y;`

Module: mrflsh4.c

Description: Calculates hyperbolic arc-cosine of a flash number.

Parameters: Two flash numbers *x* and *y*. On exit *y=arccosh(x)*.

Return value: None

Restrictions: |*x*| must be greater than or equal to 1.

9.5.6 fadd

Function: `void fadd(x,y,z)`
`flash x,y,z;`

Module: mrflash.c

Description: Add two flash numbers.

Parameters: Three flash numbers *x, y* and *z*. On exit *z=x+y*.

Return value: None

Restrictions: None

9.5.7 fasin

Function: `void fasin(x,y)`
`flash x,y;`

Module: mrflsh3.c

Description: Calculates arc-sin of a flash number, using **fatan**.

Parameters: Two flash numbers *x* and *y*. On exit *y=arcsin(x)*.

Return value: None

Restrictions: |*x*| must be less than or equal to 1.

9.5.8 fasinh

Function: `void fasinh(x,y)`
`flash x,y;`

Module: mrflsh4.c

Description: Calculates hyperbolic arc-sin of a flash number.

Parameters: Two flash numbers *x* and *y*. On exit *y=arcsinh(x)*.

Return value: None

9.5.9 fatan

Function: `void fatan(x,y)`
`flash x,y;`

Module: mrflsh3.c

Desciption: Calculates the arc-tangent of a flash number, using $O(n^{2.5})$method based on Newton's iteration.

Parameters: Two flash numbers *x* and *y*. On exit *y=arctan(x)*.

Return value: None

9.5.10 fatanh

Function: `void fatanh(x,y)`
`flash x,y;`

Module: mrflsh4.c

Desciption: Calculates the hyperbolic arc-tangent of a flash number.

Parameters: Two flash numbers *x* and *y*. On exit *y=arctanh(x)*.

Return value: None

Restrictions: x^2 must be less than 1

9.5.11 fcomp

Function:
```
int fcomp(x,y)
flash x,y;
```

Module: mrflash.c

Description: Compare two flash numbers.

Parameters: Two flash numbers *x* and *y*.

Return value: Returns -1 if *y*>*x*, +1 if *x*>*y* and 0 if *x*=*y*.

9.5.12 fconv

Function:
```
void fconv(n,d,x)
int n,d;
flash x;
```

Module: mrflash.c

Description: Convert a simple fraction to flash format.

Parameters: Integers *n* and *d*, and a flash number *x*.
On exit *x*=*n*/*d*

Return value: None

9.5.13 fcos

Function:
```
void fcos(x,y)
flash x,y;
```

Module: mrflsh3.c

Description: Calculates cosine of a given flash angle, using **ftan**.

Parameters: Two flash numbers *x* and *y*. On exit *y*=*cos(x)*.

Return value: None

Restrictions: None

9.5.14 fcosh

Function: `void fcosh(x,y)`
`flash x,y;`

Module: mrflsh4.c

Description: Calculates hyperbolic cosine of a given flash angle.

Parameters: Two flash numbers *x* and *y*. On exit *y=cosh(x)*.

Return value: None

9.5.15 fdiv

Function: `void fdiv(x,y,z)`
`flash x,y,z;`

Module: mrflash.c

Description: Divides two flash numbers.

Parameters: Three big numbers *x, y* and *z*. On exit *z=x/y*.

Return value: None

9.5.16 fdsize

Function: `double fdsize(x)`
`flash x;`

Module: mrdouble.c

Description: Converts a flash number to double format.

Parameters: A flash number *x*.

Return value: The value of the parameter *x* as a double.

Restrictions: The value of *x* must be representable as a double.

9.5.17 fexp

Function:
```
void fexp(x,y)
flash x,y;
```

Module: mrflsh2.c

Description: Calculates the exponential of a flash number using $O(n^{2.5})$ method.

Parameters: Two flash numbers *x* and *y*. On exit $y=e^x$.

Return value: None

Restrictions: None

9.5.18 fincr

Function:
```
void fincr(x,n,d,y)
big x,y;
int n,d;
```

Module: mrflash.c

Description: Add a simple fraction to a flash number.

Parameters: Two flash numbers *x* and *y*, and two integers *n* and *d*.
On exit $y = x + \frac{n}{d}$.

Return value: None

Restrictions: None

Example:
```
fincr(x,-2,3,x);
```
This subtracts two-thirds from the value of *x*.

9.5.19 flog

Function:
```
void flog(x,y)
flash x,y;
```

Module: mrflsh2.c

Description: Calculates the natural log of a flash number using $O(n^{2.5})$ method.

Parameters: Two flash numbers *x* and *y*. On exit *y=log(x)*.

Return value: None

Restrictions: None

9.5.20 flop

Function:
```
void flop(x,y,op,z)
flash x,y,z;
int *op;
```

Module: mrflash.c

Description: Perform primitive flash operation. Used internally.

Parameters: Three flash numbers *x, y* and *z*. On exit z=*Fn(x,y),* where the function performed depends on the parameter *op*. See source listing comments for more details.

Return value: None

Restrictions: None

9.5.21 fmodulo

Function: `void fmodulo(x,y,z)`
`flash x,y,z;`

Module: mrflash.c

Description: Find the remainder when one flash number is divided by another.

Parameters: Three flash numbers *x, y* and *z*. On exit *z=x mod y*;

Return value: None

Restrictions: None

9.5.22 fmul

Function: `void fmul(x,y,z)`
`flash x,y,z;`

Module: mrflash.c

Description: Multiply two flash numbers.

Parameters: Three flash numbers *x, y* and *z*. On exit *z=x.y*

Return value: None

Restrictions: None

9.5.23 fpack

Function:
```
void fpack(n,d,x)
flash x;
big n,d;
```

Module: mrcore.c

Description: Forms a flash number from big numerator and denominator.

Parameters: A flash number *x* and two big numbers *n* and *d*.
On exit $x = \frac{n}{d}$.

Return value: None

Restrictions: 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.

9.5.24 fpi

Function:
```
void fpi(x)
flash x;
```

Module: mrpi.c

Description: Calculates π using Guass-Legendre $O(n^2.log\ n)$ method. Note that on subsequent calls to this routine, π 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 π quickly).

Parameters: A flash number *x*. On exit $x=\pi$.

Return value: None

Restrictions: None. Internally allocated memory is freed when the current MIRACL instance is ended by a call to **mirexit**.

9.5.25 fpmul

Function:
```
void fpmul(x,n,d,y)
flash x,y;
int n,d;
```

Module: mrflash.c

Description: Multiplies a flash number by a simple fraction.

Parameters: Two flash numbers *x* and *y*, and two integers *n* and *d*.
On exit $y = x.\frac{n}{d}$

Return value: None

Restrictions: None

9.5.26 fpower

Function:
```
void fpower(x,n,y)
flash x,y;
int n;
```

Module: mrflsh1.c

Description: Raises a flash number to an integer power.

Parameters: Flash variables *x* and *y*, and an integer *n*.
On exit $y=x^n$.

Return value: None

9.5.27 fpowf

Function: `void fpowf(x,y,z)`
`flash x,y,z;`

Module: mrflsh2.c

Description: Raises a flash number to a flash power.

Parameters: Three flash numbers *x*, *y* and *z*. On exit $z=x^y$

Return value: None

9.5.28 frand

Function: `void frand(x)`
`flash x;`

Module: mrfrnd.c

Description: Generates a random flash number.

Parameters: A big number *x*. On exit *x* contains a flash random number in the range $0<x<1$.

Return value: None

9.5.29 frecip

Function: `void frecip (x,y)`
`flash x,y;`

Module: mrflash.c

Description: Calculates reciprocal of a flash number.

Parameters: Two flash numbers *x* and *y*. On exit $y=1/x$.

Return value: None

9.5.30 froot

Function:
```
BOOL froot(x,m,y)
flash x,y;
int m;
```

Module: mrflsh1.c

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

Parameters: Flash numbers *x* and *y*, and an integer *m*.
On exit *y* is the *m*-th root of *x*.

Return value: TRUE for exact root, otherwise FALSE

9.5.31 fsin

Function:
```
void fsin(x,y)
flash x,y;
```

Module: mrflsh3.c

Description: Calculates sine of a given flash angle. Uses **ftan**.

Parameters: Two flash numbers *x* and *y*. On exit *y=sin(x)*.

Return value: None

9.5.32 fsinh

Function:
```
void fsinh(x,y)
flash x,y;
```

Module: mrflsh4.c

Description: Calculates hyperbolic sine of a given flash angle.

Parameters: Two flash numbers *x* and *y*. On exit *y=sinh(x)*.

Return value: None

9.5.33 fsub

Function: `void fsub(x,y,z)`
`flash x,y,z;`

Module: mrflash.c

Description: Subtract two flash numbers.

Parameters: Three flash numbers *x, y* and *z*.
On exit *z=x-y*.

Return value: None

9.5.34 ftan

Function: `void ftan(x,y)`
`flash x,y;`

Module: mrflsh3.c

Description: Calculates the tan of a given flash angle, using an $O(n^{2.5})$ method.

Parameters: Two flash numbers *x* and *y*. On exit *y=tan(x)*.

Return value: None

9.5.35 ftanh

Function: `void ftanh(x,y)`
`flash x,y;`

Module: mrflsh4.c

Description: Calculates the hyperbolic tan of a given flash angle.

Parameters: Two flash numbers *x* and *y*. On exit *y=tanh(x)*.

Return value: None

Restrictions: None

9.5.36 ftrunc

Function:
```
void ftrunc(x,y,z)
flash x,z;
big y;
```

Module: mrflash.c

Description: Seperates a flash number to a big number and a flash remainder.

Parameters: Flash numbers *x* and *z*, and a big number *y*. On exit *y=int(x)* and *z* is the fractional remainder. If *y* is the same as *z*, only *int(x)* is returned.

Return value: None

Restrictions: None

9.5.37 numer

Function:
```
void numer(x,y)
flash x;
big y;
```

Module: mrcore.c

Description: Extract the numerator of a flash number.

Parameters: A flash number *x* and a big number *y*.
On exit *y* will contain the numerator of *x*.

Return value: None

Restrictions: None

9.5.38 mround

Function:
```
void mround(n,d,x)
flash x;
big n,d;
```

Module: mrround.c

Description: 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: A flash number *x* and two big numbers *n* and *d*. On exit *x=R{n/d}*, that is the flash number *n/d* rounded if necessary to fit the representation.

Return value: None

Restrictions: The denominator must be non-zero.

10. Instance variables

These variables are all member of the *miracl* structure defined in *miracl.h*. They are all accessed via the *mip* - the **M**iracl **I**nstance **P**ointer.

`BOOL EXACT;`	Initialised to TRUE. Set to FALSE if any rounding takes place during *flash* arithmetic.
`int INPLEN;`	Length of input string. Must be used when inputting binary data.
`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
`int IOBSIZ`	Size of I/O buffer.
`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.
`int ERNUM;`	Number of the last error that occurred.
`char IOBUFF[ ];`	Input/Output buffer.
`int NTRY;`	Number of iterations used in probabalistic primality test by isprime. Initialised to 6.
`int *PRIMES;`	Pointer to a table of small prime numbers.
`BOOL RPOINT;`	If set to TRUE numbers are output with a radix point. Otherwise they are output as fractions (the default).
`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.

11. MIRACL Error Messages

MIRACL error messages, diagnosis and response.

Message: `Number base too big for representation`
Diagnosis: An attempt has been made to input or output a number using a number base that is too big. For example outputting using a number base of 2^{32} is clearly impossible. For efficiency the largest possible internal number base is used, but numbers in this format should be input/output to a much smaller number base ≤ 256. This error typically arises when using using *innum*(.) or *otnum*(.) after *mirsys*(.,0*)*.
Response: Perform a change of base prior to input/output. For example set the instance variable **IOBASE** to 10, and then use *cinnum*(.) or *cotnum*(.). To avoid the change in number base, an alternatively is to initialise MIRACL using something like *mirsys*(400,16*)* which uses an internal base of 16. Now Hex I/O can be performed using *innum*(.) and *otnum*(.). Note that this will not impact performance on a 32-bit processor, as 8 Hex digits will be packed into each computer word.

Message: `Division by zero attempted`
Diagnosis: Self-explanatory
Response: Don't do it!

Message: `Overflow - Number too big`
Diagnosis: A number in a calculation is too big to be stored in its fixed length allocation of memory.
Response: Specify more storage space for all *big* and *flash* variables, by increasing the value of *n* in the initial call to *mirsys*(n.b);

Message: `Internal Result is Negative`
Diagnosis: This is an internal error that should not occur using the high level MIRACL functions. It may be caused by user-induced memory over-runs.
Response: Report to mike@computing.dcu.ie

Message: `Input Format Error`
Diagnosis: The number being input contains one or more illegal symbols with respect to the current I/O number base. For example this error might occur if **IOBASE** is set to 10, and a Hex number is input.
Response: Re-input the number, and be careful to use only legal symbols. Note that for Hex input only upper-case A-F are permissible.

Message: `Illegal number base`
Diagnosis: The number base specified in the call to *mirsys*(.) is illegal. For example a number base of 1 is not allowed.
Response: Use a different number base.

Message: `Illegal parameter usage`
Diagnosis: The parameters used in a function call are not allowed. In certain cases certain parameters must be distinct - for example in *divide*(.) the first two parameters must refer to distinct *big* variables.
Response: Read the documentation for the function in question.

Message: `Out of space`
Diagnosis: An attempt has been made by a MIRACL function to allocate too much heap memory.
Response: Reduce your memory requirements. Try using a smaller value of *n* in your initial call to *mirsys(n,b)*.

Message: `Even root of a negative number`
Diagnosis: An attempt has been made to find, for example, the square root of a negative number.
Response: Don't do it!

Message: `Raising integer to negative power`
Diagnosis: Self-explanatory.
Response: Don't do it!

Message: `Integer operation attempted on flash number`
Diagnosis: Certain functions should only be used with *big* numbers, and do not make sense for *flash* numbers. Note that this error message is often provoked by memory problems, where for example the memory allocated to a *big* variable is accidentally over-written.
Response: Don't do it!

Message: `Flash overflow`
Diagnosis: This error is provoked by Flash overflow or underflow. The result is outside of the representable dynamic range.
Response: Use bigger *flash* numbers. Analyse your progam carefully for numerical instability.

Message: `Numbers too big`
Diagnosis: The size of *big* or *flash* numbers requested in your call to *mirsys(.)* are simply too big. The length of each *big* and *flash* is encoded into a single computer word. If there is insufficient room for this encoding, this error message occurs.
Response: Build a MIRACL library that uses a bigger "underlying type". If not using Flash arithmetic, build a library without it - this allows much bigger *big* numbers to be used.

Message: `Log of a non-positive number`
Diagnosis: An attempt has been made to calculate the logarithm of a non-positive *flash* number.
Response: Don't do it!

Message: `Flash to double conversion failure`
Diagnosis: An attempt to convert a Flash number to the standard built-in C *double* type has failed, probably because the Flash number is outside of the dynamic range that can be represented as a *double.*
Response: Don't do it!

Message: `I/O buffer overflow`
Diagnosis: An input output operation has failed because the I/O buffer is not big enough.
Response: Allocate a bigger buffer by calling *set_io_buffer_size(.)* after calling *mirsys(.)*.

Message: `MIRACL not initialised - no call to mirsys()`
Diagnosis: Self-explanatory
Response: Don't do it!

Message: `Illegal modulus`
Diagnosis: The modulus specified for use internally for Montgomery reduction, is illegal. Note that this modulus must not be even.
Response: Use an odd positive modulus.

Message: `No modulus defined`
Diagnosis: No modulus has been specified, yet a function which needs it has been called.
Response: Set a modulus for use internally

Message: `Exponent too big`
Diagnosis: An attempt has been made to perform a calculation using a pre-computed table, for an exponent (or multiplier in the case of elliptic curves) bigger than that catered for by the pre-computed table.
Response: Re-compute the table to allow bigger exponents, or use a smaller exponent.

Message: `Number base must be power of 2`
Diagnosis: A small number of functions require that the number base specified in the initial call to *mirsys(.)* is a power of 2.
Response: Use another function, or specify a power-of-2 as the number base in the initial call to *mirsys(.)*

Message: `Specified double-length type isn't`
Diagnosis: MIRACL has determined that the double length type specified in *mirdef.h* is in fact not double length. For example if the underlying type is 32-bits, the double length type should be 64 bits.
Response: Don't do it!

Message: `Specified basis is not irreducible`
Diagnosis: The basis specified for GF(2^m) arithmetic is not irreducible.
Response: Don't do it!

12. The Hardware/Compiler Interface

Hardware/compiler details are specified to MIRACL in this header file *mirdef.h*

For example:-

```
/*
 *   MIRACL compiler/hardware definitions - mirdef.h
 *   This version suitable for use with most 32-bit
 *   computers
 *
 *   Copyright (c) 1988-1999 Shamus Software Ltd.
 */

#define MIRACL_32
#define MR_LITTLE_ENDIAN
                 /* this may need to be changed       */
#define mr_utype int /* the underlying type is usually int *
                  * but see mrmuldv.any              */
#define mr_unsign32 unsigned long
                 /* 32 bit unsigned type             */
#define MR_IBITS  32 /* number of bits in an int */
#define MR_LBITS  32 /* number of bits in a long */

#define MR_FLASH 52  /* delete this definition if integer  *
                  * only version of MIRACL required    *
                  * Number of bits per double mantissa */
#define MAXBASE ((mr_small)1<<(MIRACL-1))
#define MRBITSINCHAR 8
                    /* Number of bits in char type       */

/* #define MR_NOASM   * define this if using C code only   *
                  * Note: mr_dltype MUST be defined    */
/* #define mr_dltype long long
                  * double-length type               */
/*
#define MR_STRIPPED_DOWN * define this to minimize size    *
                  * of library - all error messages    *
                  * lost! USE WITH CARE - see mrcore.c */
```

This file must be edited if porting to a new hardware environment. Assembly language versions of the time-critical routines in *mrmuldv.any* may also have to be written, if not already provided, although in most cases the standard C version *mrmuldv.ccc* can simply be copied to *mrmuldv.c*.

It is best where possible to use the *mirdef.h* file that is generated automatically by the interactive *config.c* program.

Bibliography

[Blake] BLAKE, SEROUSSI, and SMART. Elliptic Curves in Cryptography, London Mathematical Society Lecture Notes Series 265, Cambridge University Press. ISBN 0 521 65374 6, July 1999

[Brassard] BRASSARD, G. Modern Cryptology. Lecture Notes in Computer Science, Vol. 325. Springer-Verlag 1988.

[Brent76] BRENT, R.P. Fast Multiprecision Evaluation of Elementary Functions. J. ACM, 23, 2 (April 1976), 242-251.

[Brent78] BRENT, R.P. A Fortran Multiprecision Arithmetic Package. ACM Trans. Math. Software 4,1 (March 1978), 57-81.

[Brick] BRICKELL, E, et al, Fast Exponentiation with Precomputation, Proc. Eurocrypt 1992, Springer-Verlag 1993.

[Cherry] CHERRY, L. and MORRIS, R. BC - An Arbitrary Precision Desk-Calculator Language. in ULTRIX-32 Supplementary Documents Vol. 1 General Users. Digital Equipment Corporation 1984.

[Comba] COMBA, P.G. Exponentiation Cryptosystems on the IBM PC. IBM Systems Journal, 29,4 (1990), pp 526-538

[CS] CRAMER, R. and SHOUP, V. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack Proc. Crypto 1998, Springer-Verlag 1999

[DSS] Digital Signature Standard, Communications of the ACM, July 1992, Vol. 35 No. 7

[Gruen] GRUENBERGER, F. Computer Recreations. Scientific American, April 1984

[Jurisic] JURISIC, A and MENEZES A.H. Elliptic Curves and Cryptography, Dr. Dobbs Journal, #264, April 1997

[Knuth73] KNUTH, D.E. The Art of Computer Programming, Vol 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1973.

[Knuth81] KNUTH, D.E. The Art of Computer Programming, Vol 2: Seminumerical Algorithms. Addison-Wesley, Reading, Mass., 1981.

[Korn83] KORNERUP, P. and MATULA, D.W. Finite Precision Rational Arithmetic: An Arithmetic Unit. IEEE Trans. Comput., C-32, 4 (April 1983), 378-387.

[Korn85] KORNERUP, P. and MATULA, D.W. Finite Precision Lexicographic Continued Fraction Number Systems. Proc. 7th Sym. on Comp. Arithmetic, IEEE Cat. \#85CH2146-9, 1985, 207-214.

[LimLee] LIM, C.H. and LEE, P.J. A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup. Advances in Cryptology, Crypto '97, Springer-Verlag 1998

[Marsaglia] MARSAGLIA, G.M. and ZAMAN, A. A New Class of Random Number Generators. The Annals of Applied Probability, Vol. 1, 3, 1991, 462-480

[Matula85] MATULA, D.W. and KORNERUP, P. Finite Precision Rational Arithmetic: Slash Number Systems. IEEE Trans. Comput., C-34, 1 (January 1985), 3-18.

[Maurer] MAURER, U.M. and YACOBI, Y. Non-Interactive Public Key Cryptography. Advances in Cryptography, Eurocrypt '91, Springer Verlag, 1992

[Menezes] MENEZES, A.J. Elliptic Curve Public key Crtyptosystems, Kluwer Academic Publishers, 1993

[HAC] Handbook of Applied Cryptography, CRC Press, 2001

[McCurley] McCURLEY, K.S. A Key Distribution System Equivalent to Factoring. J. Cryptology, Vol. 1. No. 2, 1988

[Monty85] MONTGOMERY, P. Modular Multiplication Without Trial Division. Math. Comput., 44, (April 1985), 519-521

[Monty87] MONTGOMERY, P. Speeding the Pollard and Elliptic Curve Methods. Math. Comput., 48, (January 1987), 243-264

[Morrison] MORRISON, M.A. and BRILLHART, J. A Method of Factoring and the Factorization of F7. Math. Comput., 29, 129 (January 1975), 183-205.

[Pollard71] POLLARD, J.M. Fast Fourier Transform in a Finite Field. Math. Comput., 25, 114 (April 1971), 365-374

[Pollard78] POLLARD, J.M. Monte Carlo Methods for Index Computation (*mod p*). Math. Comp. Vol. 32, No. 143, pp 918-924, 1978

[Pomerance] POMERANCE, C. The Quadratic Sieve Factoring Algorithm. In Advances in Cryptology, Lecture Notes in Computer Science, Vol. 209, Springer-Verlag, 1985, 169-182

[Reisel] REISEL, H. Prime Numbers and Computer methods for Factorisation. Birkhauser. 1987

[Richter] RICHTER, J. Advanced Windows. Microsoft Press.

[RSA] RIVEST, R., SHAMIR, A. and ADLEMAN, L. A Method for obtaining Digital Signatures and Public-Key Cryptosystems. Comm. ACM, 21,2 (February 1978), 120-126.

[Rubin] RUBIN, P. Personal Communication

[Sch] SCHOOF, R. Elliptic Curves Over Finite Fields and the Computation of Square Roots mod *p*. Math. Comp. Vol. 44, No. 170. April 1985, pp 483-494

[Scott89a] SCOTT, M.P.J. Fast rounding in multiprecision floating-slash arithmetic. IEEE Transactions on Computers, July 1989, 1049-1052.

[Scott89b] SCOTT, M.P.J. On Using Full Integer Precision in C. Dublin City University Working Paper CA 0589, 1989.

[Scott89c] SCOTT, M.P.J. Factoring Large Integers on Small Computers. National Institute for Higher Education Working Paper CA 0189, 1989

[Scott92] SCOTT, M.P.J. and SHAFA'AMRY, M. Implementing an Identity-based Key Exchange algorithm. Available from ftp.computing.dcu.ie /pub/crypto/ID-based_key_exchange.ps

[Scott93] SCOTT, M.P.J. Novel Chaining Methods for Block Ciphers, Dublin City University, School of Computer Applications Working Paper CA-1993

[Scott96] SCOTT, M.P.J. Comparison of methods for modular multiplication on 32-bit Intel 80x86 processors. Available from ftp.computing.dcu.ie /pub/crypto/timings.ps

[Shoup] SHOUP, V. A New Polynomial Factorisation Algorithm and Its Implementation. Jl. Symbolic Computation, 1996

[Stinson] STINSON, D.R. Cryptography, Theory and practice. CRC Press, 1995

[Silverman] SILVERMAN, R.D. The Multiple Polynomial Quadratic Sieve, Math. Comp. 48, 177, (January 1987), 329-339

[Walmsley] WALMSLEY, M., Multi-Threaded Programming in C++. Springer-Verlag 1999.

[WeiDai] DAI , W. Personal Communication