Brock University

Department of Computer Science

rmath User's and Technical Guide

Michael Letourneau
Technical Report # CS-02-19
August 2002

To obtain the software, please visit http://www.cosc.brocku.ca/rmath.

Brock University
Department of Computer Science
St. Catharines, Ontario
Canada L2S 3A1
www.cosc.brocku.ca

rmath User's and Technical Guide

Michael Letourneau
Department of Computer Science
Brock University

Front Matter

This work was originally prepared to satisfy the requirements of the course COSC 4F90, offered by the Department of Computer Science, Brock University, during the 2001/02 academic year. The project undertaken in this course was entitled “Methods for Finding Exact Bounds on Non-Linear Error Correcting Codes” and was undertaken under the supervision of Dr. Sheridan Houghten

Copyright, *et c.*

This manual, the product that it is intended to accompany, and all other related materials, are presented here WITHOUT WARRANTY. Any and all use of these products is at the risk of the end user.

This manual is copyright 2002 by Michael Letourneau.

This manual may be freely distributed by itself, or as part of the *rmath* software package. It may not be modified or altered in any fashion without the express permission of the author.

The production of the *rmath* system was, in part, funded by a research grant from the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

1 - Introduction to *rmath*

rmath is a system for representing vectors of elements of finite fields. It supports arbitrary length vectors, addition of vectors, multiplication of vectors, determining the weight of vectors, determining the distance between vectors, and more. It is designed as a library of C code, and is intended to be platform independent[1]. Vectors are stored in a bit-level representation, and bit-level versions of the various operations are provided for highly efficient operation on vectors of elements from certain fields. *rmath* provides means for users to specify any finite field of up to 2^n elements, where n is the word size of the computer.

rmath was designed for use in combinatorial searches, especially for error-correcting codes. Auxiliary routines have been included for expanding linear codes, verifying minimum weight and distance, and for performing simple equivalence testing on linear codes.

rmath has been designed to be fully modifiable. Such modifications permit *rmath* to be extended to include bit-level operations for any field representable with the system. It has built-in representations for various Galois fields, GF(2), GF(3), GF(4), and GF(5).

[1]To date, *rmath* has been tested on only a limited set of platforms. No significant problems have arisen on any of those platforms.

1.1 - Using *rmath*

rmath is written as a C library. Thus, it will need to be compiled into your C program. Unlike most traditional libraries, it requires special compilation, depending on the nature of the application. Here is what must be done in order to use it:

1.) Determine the requirements for representation,
2.) Specify those requirements in code,
3.) Provide an *rmath* initialization point
4.) Compile *rmath* and your application code.

The following sections will discuss each of those steps.

1.1.1 - Determining and Specifying requirements

All of the requirements for *rmath* are specified by the use of preprocessor macros. This section will describe how to define those macros for your application. Those with applications which compile with multiple files that need to reference *rmath* features should fully read the section Compiling *rmath* in order to determine where to locate these macros. Those with only one file which requires *rmath* should still read that section for information, but may go ahead and place the following macros at a location in that file prior to the "#include" line for *rmath*.

rmath stores each vector in at least one unsigned integer word. The word size for the machine must be specified by the user. At present, only word sizes of 32 and 64 bits are accepted. A 32-bit word will assume that the C type **int** has 32 bits. (If it doesn't it will be necessary to modify the definition of the *rmath* type **word** - see Modifying *rmath* for more information.) If a 64-bit word is specified, then it will be necessary to specify to *rmath* the C data type for your machine that is 64 bits, integral, and can be given as "unsigned". These items may be specified as:

```
#define WORDLEN n        /*   n is word size: 32, 64 */

#define LONG64 m         /*   m is 64-bit integer type name.
                              Do not include 'unsigned' in
                              the name. */
```

If **int** on your machine is 64-bit, then it would be wise to specify it as the 64-bit word type. If a word size of 32 bits is specified, *rmath* will only use 32 bits of an **int**, even if it is 64 bits (or bigger.)

It is now necessary to inform *rmath* of the number of elements in the field that you wish to work on. To give this value:

```
#define FIELDSIZE s      /*   s is the number of elements in
                                the field */
```

At present, *rmath* works on fields of up to five elements. It may be modified to accept larger field sizes with little effort.

It is now necessary to define the mode of operations that *rmath* will use when operating on vectors. For the built in functions, there are two modes built in: **bitwise** and **procedural**. Bitwise operations work just as their name implies, on the individual bits of the vector. These operations are composed from the standard bitwise C language operators. Procedural operations work by referencing a value in a look-up table. They are slower than bitwise operations, but provide guaranteed results on systems where the prebuilt bitwise operations may not work correctly. They also allow for easy redefinition of the results of the operations; replacing the table changes the results of the operations. If the application will be redefining the field values, then it will be necessary to consult the section on customization for the correct value for this setting. If the built-in field representations are being used, then it will generally suffice to use the bitwise operations. If these fail to give correct results on your system, then it will be necessary to use the procedural operations. (The results of a program can be verified by compiling and running it once each with the bitwise and procedural operations. If the results are the same, then the bitwise operations have given the correct results.) Specifying the operation modes would appear as:

```
#define ADDITION_MODE n       //  In each case, n is either
#define MULTIPLICATION_MODE n //  BITWISE or PROCEDURAL
```

The last value which needs to be specified is the number of words used to hold each vector. If a vector requires more than one word for storage, *rmath* will automatically create a structure type to hold the required number of words in an array. It will also modify the operations to work on the structure type. At present, it is necessary to specify the number of words required to hold the longest vector required by your application. (A future revision will allow the direct specification of the length.) To determine the number of words required, use the following formula:

$$words\ required = \left\lceil \frac{vector\ length \times bits\ per\ place}{bits\ per\ word} \right\rceil$$

The number of bits per place depends on the size of the field being represented. This value will generally correspond to the following table:

Size of Field	Bits per place
2	1
3	2
4	2
5	3

Larger fields will require extension of *rmath*. Information on calculating the correct value can be found in the Customizing *rmath* section. Once the number of words has been calculated, it can be given by specifying:

```
#define NUMWORDS n        /*   n is the number of words */
```

There are other values for *rmath* which can be specified through other macros. These values are described in other parts of this document.

1.1.2 - Providing an *rmath* Initialization Point

rmath uses several auxiliary tables to support certain operations. These tables must be initialized. In order to do so, it is necessary to call the function:

```
setupRMath();
```

once within the application code. This call should be made before any calls to *rmath* functions are made. Typically, this would be made during program initialization.

1.1.3 - Compiling *rmath*

Because *rmath* requires knowledge of the size of the field being compiled, it is necessary to specify the information previously discussed before the *rmath* code is compiled. Since *rmath* is contained in two files, one header file and one source file, it is necessary to `#include` the header in any code that requires the ability to work with the field vectors. The source file needs to be compiled only once, but it must be compiled with the same specifications as the program uses. There are two simple approaches for this, depending on the number of files used to build your application:

If only one source file needs to use *rmath* and its data types, then one should place the macros at the top of that source file, then the lines `#include "rmath.h"` and `#include`

`"rmath.c"`, in that order. This will ensure that the basic *rmath* types will be established in the header, and the function library will have the correct values when it is compiled with your source file.

If more than one source file needs to use *rmath* or its data types, then it is essential to ensure that each file has access to the information from the header, which is where the data types are defined and the functions prototyped. The *rmath* functions themselves can be externally linked later in compilation. To accomplish this easily:

1.) Create a source file which will hold your *rmath* setup value macros. This file will be called the **setup header**.
2.) Place the macros discussed previously in the setup header.
3.) Place the line `#include "rmath.h"` at the end of the setup header.
4.) `#include` the setup header with **every** file requiring *rmath*.
5.) Create a second source file. This file will be called the **library stub**.
6.) In the stub file, `#include` the setup header.
7.) Also `#include "rmath.c"` in the stub file.
8.) The stub file need only be compiled once and then linked into the application.

The setup header ensures that the same setup values are used by all source files. The stub file ensures that the function code is only compiled once, and prevents any problems with duplicate definition errors that can result from repeated compilation of the same code.

1.2 - *rmath* Basics

1.2.1 - Data Types

rmath provides access to the vectors of field elements through a data type called **field**. This data type may be a renaming of a basic type, or it may be a more complicated structure type. *rmath* handles these details transparently. This does necessitate, however, use of the `sizeof()` operator to provide the size of the data type whenever it is necessary. For, example, here are some variable declarations:

```
int a, b, c;              /* Integer variables */
float i, j;               /* Floating point variables */
field x, y, z;            /* Vectors of field elements */
field *f;                 /* Pointer to a 'field'*/
```

It is also possible to allocate storage using the **word** data type, which underlies the field data type. This will be necessary when working with the storage routines that pack the field vectors. This data type will always be a base data type for your system (at present, either **int** or the system-specific type specified for 64-bit computation). For example:

```
word k, l, m[20];             /*   'word's, and an array of
                                   'word's.*/
```

1.2.2 - Initializing *rmath*

rmath uses a set of auxiliary tables to perform operations such as determining the weight of vectors and taking linear combinations thereof. These tables must be initialized during the operation of the program. To perform this initialization, include the following line in your code:

```
setupRMath();
```

Ideally, this call should be made at the start of the **main()** function. Practically, it can be made anywhere in the code that occurs before any other *rmath* code is called. It needs to be called once.

1.2.3 - Setting Values

Now that there is storage available, the next logical step is to put something into it. The simplest operation is to clear the vector by setting all the elements to 0. This can be accomplished by calling ZEROV(*v*), with the name of the field to be cleared as *v*.

```
. . .
ZEROV(x);                 /* 'x' is now 00000...0 */
. . .
```

Several methods exist to set the values in vectors to values other than 0s. One is to set the individual values of the field positions. To do this, use the SETPOSN(*a*, *i*, *v*) procedure, with *a* as the field variable to be set, *i* as the position within the field to be set (see note below), and *v* as the value of the field, from 0 .. *q*-1, where *q* is the size of the field. For example:

```
. . .
SETPOSN(x, 3, 2)          /* 'x' is now 000...002000 */
. . .
```

Note: In the examples, a vector of *p* positions is printed with the (*p* - 1)st position first, followed by the (*p*-2)nd position, through to the 0th position. This is the standard form used by the printing functions which will be discussed later.

It is often desirable to read in a textual string representing values to be stored in a field. Two functions are provided to perform these functions: *sToField* and *sToField_Sized*. The latter is a more functional version of the former. Both are discussed in the *rmath* Functions List section; an example of using *sToField_Sized*:

```
. . .
char s[11] = "1020101221";
. . .
x = sToField_Sized(a);
. . .
```

In this example, the field vector *x* receives the values given by the string *a*. The values are stored in such a manner that they will be output in the same fashion. Thus, the rightmost character of the string is stored in position 0, the next-rightmost will be stored in position 1, ... There are a few issues regarding the number of characters presented in the string and the storage available in the vector. Consult the functions list for more information on this.

1.2.4 - Manipulating and Analyzing Vectors

There are two manipulations provided which work with the Galois field representations. These are addition and multiplication. These operations work independently on each field position, and have no carry-overs. They provide their results as evaluatable C expressions, and can be 'strung' together. For example, in GF(4):

```
field a, b, c, d;
char s1[11] = "0123012301";
char s2[11] = "1200321201";
char s3[11] = "0103320011";

a = sToField_Sized(s1);
b = sToField_Sized(s2);
```

```
c = sToField_Sized(s3);

d = ADD(a, b);      /*   d is 0123012301 + 1200321201 =
                         1323333101 */

d = MULT(b, c);     /*   d is 1200321201 * 0103320011 =
                         0200230001 */

d = MULT(ADD(a, b), MULT(b, c));
                    /*   d is 1323333101 * 0200230001 =
                         0100120001 */
```

The **ADD** and **MULT** operations evaluate using either C operators (when using bitwise operations) or array references (when using procedural operations). In either case, they provide a value that can be directly assigned to a field variable, or used with another operator requiring field operators.

The other significant operation is the **LEX_NEXT** operation. This operation, when given a field *f* and a width *w*, will evaluate to the vector with *w* useful positions which is lexicographically next after *f* . If there is no vector of that size, then the all-zero vector is returned. For example, in GF(4):

```
field a, b, c;
char s1[11] = “0130320312";
char s2[11] = “3333333332";

a = sToField_Sized(s1);
b = sToField_Sized(s2);

c = LEX_NEXT(a, 10);     /*   c is now 0130320313 */
c = LEX_NEXT(a, 10);     /*   c is now 0130320320 */
c = LEX_NEXT(a, 10);     /*   c is now 0130320321 */

c = LEX_NEXT(b, 10);     /*   c is now 3333333333 */
c = LEX_NEXT(b, 10);     /*   c is now 0000000000 */
```

Operations to take the weight of a vector and the distance between any two vectors are also provided. WEIGHT(*a*) evaluates to the weight of the field *a*. DISTANCE(*a*, *b*) evaluates to the distance between the fields *a* and *b*. Both can be used anywhere an integer expression is needed. An example of the two operations in GF(3):

```
field a, b;

char s1[11] = “0120120120";
char s2[11] = “2102102102";
```

```
int w, d;

a = sToField_Sized(s1);
b = sToField_Sized(s2);

w = WEIGHT(a);           /*   w is now 6 */
w = WEIGHT(b);           /*   w is now 7 */
w = WEIGHT(ADD(a, b);    /*   w is now WEIGHT(0120120120 +
                              2102102102) =
                              WEIGHT(0000000000)= 0 */

d = DISTANCE(a, b);      /*   d is now DISTANCE(0120120120,
                              2102102102) = 7 */
```

1.2.5 - Vector Output

There are two primary methods for the output of vectors: formatted and packed. Formatted output produces strings which represent the values in the vectors. Packed output takes a number of vectors and stores them in reduced space. Formatted output is suitable for human reading and interchange with programs compiled on other systems. Packed output is suitable for storing large quantities of codes for use by programs on the same machine.

There are two functions that provide formatted output: *fieldToS* and *fieldToS_w*. *fieldToS_w* will be described here. *fieldToS* is its less-functional parent function; it is described in the *rmath* Functions List. Taking three parameters, *fieldToS_w*(*p*, *s*, *w*) will place a null-terminated string containing *w* characters (followed by a null character) into *s*, which *rmath* assumes has memory allocated for those *w* + 1 characters. Those *w* characters represent the values in the positions 0 .. *w* - 1 in the vector *p*. The remaining positions are ignored. The format of the string is the same format as the string input functions uses. Thus, a field *a* output through this function and then input into a field *b* through a string input function like *sToField_Sized* will meet the condition *a* == *b*.

```
field a;
char s1[11] = “0122231012", s2[11];

a = sToField_Sized(s1);

fieldToS_w(a, s2, 10);   /*   s1 and s2 are now identical
                              strings. */
```

Packed output is intended for use with blocks of vectors. It works on the principle of eliminating the unused positions from the vectors. To understand its use, consider first the following example code:

```
field vectors[15];
word *packWords;
FILE *fp;

. . .
/*   code which puts values into ‘vectors’. Assume that the
     first 10 positions of the vectors are used. Also, ‘fp’
     is opened on a file. */
. . .

packWords = malloc(PACKED_FIELD_ARRAY_SIZE(15, 10),
                                              sizeof(word));

packFieldArray(vectors, packWords, 15, 10);

/*   ‘packWords’ now contains all of the values of the first
     10 places from the 15 vectors in ‘vectors’. */

fwrite(packWords, sizeof(word),
                    PACKED_FIELD_ARRAY_SIZE(15, 10), fp);
. . .
```

This code contains an array of vectors called *vectors* and an array of words called *packWords*. It assumes that only the first 10 positions in each vector are significant. The macro PACKED_FIELD_ARRAY_SIZE(*qty*, *width*) is used to return the number of words required to have 15 vectors of 10 significant places packed into them. *packWords* has a block of memory of that size allocated to it. packFieldArray(*source*, *dest*, *qty*, *width*) packs the significant places from *source* into *dest*, with *qty* indicating how many vectors from *source* will have their first *qty* values packed into *dest*. At the end of the code section, *packWords* has all the information from the first 10 positions in the 15 vectors from *vectors*. This information is then be written out to a file in a single *fwrite*() call.

packFieldArray(...) works by eliminating the vector positions deemed unnecessary from each vector. This elimination is done by simply copying the necessary bits from each of the source vectors in turn into enough word-sized storage. Of course, this form of representation is useless unless there is a way to ‘unpack’ the information. To do this, use the function *unpackFieldArray*(*dest*, *source*, *size*, *width*), where *dest* points to enough memory to store *size* vectors, and where *source* contains the packed form of the first *width* positions of *size* vectors. The use of this function is best illustrated by continuing the previous code snippet:

```
. . .
/* ‘fp’ is reset to point to the block previously written.*/

fread(packWords, sizeof(word),
                    PACKED_FIELD_ARRAY_SIZE(15, 10), fp);
```

```
unpackFieldArray(vectors, packWords, 15, 10);

/*   The first 10 symbol positions in the first 15 places of
     'vectors' are restored to the values stored during the
     call to packFieldArray(...). The remaining symbol
     positions are assigned 0 values. */
```

The obvious advantage of this storage method is the reduction in space. Since operations on the bits of an individual word are the smallest provided in C, no more than one vector is stored in a word during normal operation. This can leave many symbol positions unused. For example, if the word size was 64 bits in our previous example, and only the first 10 positions of each vector were significant, then at 2 bits per position, there were (64 - (2 × 10) =) 44 unused bits in each word. This use of only 1/4 of the storage is acceptable when dealing with actual computation, but it is far too high for pure storage purposes. By packing the vectors in an array of words, the whole array can be moved by using the *fread*() and *fwrite*() commands. This also provides a degree of speedup in the code itself, since there is no need to do formatted-I/O.

It should be noted that the operation of the vector packing commands is dependent on the settings with which *rmath* was compiled. If settings like word size, field size, or operating platform change, there is no guarantee that the stored data formats will be compatible. This is especially true for platform changes, since the byte-order and bit-significance of the data can vary between platforms. For moving data between platforms, use the string input/output functions, whose formatting is not affected by such issues. Similarly, the data stored from a 32-bit word size application cannot be read by a 64-bit word size application, or vice versa, even if they are running on the same machine and OS.

1.3 - *rmath* User Functions and Macros Lists

The following functions are provided by the *rmath* system. This list comprises all of the functions which should be called by end-users. Other functions exist, but are used by the system for internal computations. Many of the functions presented here are aliases to other functions. The particular functions that they alias depend on the size and number of field positions of the field being employed.

Note: Some of these functions are implemented as macros. Consequently, in some cases (i.e. CLEAR_FIELD(field)), parameters passed to them will CAN BE MODIFIED even though they appear to be passed-by-value.)

1.3.1 - Basic Functions

Name	Description
field ADD(field *a*, field *b*)	Evaluates to the result of adding *a* to *b*.
field MULT (field *a*, field *b*)	Evaluates to the result of multiplying *a* by *b*.
short int WEIGHT (field *a*)	Evaluates to the number of non-zero positions in *a*.
short int DISTANCE (field *a*, field *b*)	Evaluates to the number of field positions in which *a* and *b* differ.
void CLEAR_FIELD (field *a*)	Sets each of the field positions in *a* to be zero.
word EXTRACT (field *a*, int *i*)	Evaluates to the value in the *i*th position of *a* as a value between 0 and *q*, for GF(*q*).
void SETPOSN (field *a*, int *i*, int *v*)	Sets the value of the *i*th position in *a* to be the value *v*.
[conditional] EQUAL (field *a*, field *b*)	Evaluates as a C conditional. Returns true (1) if field *a* and field *b* are the same field (i.e. if adding them together will produce the all-zero word.), and false (0) otherwise.

1.3.2 - Ordering Functions:

Name	Description
field LEX_NEXT(field *a*, int *w*)	Returns the field which is lexicographically directly after *a*, unless it is impossible to represent that field without having a non-zero character in a field position > *w*, in which case it returns the all-zero field. Practically, *w* constrains the number of field positions used. Thus, if *w* = 3, it will generate the following sequence of fields if given 000 in GF(2): 001, 010, 011, 100, 101, 110, 111, 000. If *w* > NUMWORDS * PLACES_PER_WORD, this function will cause the program to terminate with an error message indicating this condition.
[conditional] LEX_LESS (field *a*, field *b*)	Evaluates as a C conditional. Returns true (1) if *a* occurs earlier in the lexicographic ordering than *b*, and false (0) otherwise.
[conditional] LEX_GREATER (field *a*, field *b*)	Evaluates as a C conditional. Returns true (1) if *a* occurs later in the lexicographic ordering than *b*, and false (0) otherwise.

1.3.3 - I/O Functions

Note: The string input and output functions are intended to be complementary. Thus, *a* == *b* should be true for field *b* produced from the string produced from field *a*.

Note: Field output in a binary form can be accomplished using *fwrite*() calls. See the function *packFieldArray(. . .)* for information on saving space in binary output.

Name	Description
field sToField_Sized (char *s)	Returns the field represented by the string *s*. *s* should only consist of the appropriate input characters (ASCII digits from 0 .. q-1 for GF(q), unless otherwise adjusted) and be terminated by a null character. If the string does not have enough characters to fill the available positions, it will pad the left end of the string, and thus the high-index field positions, with zeros. If the string has more characters than available field positions, then the function will use as many of the rightmost characters from the string as is necessary.
void fieldToS (field *p*, char *s)	Sets the characters of *s* to represent the values of *p*, according to the appropriate output characters (ASCII digits from 0 .. q-1 for GF(q), unless otherwise adjusted) and terminates them with a null character. As many characters are set as *p* has positions to represent. (The macro FIELD_STRING_SPACE evaluates to amount of memory which must be allocated (via malloc(...)) for *s*.) Presumes that *s* has enough space allocated to hold all of the necessary characters.

void fieldToS_w (field *p*, char **s*, int *w*)	Sets the first *w* characters of *s* to represent the rightmost *w* positions of *p* according to the appropriate output characters (ASCII digits from 0 .. q-1 for GF(q), unless otherwise adjusted) and places a null character in the *w* + 1st position. Presumes that *s* has allocated space for at least *w* + 1 characters.

1.3.4 - Field Packing Functions

Name	Description
packFieldArray (field **source*, word **dest*, int *size*, int *width*)	This function will store the first *width* field elements from each of the first *size* fields stored at *source* to *dest*. It returns 0 if the packing process succeeded, and a negative number if the process failed.
unpackFieldArray (field **dest*, word **source*, int *size*, int *width*)	This function will restore the first *width* field elements to each of the first *size* fields pointed to by *dest* from *source*, and store zero values in all the other positions. This function is the counterpart to packFieldArray (...). It returns -1 if the unpacking process fails and 0 otherwise.
int PACKED_FIELD_ARRAY_SIZE (int *size*, int *width*)	Evaluates to the number of 'word'-sized variables required to store the packing of *size* * *width* field elements. This function is intended to be used with malloc(...) when allocating memory for packing fields via packFieldArray(...).

1.3.5 - Linear Combination Functions

Note: All of these functions assume that suitable quantities of memory are allocated for storing the combinations at their destinations.

Name	Description
void COMBINE (field *a*, field *b*, field **mloc*)	Produces linear combinations of *a* and *b* and stores them starting at the location pointed to by *mloc*. The actual combinations are specified by the value of the macro COMBINATION_TABLE_TYPE. The number of combinations to be generated will be given by the macro NUM_POSSIBLE_2_COMBINATIONS.
int generateCombinations_Block (field **base*, field **dest*, int *qty*)	Produces all linear q^{qty} - 1 non-zero linear combinations of the *qty* vectors at base and stores them at dest. Returns the number of combinations produced.
int generateCombinations_Inc (field *val*, field **dest*, int *qty*)	Produces all linear combinations of *val* and each of the first *qty* fields stored at *dest* and stores them in the positions in *dest* directly following the first *qty* fields. Returns the new number of fields stored at *dest* (*qty* * q + q - 1, for GF(q).)

int generateCombinations_Inc_Weight (field *val*, field **dest*, int *qty*, int *weight*)	Produces linear combinations of *val* and each of the first *qty* fields stored at *dest* and stores them in the positions in *dest* directly following the first *qty* fields. If any combination is generated that has a weight less than *weight*, then the function will stop generating linear combinations and return the number of fields stored at *dest* (not including the underweight combination) * -1. If no underweight combination is generated, then the function will return the number of fields stored at *dest*. (This function is a 'short-circuited' form of generateCombinations_Inc(...). It is intended for determining if a linear code meets its minimum distance requirement. If any linear combination of the vectors does not meet the criterion, then the whole code can be rejected. Thus, there is no reason to continue the combination generation.)

1.3.6 - 'Description' Functions

Note: These functions are intended to provide a form of isomorph checking for field vectors. describe(...) provides a certificate for a given field vector, and sameDescription(...) can be used to compare those certificates.

Name	Description
description describe (field *a*)	This function will determine the quantities of each symbol in *a* and return those values in a 'description' structure.
int sameDescription (description *a*, description *b*)	This function will return 1 if the field described by *a* can be converted to *b* by means of a multiplication of all field elements by a constant non-zero factor (and vice versa), and 0 otherwise.

1.3.7 - Various (and Sundry) Other Functions

Name	Description
void setupRMath ()	Serves as a single activation point for all of the various initialization functions. This needs to be called only once in the operation of the program. It should be called prior to using any *rmath* operations.

1.3.8 - Macros - User '#defined' and required

Note: All of these macros should be defined prior to inclusion of "rmath.h".

Name	Possible Values	Description
WORDLEN	32, 64	Defines the size in bits of the datatype defining the 'word' datatype, with which *rmath* defines the 'field' datatype. If set to 64, the macro LONG64 must be defined as well (see below).
LONG64	{valid 64-bit type name}	Defines the 64-bit integral datatype used to define the 'word' datatype. Necessary to overcome the lack of a fully defined standard for 64-bit datatypes in C. The type must be specifiable as 'unsigned'. Example: On SGI equipment running 64-bit MIPS processors and IRIX 6+, the type 'long long int' is a suitable value.
FIELDSIZE	2, 3, 4, 5	Defines the number of symbols to be represented in the field. Is *q* for GF(*q*).

NUMWORDS	1, 2, 3, . . .	Defines the number of words required to compose a field. This value depends on the size of the datatype and the field being used. See *Introduction to* rmath - *Defining and Specifying Requirements* for more information.
ADDITION_MODE	BITWISE, PROCEDURAL	Defines how *rmath* performs addition. BITWISE specifies the use of inlined logical expressions operating on whole words at a time. PROCEDURAL specifies the use of lookup tables operating on single field positions at a time.
MULTIPLICATION_MODE	BITWISE, PROCEDURAL	Defines how *rmath* performs multiplication. See the macro ADDITION_MODE's entry for an explanation of the values.

COMBINATION_TABLE_TYPE	FULL, NON_ZERO, NON_ZERO_ NON_IDENTITY, OTHER	Specifies the form of results from the COMBINE(...) operation. FULL indicates that all possible combinations, including the zero and both identity combinations are generated. NON_ZERO indicates that all possible combinations except the zero combination are to be generated. NON_ZERO_NON_IDENTITY indicates that all combinations other than the zero and identity combinations are to be generated. OTHER indicates that a user specified table is to be used. (See *Modifying* rmath - *Combination Generation* for more information.)

1.3.9 - Macros - System '#defined' and Useful

FIELD_STRING_SPACE	Evaluates to an integer defining the number of characters necessary to write out the shortest string necessary to hold the string representation of a full field variable (including zero-positions not set by the user's program.)

2.0 - Modifying *rmath*

rmath was designed to support different levels of customization and modification. Modification is only necessary, however, if you wish to:

Work with non-Galois fields
Work with a Galois Field with more than 5 elements
Adjust the string representation used
Specify a particular order for the generation of linear combinations of vectors
Do something different or bizarre with *rmath* which is not a different or bizarre operation supported by *rmath*.

If it is your intention to do any of the above, you should read this section. Details are given in it describing how *rmath* works and how it can be modified safely.

2.1 - Philosophy of Representation and Operation

This system was created with a mandate to:

1.) Perform operations on GF(2), GF(3), GF(4), GF(5),
2.) Efficiently (in terms of both time and space.),
3.) In a reusable
4.) And extensible manner,
5.) In a platform-independent way.

To handle the first four constraints, a 'packed bitfield' representation was chosen. Bitwise operations are not normally recommended for platform-independence, but this primarily applies to shifts, since the significance of the bits can change. It may be necessary to extend this system to handle differences in the underlying machine representations. Since the first four items given above were the most timely during construction of the original system, they became the focus of design.

Each value in its given field is mapped onto at least one scalar variable. The scalar variable used depends on the system configuration. If the system supports 64-bit operations, then an unsigned 64-bit integer will be used. Otherwise, a 32-bit unsigned integer will be used. These scalar variable is 'typedef'ed into a type called 'word'.

In normal operation, each word is broken down into individual sections of *n* bits, where *n* is the minimum number of bits required to represent the range of values in the field.

Field	Bits Required
GF(2)	1
GF(3)	2
GF(4)	2
GF(5)	3

These bit sections are aligned with the 'right' end of the word. These sections are referred to as 'unit-positions', since they represent a single unit in a given position.

For example, to represent the value 41023 in GF(5), each position would be converted to binary and stored as follows:

4 = 100
0 = 000
1 = 001
2 = 010
3 = 011

Presuming that the remaining values to the left of these are to be all zeros, the storage of the value would be:

In a 32 bit word:
00000000000000000100000001010011

In a 64 bit word:
0000000000000000000000000000000000000000000000000100000001010011

Depending on the value of the macro NUMWORDS, a new datatype, 'field', will be defined.

If NUMWORDS = 1, then 'field' will be 'typedef'ed as a 'word'.

If NUMWORDS > 1, then 'field' will be 'typedef'ed as a structure with one element, an array 'words', of size NUMWORDS.

When 'field' is a structure, the values will be distributed across the array 'words' by storing a fixed number of representation places in each word. This is equivalent to breaking the given value into NUMWORDS individual values by partitioning the digits so that each word is used fully. (Refer to the 'PLACES_PER_WORD' tables in the 'Using rmath' section for the storage capacities of the 'word' types for the different fields.)

Operations on fields (like multiplication and addition) are performed on individual unit-positions. Since there are no carries or other 'inter-position' modifications, the operations can be defined (relatively) easily using bitwise operations. The bitwise operations given in this file are designed to operate on individual words. (It is impossible to define a bitwise operation a structure.) For each field size, there are two operations, ADDWORD(a, b) and MULTWORD(a, b), which define addition and multiplication in that field using only bitwise operations. Each one operates on a single 'word', and gives its result as a single 'word'.

For 'fields' which are just a single 'word', the bitwise operations ADD(a, b) and MULT(a, b) are '#defined' directly to the ADDWORD(a, b) and MULTWORD(a, b).

For 'fields' which are structures with an array of 'word's, the ADD(a, b) and MULT(a, b) operations are defined through functions, 'structAdd(a, b)' and 'structMult(a, b)', where 'a' and 'b' are 'fields'. These functions apply ADDWORD and MULTWORD, respectively, to the individual members of the 'words' array of 'a' and 'b'. They work by ADDWORDing/MULTWORDing the 'i'th 'word's of 'a' and 'b' together and placing the result in the 'i'th word of the result value.

The other major operation on fields which is provided here is WEIGH(a). This operation will return a short integer value which is the number of non-zero values in the field 'a'. Like its additive and multiplicative counterparts, this operation has definitions for working on both individual 'word's (WEIGHWORD(a)) and on 'field's (WEIGHT(a)). The 'word'-based operation

is used directly whenever 'field' is defined as simply a word, and a function 'structWeigh(a, b)' is used to return sum the weights of the words in a structural 'field'.

The WEIGHWORD(a) operation makes use of an auxiliary table to assist in the weight determination operation. This table 'weights[]', has elements of type 'short' and size 2^16. This table is loaded with the weights for each possible representation in a 16-bit word. When WEIGHWORD(a) is called, a summation is made of the weights of each section of the word 'a' that is no longer than 16 bits and such that no unit-position spans the boundary between two of these sections. (In practice, this means that anywhere from 2 sections (in the case of a 32-bit GF(2) word) to 5 sections (in the case of a 64-bit GF(5) word) will be summed together.). A function, 'establishWeightTable()', is used to build this weight table. This function needs to be called only once, and that should be during program initialization. (A call to this function has since been incorporated into the 'setupRMath()' function.)

The definitions of the operations and types are designed to be transparent to the casual user. By referencing only the basic operations, the construction of the field and the underlying datatype needn't be a worry of the end user.

2.2 - Customizing rmath Operations

Since the general purpose of this library was to provide access to a non-standard mathematical process, it required the design of a system which could work with data in a non-standard way. Rather than constrain this system to working with just with the defined operations, it was felt that some access to the underlying system should be provided, so that the flexibility of the system could be used by a potentially larger population.

The customization options will be described in a fashion proceeding generally from the easiest/least-'dangerous' options to the most-confusing/most-'dangerous'. Feel free to experiment with any that you feel comfortable with.

2.2.1 - Procedural Operations

rmath provides two different methods for performing addition and multiplication on fields: bitwise and procedural.

The recommended method is to use the bitwise operations. These have been composed of nothing but elementary bitwise operations, and have been found to be from 4 to 40 times faster than their procedural counterparts. Their major drawback is that some platforms may have bit/byte/word alignments and orientations which are incompatible with the bitwise operations as they are defined here.

In these cases, the procedural operations can be used. They have been designed so that they *SHOULD* work correctly on all platforms.

These procedural operations can be used to verify the operation of the bitwise operations. By comparing the results of the same program run once with the bitwise operations and once with the procedural operations, any discrepancies in the bitwise operations can be discovered. If there is no difference between the two results (beyond the time required to produce them), then the bitwise operations should work correctly. If discrepancies arise, then the results of the procedural operations should be taken as correct.

To use the procedural operations, define the following macros as necessary:

```
#define ADDITION_MODE PROCEDURAL
#define MULTIPLICATION_MODE PROCEDURAL
```

To use the bitwise operations, define the following macros as necessary:

```
#define ADDITION_MODE BITWISE
#define MULTIPLICATION_MODE BITWISE
```

You can 'Mix-and-Match' these operation modes by having one bitwise and one procedural.

2.2.2 - String Interface Options

For the string interface, the normal mode of operation is to transform characters to and from base-10 integer values. For going from ASCII to binary, this is performed by simply subtracting the value of '0' from the read character value (which is thanks to the construction of the ASCII table and the integer-esque definition of characters), the result is a binary value from 0x0 to 0x9 which represents the original value and which can be placed in a unit-position within a 'word'. The reverse of this operation can performed by adding the value of '0' to the binary representations to obtain the printable ASCII value.

Should it be desirable to use other ASCII characters for these operations, the mappings can be altered by defining the following macros PRIOR TO THE *FIRST INCLUSION* of 'rmath.h':

BASE_READ_CHAR - With a value to be subtracted from read ASCII characters.

BASE_PRINT_CHAR - With a value to be added to values to be printed as ASCII characters.

(Generally, these should be the same.)

2.2.3 - Combination Generation

The COMBINE(a, b, *mloc) operation generates all possible combinations of the field values 'a' and 'b' and stores in the memory at 'mloc'. This generation is performed by multiplying 'a' and 'b' by fields containing the same value repeated through all the positions, and then adding the results. This can be best understood through an illustrative example:

For GF(3) (9 possible combinations), a = 1022, b = 2121

Multiplier a	Multiplier b	a * Mult.	b * Mult.	Sum Result
0000	0000	0000	0000	0000
0000	1111	0000	2121	2121
0000	2222	0000	1212	1212
1111	0000	1022	0000	1022
1111	1111	1022	2121	0210
1111	2222	1022	1212	2221
2222	0000	2011	0000	2011
2222	1111	2011	2121	1102
2222	2222	2011	1212	0220

The process of producing these combinations is done by multiplying 'a' and 'b', by constant factors contained in auxiliary tables. The number of combinations made is defined by the macro NUM_POSSIBLE_2_COMBINATIONS. To produce the 'i'th combination, 'a' is multiplied by the 'i'th value in the first auxiliary table, 'b' is multiplied by the value in the second auxiliary table, the two products are then added together, and the result is stored in the 'i'th field location pointed to by 'mloc'.

It may be desirable to change the results of this combination process. To do this, the following must be done *PRIOR TO THE FIRST INCLUSION OF 'rmath.h'* :

1.) Define the macro NUM_POSSIBLE_2_COMBINATIONS to have a value corresponding to the size of the array which you going to be creating. This is used as an array index, and the results of defining a value which does not correspond to the size of are undefined.

2.) Define two arrays as follows:

```
word baseCombinationTableA[NUM_POSSIBLE_2_COMBINATIONS]
     = {[A_Values]};
word baseCombinationTableB[NUM_POSSIBLE_2_COMBINATIONS]
     = {[B_Values]};
```

Where [A_Values] and [B_Values] are the initial values of the arrays. They are composed as follows:

The 'i'th value of the array should be the constant value for which 'a' (for baseCombinationTableA) or 'b' (for baseCombinationTableB) is multiplied by to prepare the 'i'th combination.

The values which must be specified here are defined based upon both the field size and the size of the underlying word. A series of macros have been defined to handle these circumstances. To specify the value of all-'m' for GF(n), use a macro following this naming convention:

```
_RMATH_n_WORDVAL_m
```

For example: If you are using GF(4), and you wish to specify the value of all-2s, then use the macro '_RMATH_4_WORDVAL_2'.

If you are *NOT* redefining NUM_POSSIBLE_2_COMBINATIONS, then it is necessary to know the appropriate number of initial values for each table. For GF(n), the appropriate number is n^2. Thus:

GF(n)	Number of Initial Combination Table Values
2	4
3	9
4	16
5	25

If you are redefining the value of NUM_POSSIBLE_2_COMBINATIONS, then it is necessary to supply as many values as the new value of NUM_POSSIBLE_2_COMBINATIONS dictates. (See below.)

For the example given above, the definition of the tables would be as follows:

```
word baseCombinationTableA[NUM_POSSIBLE_2_COMBINATIONS] = {
     _RMATH_3_WORDVAL_0, _RMATH_3_WORDVAL_0,
     _RMATH_3_WORDVAL_0, _RMATH_3_WORDVAL_1,
     _RMATH_3_WORDVAL_1, _RMATH_3_WORDVAL_1,
     _RMATH_3_WORDVAL_2, _RMATH_3_WORDVAL_2,
     _RMATH_3_WORDVAL_2};

word baseCombinationTableB[NUM_POSSIBLE_2_COMBINATIONS] =
     {_RMATH_3_WORDVAL_0, _RMATH_3_WORDVAL_1,
     _RMATH_3_WORDVAL_2, _RMATH_3_WORDVAL_0,
     _RMATH_3_WORDVAL_1, _RMATH_3_WORDVAL_2,
     _RMATH_3_WORDVAL_0, _RMATH_3_WORDVAL_1,
     _RMATH_3_WORDVAL_2};
```

3.) Define the macro BASE_COMBINATION_TABLES. This will override the default tables and use the user-specified ones instead.

4.) Define the macro COMBINATION_TABLE_TYPE to have the value 'OTHER' (without the quotation marks). This will override the built-in table definitions.

There is a further mechanism provided for producing combinations of various field values. Functions like 'generate_Combinations_b' work by augmenting an existing table of generated combinations. To produce these augmentations, the a series of q-1 (for GF(q)) multiplications are made. Each of these multiplications are then added to each of the vectors in an existing array. The

order in which these multiplications are made is established by an array of words, which is defined in the following style:

```
#if (NUMWORDS > 1)
     word baseSmallCombinationTable[FIELDSIZE - 1] =
          {_RMATH_q_WORDVAL_1, . . . , _RMATH_q_WORDVAL_q};
#elif (NUMWORDS == 1)
          const word smallCombinationTable[FIELDSIZE - 1] =
               {_RMATH_q_WORDVAL_1, . . . , _RMATH_q_WORDVAL_q};
#endif
```

Where the last digit in the name of each macro denotes the field vector consisting only of the value of that digit. To produce the combinations in a different order, simply define these arrays manually, with the macros in the desired order.

If defining these tables manually, it will be necessary to #define the macro BASE_COMBINATION_TABLES. This will require that definitions be provided for the 'baseCombinationTable[a/b]' arrays described above. If these are necessary for the operation of your program, it is recommended that if these tables are not to be customized, that the appropriate table be 'cut-and-pasted' out of this file and into your code. If these tables are not needed, arrays of their size can be defined and left unused.

2.2.4 - Procedural Operation Customization

As stated previously, it is possible to use a procedural method to perform the addition and/or multiplication operations.

These 'procedural' modes use function calls, as opposed to bitwise operations perform their operations. They work by adding/multiplying each unit-position individually, looking up the result of operating on two operands 'a' and 'b' by looking up the result in a 2-dimensional array. The table is indexed by using each unit-position of 'a' and 'b' as indices for a table lookup, with the table value being placed in the corresponding unit-position in the resulting 'field'.

Using the procedural operations with normal GF() operations is somewhat of a waste of time, since the bitwise operations perform anywhere from 4X - 40X faster than their procedural counterparts. If it is desired, however, to have non-standard addition and/or multiplication results, then the procedural methods are the easiest way to implement them.

To redefine the results of an operation, it is necessary to do the following (modification of the addition table is demonstrated; any differences for the multiplication table will be listed):

1.) Declare an array of type 'word' in the following format:

```
const word addTable[n][n] = {[Table_Contents]};
```

where 'n' is the size of the current field (i.e. for GF(3), n = y). (replace 'addTable' with 'multTable' for multiplication.)

`[Table_Contents]` will be the values of the lookup table. The form they should take should have the results of the operation a + b (or a * b) stored in entry [a][b]. To specify these values, use the macros from the following naming convention:

For GF(n) fields, where the result you wish to return is m, use:

```
_RMATH_n_SINGLEVAL_m
```

For example, if you are redefining the addition operation for GF(3), and the result of 0 + 2 is to be 1, then in the declaration of 'addTable', the entry [0][2] should have the value '_RMATH_3_SINGLEVAL_1'.

2.) Define the macro ADDITION_TABLE (or MULTIPLICATION_TABLE). This will override the default table and use the user-specified replacement table instead.

Be *very* careful when redefining these operations. There must be the correct number of values in the array, they must be arranged in an n * n 2D array fashion, and if the operation being defined is not symmetric (i.e. 1 + 2 != 2 + 1) then the correct placement of the results in the array is vital.

If you perform a redefinition in this manner, ensure that you test the new tables by running through each possible combination of parameters and verifying their result.

2.2.5 - Extending rmath

It is possible to extend rmath to work on fields beyond GF(5). This should not be attempted except by those with significant C experience, especially in terms of understanding bitwise operations.

Rather than provide a step-by-step guide to the process, consult the lists of necessary macros, variables. For additional guidance, examine the body of the “rmath.h” file. The definitions for GF(2), GF(3), GF(4), and GF(5) are very similar, and illustrate the necessary parts for a full field definition. Simply create new code which resembles the logical structure of that code. Not all parts need to be fully implemented. If, for example, you wish to implement GF(7), but do not wish to create bitwise operations, then simply define the procedural operations for it and use an preprocessor #error directive to prevent compilation for bitwise operations.

2.3 - *rmath* System Functions and Macros list

This list supplements the list given in the rmath *User functions and macros list*. The functions given here are those functions used by *rmath* to support the user functions. They need only be used when attempting to modify *rmath* in some way.

2.3.1 - Functions

short weightValue (word *a*)	Returns the integer weight of the given word, according to the current field type. Breaks each field down into individual field positions. If any of the bits for a position are non-zero, counts that position as valued, even if the bits do not represent a valid field element (i.e. 11 in GF(3)). This result is useful for the distance function. Called during setup to establish the weight tables.
void establishWeightTable (void)	Defines an array with 2^{16} elements. The integer indices of the table entries are with weightValue (word). When vectors are weighed, they are broken into 16- bit sections and used as indices to look up their weight in the table. Called during setup.
void establishCombinationTable (void)	Defines the combination coefficient arrays. Only strictly necessary when NUMWORDS > 1, in order to fill all the field positions of all the coefficient fields. Called during setup.
void establishExtractionMasks (void)	Defines the extraction masks array. The array has PLACES_PER_WORD entries, and this function sets each position to contain the bitmask necessary for extracting only the bits from that position. Called during setup.

void fieldToS (field *p*, char **s*)	Somewhat obsolete printing function. Will store a representation of all positions of the field *a* into *s*, including the insignificant zero positions. For general use, use fieldToS_w(...).
field structAdd (field *a*, field *b*)	Returns the field which is result of adding *a* and *b*. Works by adding the individual words of each field pairwise using ADDWORD(field *a*, field *b*). Conditionally compiled: only if NUMWORDS > 1. Used as the value of ADD(field *a*, field *b*) when NUMWORDS > 1.
field structMultiply (field *a*, field *b*)	Returns the field which is result of multiplying *a* and *b*. Works by multiplying the individual words of each field pairwise using MULTWORD(field *a*, field *b*). Conditionally compiled: only if NUMWORDS > 1. Used as the value of MULT(field *a*, field *b*) when NUMWORDS > 1.
short structWeigh (field *a*)	Returns the weight of the field *a*. Works by summing the weights of each word in *a*. Conditionally compiled: only if NUMWORDS > 1. Used as the value of WEIGHT(field *a*, field *b*) when NUMWORDS > 1.
short structDistance (field *a*)	Returns the distance between the fields *a* and *b*. Works by summing the distances between each word pairwise in *a* and *b*. Conditionally compiled: only if NUMWORDS > 1. Used as the value of DISTANCE(field *a*, field *b*) when NUMWORDS > 1.
word tableAdd (word *a*, word *b*)	Returns the result of adding the field positions in words *a* and *b* by using each pair of positions as indices to the array addTable[]. Conditionally compiled: only if ADDITION_MODE == PROCEDURAL. Used as the value of ADDWORD(field *a*, field *b*) when compiled.

word tableMult (word *a*, word *b*)	Returns the result of multiplying the field positions in words *a* and *b* by using each pair of positions as indices to the array multTable[]. Conditionally compiled: only if MULTIPLICATION_MODE == PROCEDURAL. Used as the value of MULTWORD(field *a*, field *b*) when compiled.
field sToField (char *s)	Returns the field represented by the string s. The function will terminate the program if the string does not consist of the correct number of characters to fill the field. The correct number of characters in the string, including the terminating null character, is given by the macro FIELD_STRING_SPACE. This function underlies sToField_Sized(...).
int structEqual (field *a*, field *b*)	This function returns 1 if *a* and *b* represent the same field, and 0 otherwise. This function works by pairwise comparisons of each word in a multi-word field. Conditionally compiled: only if NUMWORDS > 1. Used as the value of EQUAL(field *a*, field *b*) when NUMWORDS > 1.
int structLess (field *a*, field *b*)	Returns 1 if *a* represents a vector lexicographically less than *b*. Higher indexed positions are lexicographically superior to lower indexed positions. This function works by pairwise comparisons of the words in a pair of multi-word fields, starting in the most significant word and working back to the least significant word. Conditionally compiled: only if NUMWORDS > 1. Used as the value of LEX_LESS(field *a*, field *b*) when NUMWORDS > 1.

int structGreater (field *a*, field *b*)	Returns 1 if *a* represents a vector lexicographically less than *b*. Higher indexed positions are lexicographically superior to lower indexed positions. This function works by pairwise comparisons of the words in a pair of multi-word fields, starting in the most significant word and working back to the least significant word. Conditionally compiled: only if NUMWORDS > 1. Used as the value of LEX_GREATER(field *a*, field *b*) when NUMWORDS > 1.

2.3.2 - Macros - Functional

Note: These are written in standard C functional notation. Since they are macros, they should evaluate to a result of this type. In certain cases, they will substitute for calls to functions with the listed return type. Those which do not evaluate to function calls should evaluate to the given data type.

Note: A small number of macros from the rmath *user functions and macros list* have been duplicated here for further explanation. Those which are designed to be used by end-users are designated as such.

word ADDWORD (word *a*, word *b*)	Should evaluate to the result of adding *a* to *b* according to the specified Galois Field. This will be a function call if ADDITION_MODE == PROCEDURAL and an inline expression if ADDITION_MODE == BITWISE.
word MULTWORD (word *a*, word *b*)	Should evaluate to the result of multiplying *a* by *b* according to the specified Galois Field. This will be a function call if MULTIPLICATION_MODE == PROCEDURAL and an inline expression if MULTIPLICATION_MODE == BITWISE.
field ADD (field *a*, field *b*)	Should evaluate to the result of adding *a* to *b* according to the specified Galois Field. This will be a function call if NUMWORDS > 1 and will inline ADDWORD if NUMWORDS == 1. This is intended to be user called.

field MULT (field *a*, field *b*)	Should evaluate to the result of multiplying *a* to *b* according to the values of the specified Galois Field. This will be a function call if NUMWORDS > 1 and will inline MULTWORD if NUMWORDS == 1. This is intended to be user called.
void CLEAR_FIELD (field *a*)	Sets all positions within *a* to be zero-valued. This inlines a for-loop which sets each word to zero if NUMWORDS > 1 and inlines a zero assignment if NUMWORDS = 1. This is intended to be user called.
short WEIGHT (field *a*)	Evaluates to the weight of *a*. This inlines a function call if NUMWORDS > 1 and inlines WEIGHTWORD if NUMWORDS == 1. This is intended to be user called.
short WEIGHWORD (word *a*)	Evaluates to the weight of *a*. This inlines an expression which takes the sum of the weights of sections of $a \leq 16$ bits in length.
short DISTANCE (field *a*, field *b*)	Evaluates to the distance between *a* and *b*. This inlines a function call if NUMWORDS > 1 and inlines an expression if NUMWORDS == 1. This is intended to be user called.
short DISTANCEWORD (word *a*, word *b*)	Evaluates to the distance between *a* and *b*. This inlines the expression WEIGHT(XORWORD(*a*, *b*)).

word XORWORD (word *a*, word *b*)	Evaluates to taking the binary XOR of *a* and *b*. Thus, it inlines *a* ^ *b*. The XOR of two fields will have zero values in all fields that are the same and non-zero (but not necessarily valid) values in all others. Taking the weight of the XOR will give the distance between the two words.
word EXTRACT (word *a*, int *i*)	Evaluates to the contents of the *i*th field position of *a*. This inlines the expression EXTRACTWORD(...) where the word provided is the word for *a* when NUMWORDS == 1, and is determined by modular arithmetic from the word array in the field structure for *a* when NUMWORDS > 1. This is intended to be user called.
word EXTRACTWORD (word *a*, int *i*)	Evaluates to the contents of the *i*th field position in the word *a*. This masks *a* with appropriate extraction mask from singlePlaceExtractionMasks[] and shifts the value to the low-order end of the word.
void SET (field *a*, int *i*, word *v*)	Sets the value of the *i*th position in *a* to be the value given by *v*. This inlines the expression SETWORD(...) where the word provided is the word for *a* and the position is *i* when NUMWORDS == 1, and where the word is determined by modular arithmetic from the word array in the field structure for *a* and the position is determined by modular arithmetic when NUMWORDS > 1.

void SETWORD (word *a*, int *i*, word *v*)	Sets the *i*th position in *a* to be the value given by *v*. This works by masking the bits representing the desired position in *a* to zeros, shifting *v* to align it's lowest-order position with the desired position in *a*, and then ORing those two bit patterns together. This requires that *v* be all zeros except for the bits in the lowest-order position, which should represent the value to be set in *v*.
void SETPOSN (field *a*, int *i*, int *v*)	Sets the *i*th position in *a* to be the value given by the *v*th entry in the array setValueTable[]. This expression inlines a call to SET(*a*, *i*, setValueTable[*v*]). This array contains 'safe' values which represent each of the possible field elements. Thus, for GF(q), there are q entries in the array with safe values from 0 .. q-1. This is intended to be user called.
int EQUAL (field *a*, field *b*)	Evaluates to the result of a conditional expression which is 1 if *a* and *b* represent the same field, and 0 if they do not. This inlines a call to structEqual (*a*, *b*) if NUMWORDS > 1, and the expression EQUALWORD(*a*, *b*) if NUMWORDS = 1. If the macro EXTERNAL_OPERATIONS is defined, then this macro must be defined explicitly by the user. This is intended to be user called.

int EQUALWORD (word *a*, word *b*)	Evaluates to the result of a conditional expression which is 1 if *a* and *b* represent the same values, and 0 if they do not. This inlines to the expression (*a* == *b*). If the macro EXTERNAL_OPERATIONS is defined, then this macro must be defined explicitly by the user.
int LEX_LESS (field *a*, field *b*)	Evaluates to the result of a conditional expression which is 1 if *a* is lexicographically less than *b*, and 0 otherwise. This inlines the function call structLess(*a*, *b*) if NUMWORDS > 1, and the expression LEX_LESSWORD(*a*, *b*) if NUMWORDS == 1. If the macro EXTERNAL_OPERATIONS is defined, then this macro must be defined explicitly by the user. This is intended to be user called.
int LEX_LESSWORD (word *a*, word *b*)	Evaluates to the result of a conditional expression which is 1 if the field values in *a* are lexicographically less than the field values in *b*, if *a* and *b* were taken as 'free-standing' field vectors, and 0 otherwise. This inlines the expression (*a* < *b*). If the macro EXTERNAL_OPERATIONS is defined, then this macro must be defined explicitly by the user.

int LEX_GREATER (field *a*, field *b*)	Evaluates to the result of a conditional expression which is 1 if *a* is lexicographically greater than *b*, and 0 otherwise. This inlines the function call structGreater(*a*, *b*) if NUMWORDS > 1, and the expression LEX_GREATERWORD(*a*, *b*) if NUMWORDS == 1. This is intended to be user called. If the macro EXTERNAL_OPERATIONS is defined, then this macro must be defined explicitly by the user.
int LEX_GREATERWORD (word *a*, word *b*)	Evaluates to the result of a conditional expression which is 1 if the field values in *a* are lexicographically greater than the field values in *b*, if *a* and *b* were taken as 'free-standing' field vectors, and 0 otherwise. This inlines the expression (*a* > *b*). If the macro EXTERNAL_OPERATIONS is defined, then this macro must be defined explicitly by the user.
field LEX_NEXT (field *a*, int *w*)	This evaluates to the field with at most *w* significant positions which is the nearest lexicographically greater field to *a*, or the all-zero field if there is no such field. This inlines a function call to lexNextField (*a*, *w*). If the macro EXTERNAL_OPERATIONS is defined, then this macro must be defined explicitly by the user. This is intended to be user called.

void COMBINE (field *a*, field *b*, field **l*)	This evaluates to a function call to generateCombinations(*a*, *b*, *l*). This is intended to be user called.
int PACKED_FIELD_ARRAY_SIZE (int *n*, int *w*)	This evaluates to the number of word-sized variables required to store *n* lines of *w* positions each. This is intended to be user called.

2.3.3 - Reserved Functional Macro Namespaces

word _RMATH_*q*_ADD_OPN*x* (word *a*, word *b*)	Macros with names of this form, where *q* is for operations in GF(*q*) and where *x* is any arbitrary string pattern, are intended for use in defining bitwise addition operations on words where trying to develop a single expression is far too cumbersome. These macros can be referenced in other macros of the same name, and, if used, would form the basis of the ADDWORD(*a*, *b*) expression.
word _RMATH_*q*_MULT_OPN*x* (word *a*, word *b*)	Macros with names of this form, where *q* is for operations in GF(*q*) and where *x* is any arbitrary string pattern, are intended for use in defining bitwise multiplication operations on words where trying to develop a single expression is far too cumbersome. These macros can be referenced in other macros of the same name, and, if used, would form the basis of the MULTWORD(*a*, *b*) expression.

2.3.4 - Macros with Significant Numeric Values

Note: All numeric values are represented as non-negative integers.

FIELDSIZE	Evaluates to the number of elements in the field being represented. Valid values: 2, 3, 4, 5
WORDLEN	Evaluates to the number of bits in a word. Valid values: 32, 64
LONG64	Evaluates to the name of a 64-bit integral datatype which can be specified as 'unsigned' on the target machine. Necessary for WORDLEN == 64 only. Valid values are system specific.
PLACES_PER_WORD	Evaluates to the number of field positions representable in a single word. Value should be: ⌊sizeof(word) / FIELDSIZE⌋
BITS_PER_PLACE	Evaluates to the number of bits used to represent each field position. Value should be: ⌈lg(FIELDSIZE)⌉
HIGH_ORDER_PAD_BITS	Evaluates to the number of bits in a word which are not used for representing any field position. Value should be: WORDLEN - (BITS_PER_PLACE * PLACES_PER_WORD)

LOW_ORDER_PLACE_MASK	Evaluates to a bitmask with the lowest BITS_PER_PLACE bits set to 1 and the rest set to 0. ANDing this mask with a word containing field values should produce a word with all but the lowest-order field position set to zero. Example, for BITS_PER_PLACE == 3: LOW_ORDER_PLACE_MASK = 0x7
SINGLE_PLACE_EXTRACTION_MASK	Must evaluate to the same expression as LOW_ORDER_PLACE_MASK.
FIELD_STRING_SPACE	Evaluates to an integer defining the number of characters necessary to write out the shortest string necessary to hold the string representation of a full field variable (including zero-positions not set by the user's program.) Value: (NUMWORDS * PLACES_PER_WORD + 1) * sizeof(char)
NUM_POSSIBLE_2_COMBINATIONS	Evaluates to the number of fields produced by the COMBINE(...) operation. Value: varies with the values of FIELDSIZE and COMBINATION_TABLE_TYPE
BASE_PRINT_CHAR	Evaluates to the ASCII value which is to be issued for the zero character when converting fields to strings. The characters produced for the other symbol positions are determined by adding their numeric representations to the value of this character. Value: '0'.

BASE_READ_CHAR	Evaluates to the ASCII value which, when read from a string, represents a zero element. The characters read for the other field elements are determined by adding their numeric representations to the value of this character. Value: ‘0'

2.3.5 - Reserved Namespaces for Significant Value Macros

_RMATH_*n*_BIT_WEIGHT_MASK_*m*_BIT	Reserved for the bitmasks used during extraction of 16-bit chunks. *n* is WORDLEN. *m* is an integer which is used to differentiate masks when implementing different systems of representation for the same field. Each mask should define the bits necessary for extraction of a ≤ 16-bit section of a given word for determining the weight of that section. These are especially useful when a field position spans the boundary between 16-bit sections.
_RMATH_*q*_WORDVAL_*n*	Reserved for words which represent a group of field elements all set to value *n* from GF(*q*). Non-significant bit positions should be set to 0. Each field representation must define a full complement of these. They are used for seeding the combination tables.
_RMATH_*q*_SINGLEVAL_*n*	Reserved for words which represent a group of field elements with all positions set to 0, except for the least-significant field position, which is set to value *n* from GF(*q*). Non-significant bit positions should be set to 0. Each field representation must define a full complement of these. They are used for the SET family of operations.

2.3.6 - Macros with Significant Symbolic Values

Name	Description	Values
ADDITION_MODE	Defines how addition is to be carried out.	BITWISE (specifies processor-level bitwise operations) PROCEDURAL (specifies array lookups)
MULTIPLICATION_MODE	Defines how multiplication is to be carried out.	BITWISE (specifies processor-level bitwise operations), PROCEDURAL (specifies array lookups)
COMBINATION_TABLE_TYPE	Defines what combinations will be produced by the COMBINE(a, b, ...) operation.	FULL (all q^2 combinations for GF(q)) NON_ZERO (q^2 - 1 combinations, does not produce the result of 0*a + 0*b), NON_ZERO_NON_IDENTITY (q^2 - 3 combinations, like NON_ZERO but also ignores 1*a + 0*b, 0*a + 1*b) OTHER (indicates user specified table)
RMATH_5_SYSTEM	Defines how GF(5) is represented. Currently defined in “rmath.h” to be OLD_SYSTEM	NEW_SYSTEM (specifies experimental representation) OLD_SYSTEM (specifies standard representation)

2.3.7 - Non-Valued Macros

Note: Defining these macros will affect some aspect of the compilation and/or operation of the system. They do not need to evaluate to any particular value.

EXTERNAL_OPERATIONS	If defined, this macro overrides all definitions which compose the lowest level field operations. This includes all macros related to word level representation, WORDVAL and SINGLEVAL constants, weight determination procedures, addition and multiplication procedures, combination and equivalence tables. This should be defined if the user intends to provide their own field definition. It should be #included in the compilation prior to this macro.
ADDITION_TABLE	If defined, overrides the internal definition of addTable[][] and uses one defined by the user. Only significant if ADDITION_MODE == PROCEDURAL.
MULTIPLICATION_TABLE	If defined, overrides the internal definition of multTable[][] and uses one defined by the user. Only significant if MULTIPLICATION_MODE == PROCEDURAL.
BASE_COMBINATION_TABLE	If defined, overrides the internal definition of the various combinationTable[][] arrays and uses those defined by the user. Overriding these tables allows the user to specify what combinations are to be produced by the COMBINE operation, and in what order.

2.3.8 - Arrays - Globally Defined

word addTable[][]	q × q array (for GF(q)). Defines the results of addition for each possible pair of field values. Entry [i][j] should contain the SINGLEVAL (see *Reserved Namespaces for significant value macros*) result of the operation 'i + j'.
word multTable[][]	q × q array (for GF(q)). Defines the results of multiplication for each possible pair of field values. Entry [i][j] should contain the SINGLEVAL (see *Reserved Namespaces for significant value macros*) result of the operation 'i * j'.
word baseCombinationTableA[]	Array of NUM_POSSIBLE_2_COMBINATIONS WORDVAL (see *Reserved Namespaces for significant value macros*) entries. The *i*th entry is the constant value by which field *a* is multiplied by in the operation COMBINE(*a*, *b*, ...) to produce the *i*th result vector. When taken in combination with baseCombinationTableB[], there should be each combination of constants necessary to produce all NUM_POSSIBLE_2_COMBINATIONS combinations.
field combinationTableA[]	Array of NUM_POSSIBLE_2_COMBINATIONS fields. If NUMWORDS == 1, then this table is identical to the description of smallCombinationTableA[]. If NUMWORDS > 1, then the entries in baseCombinationTableA[] are used to populate each word underlying the corresponding entries in this array.

word baseCombinationTableB[]	Array of NUM_POSSIBLE_2_COMBINATIONS WORDVAL (see *Reserved Namespaces for significant value macros*) entries. The *i*th entry is the constant value by which field *b* is multiplied by in the operation COMBINE(*a*, *b*, ...) to produce the *i*th result vector. When taken in combination with baseCombinationTableA[], there should be each combination of constants necessary to produce all NUM_POSSIBLE_2_COMBINATIONS combinations.
field combinationTableB[]	Array of NUM_POSSIBLE_2_COMBINATIONS fields. If NUMWORDS == 1, then this table is identical to the description of smallCombinationTableB[]. If NUMWORDS > 1, then the entries in baseCombinationTableB[] are used to populate each word underlying the corresponding entries in this array.
word baseSmallCombinationTable[]	Array of (q - 1) (for GF(q)) entries which represent the constant factors that are used to produce linear combinations of vectors in the generateCombinations...(...) family of operations. All non-zero WORDVAL (see *Reserved Namespaces for significant value macros*) values should be represented. Changing the order of these constants will change the order in which the resultant combinations are produced. This table is only defined if NUMWORDS > 1.

field smallCombinationTable[]	Array of (q - 1) (for GF(q)) entries which represent the constant factors that are used to produce linear combinations of vectors in the generateCombinations...(...) family of operations. If NUMWORDS == 1, then this table is identical to the description of baseSmallCombinationTable[]. If NUMWORDS > 1, then the entries in baseSmallCombinationTable[] are used to populate each word underlying the corresponding entries in this array.
int equivTable[][]	(q - 1) × (q - 1) array (for GF(q)). Used by the function sameDescription(*a*, *b*). The rows of the array define mappings on the list of field value quantities such that when the two descriptions are compared. If equivTable[*i*][*j*] = *k*, then during the *i*th set of comparisons the quantities of symbol *j* in description *a* will be compared against the quantities of symbol *k* in description *k*. The values assigned to the array should be in the range 1..(q - 1), and each row should contain one each of those values. In fact, the [*i*][*j*]th entry should be the result of taking *i* * *j* in GF(q).
word setValueTable[]	Array of q entries (for GF(q)). The *i*th entry should be the SINGLEVAL (see *Reserved Namespaces for significant value macros*) representing the *i*th symbol in GF(q). These values are used for the SETPOSN(...) operation to ensure that the proper values are used for position setting.
word singlePlaceExtractionMasks[]	Array of PLACES_PER_WORD entries. The *i*th entry should be the bit pattern which when ANDed to a given word containing field vector elements should result in a word with all bits being 0 except for those from position *i* in the original word. These values are used in the EXTRACT operation.

2.3.9 - Data Types Defined by *rmath*

Name	Description
word	Underlying data type for storing field values. It must be an integral type of either 32 or 64 bits in length, specifiable as unsigned, and should have no unusual characteristics (i.e. in the high-order bits) when dealt with via the bitwise logical operations when unsigned. Values vary from system to system. 'long int' should work for all 32 bit requirements. 'long long int' is the data type for SGI machines running MIPS R10000+ processors and IRIX 6+. Consult your machine's compiler guide for more information.
struct fieldStruct	A structure with a single field, an array 'words' consisting of NUMWORDS words. Implementing this as a structural type permits assignments of whole arrays (by structure assignment) at the language level. This type is only defined whenever NUMWORDS > 1.
field	For the end user, the fundamental data type of the *rmath* system. These are what the user will work on in the body of their code. If NUMWORDS == 1, field is 'typedef'ed to be a word. If NUMWORDS > 1, field is 'typedef'ed to be a fieldStruct structure. Operations on fields generally have two sets of definitions: one for the NUMWORDS == 1 case (which are generally simpler), and one for the NUMWORDS > 1 case (which are generally more involved and often require additional procedure calls to support the necessary looping.)
struct descriptionStruct	A structure containing a single field, which is an array 'values' of q short ints (for GF(q)). When initialized as a description, values[i] will contain the an integer indicating the quantity of the symbol i in the field vector which it describes.
description	This is 'typedef'ed to be struct descriptionStruct structure. This is intended for use by the end user.

3 - References

King, K. N. C Programming: A Modern Approach. New York: Norton, 1996.

Kerninghan, Brian W., and Ritchie, Dennis M. The C Programming Language. 2nd ed. Englewood Cliffs: Prentice Hall, 1998.

McKay, Brendan D. *nauty* User's Guide. Version 1.5. Technical Report TR-CS-90-02, Australian National University, Department of Computer Science, 1990.

SGI, Inc. C Language Reference Manual (007-0701-130). Mountain View: SGI, 1999.

4 - Appendices

4.1 - Appendix A - Sample program using *rmath*

4.1.1 - Introduction

The following code is an example program using *rmath*. While it does not demonstrate all of the features, it should provide a taste of how to set up a program and to use some of the input and output functions.

4.1.2 - Code

```
/* Linear Code Vector Determination Program

   Author: Michael Letourneau
   Version: 1.0;

   This program is primarily an example of the use of the 'rmath'
   system for representing Galois field elements.

   The program itself accepts the parameters n, k, and d for a
   linear code in GF(q), where q is the value of the preprocessor
   macro FIELDSIZE. It will then accept the k basis vectors of
   the code.

   The program will use 'rmath' operations to produce the vectors
   containing the words of the code specified by the basis vectors.
   The generation of codewords will cease if any combination of
   the basis vectors does not meet the minimum weight requirement.

   Once the generation of codeword vectors is complete, they will
   be printed to standard output along with their weights. */


#include <stdio.h>
#include <stdlib.h>

                                    /* rmath parameters  - specify before
                                      including rmath files. */

#define WORDLEN 64                  /* Length of word variable */

#define LONG64 long long int        /* Name of 64-bit word data type.
                                      64-bit mode requires a type name.
                                      32-bit assumes 'int'. */

#define FIELDSIZE 2                 /* Size of the field to be represented.
                                       This program should run without
                                        modification (beyond this value) for
                                       any valid field size. */
```

```
#define NUMWORDS 1                    /* Number of words used to represent
                                         a single 'field' variable. */

#define ADDITION_MODE BITWISE         /* Parameters defiing how operations */
#define MULTIPLICATION_MODE BITWISE   /* are carried out.                  */

#define COMBINATION_TABLE_TYPE NON_ZERO_NON_IDENTITY  /* Control the operation
                                                         of COMBINE(..) */

#include "rmath.h"                  /* Both the header and library sources  */
#include "rmath.c"                 /* be compiled with the above settings. */


void main (void) {

      int n, k, M, d, i, numWords;

      field *basis, *words;

      char tempString[100];         /* Used as a temporary string for I/O. */

      setupRMath();                 /* This *MUST* be called prior to using
                                       any rmath operations. It establishes
                                       certain auxilary tables. */

      printf("Linear Code Vector Determination Program\n\n");

      printf("Code Parameters\n");

      printf("n: ");
      scanf("%d", &n);

      printf("k: ");
      scanf("%d", &k);

      printf("d: ");
      scanf("%d", &d);

      M = 1;

      for (i = 0; i < k; i++)
            M *= FIELDSIZE;

      basis = malloc(sizeof(field) * k);
      words = malloc(sizeof(field) * M);

      printf("\nBasis Vectors\n");

      for (i = 0; i < k; i++) {
            printf("%d of %d: ", i + 1, k);         /* Read basis vector as */
            scanf("%s", tempString);                /* a string.            */

            basis[i] = sToField_Sized(tempString);  /* Convert string to    */
                                                    /* field type.          */
      }

      numWords = 0;
```

```
        for (i = 0; i < k; i++) {            /* Generate combinations for each
                                                basis vector. If the function
                                                returns <= 0, a combination is
                                                underweight, and the code is not
                                                valid. */

              numWords = generateCombinations_Inc_Weight(basis[i],
                                                          words, numWords, d);

              if (numWords <= 0) {
                    numWords *= -1;
                    printf("\nUnderweight combination after %d. Stopping.\n",
                                                                  numWords);
                    break;
              }
        }

        printf("\nCode Vectors Computed:\n\n");

        for (i = 0; i < numWords; i++) {
              fieldToS_w(words[i], tempString, n); /* Convert field to string */

              printf("%d:\t%s   Weight: %d\n", i + 1, tempString,
                                                    WEIGHT(words[i]));
        }

}
```

4.1.3 - Sample runs with FIELDSIZE == 2

[Run #1 - Vectors describe good code]

```
Linear Code Vector Determination Program

Code Parameters
n: 7
k: 4
d: 3

Basis Vectors
1 of 4: 1000011
2 of 4: 0100101
3 of 4: 0010110
4 of 4: 0001111

Code Vectors Computed:

1:   1000011   Weight: 3
2:   1100110   Weight: 4
3:   0100101   Weight: 3
4:   1010101   Weight: 4
5:   1110000   Weight: 3
6:   0110011   Weight: 4
7:   0010110   Weight: 3
8:   1001100   Weight: 3
9:   1101001   Weight: 4
10:  0101010   Weight: 3
11:  1011010   Weight: 4
12:  1111111   Weight: 7
13:  0111100   Weight: 4
14:  0011001   Weight: 3
15:  0001111   Weight: 4
```

[Run #2 - Vectors Describe Incorrect Code]

```
Linear Code Vector Determination Program

Code Parameters
n: 7
k: 4
d: 3

Basis Vectors
1 of 4: 1000011
2 of 4: 0100101
3 of 4: 0010010
4 of 4: 0001111

Underweight combination after 6. Stopping.

Code Vectors Computed:

1:   1000011   Weight: 3
2:   1100110   Weight: 4
3:   0100101   Weight: 3
4:   1010001   Weight: 3
5:   1110100   Weight: 4
6:   0110111   Weight: 5
```

4.1.4 - Sample run with FIELDSIZE == 3

```
Linear Code Vector Determination Program

Code Parameters
n: 13
k: 3
d: 9

Basis Vectors
1 of 3: 1001111001111
2 of 3: 0101200111122
3 of 3: 0010012121212

Code Vectors Computed:

1:   1001111001111   Weight: 9
2:   2002222002222   Weight: 9
3:   1102011112200   Weight: 9
4:   1200211220022   Weight: 9
5:   2100122110011   Weight: 9
6:   2201022221100   Weight: 9
7:   0101200111122   Weight: 9
8:   0202100222211   Weight: 9
9:   1011120122020   Weight: 9
10:  1021102210202   Weight: 9
11:  2012201120101   Weight: 9
12:  2022210211010   Weight: 9
13:  1112020200112   Weight: 9
14:  1122002021021   Weight: 9
15:  1210220011201   Weight: 9
16:  1220202102110   Weight: 9
17:  2110101201220   Weight: 9
18:  2120110022102   Weight: 9
19:  2211001012012   Weight: 9
20:  2221010100221   Weight: 9
21:  0111212202001   Weight: 9
22:  0121221020210   Weight: 9
23:  0212112010120   Weight: 9
24:  0222121101002   Weight: 9
25:  0010012121212   Weight: 9
26:  0020021212121   Weight: 9
```