Gerald's Column
by Gerald Fitton

In my last three articles I have described some of the properties and theorems of Modulus Algebra. I must dispel a misconception I might have given. My primary interest is not in Encryption. I have used Encryption as my extended example of Modulus Algebra because Encryption is an easily understood application which demonstrates usefulness of Modulus Algebra—and because I can create relatively simple examples which will run in my favourite spreadsheets rather than in (harder to follow) machine code or BASIC.

Nevertheless I am grateful to those of you who have written to me about the History, Methods and Practices of Encryption ranging from the so called Caesar Cypher or Wheel, through the Enigma machine, to the modern RSA Public Key system. Also I was delighted by the photocopies of magazines, books and technical reports which you have sent to me as well as the web sites to which you have directed me! There are so many of you so please allow me a general "Thank you" rather than trying to thank you individually,

Instead describing RSA system this month I have decided to postpone it and, instead, I shall describe an earlier system known as the DHM Protocol.

Telephone

As a rule I reserve notes about contacting me until the end of my article but, for those of you who might get bored before the end I include this item in an unconventional place in the hope that it will make more impact.

The telephone line I have used for years is becoming clogged with junk faxes and with junk telephone calls. I have tried all the usual remedies from writing to many central 'do not send me junk' Agencies to attaching a permanently connected answering machine with a call monitoring facility. With some reluctance I have decided that the only hope for a permanent solution is to have the line disconnected; I do not intend to make public my new 'phone number (anyway I haven't got it yet).

If you are 'connected' (to the Internet) then please communicate with me by email. If not then you will have to accept 'snail mail' speed.

Lowest Common Multiple (LCM)

Last month I described exactly what was meant by the Highest Common Factor (HCF) but I failed to describe the LCM!

The LCM of a set of numbers is a number which is equal to or larger than the largest (of the set). Every number in the set divides into the LCM leaving no remainder and there is no number smaller than the LCM for which this is true. Using the Modulus notation, if the LCM of the set of numbers a, b, c, etc is L then (L mod a) = 0, (L mod b) = 0, etc; L is the smallest number for which this set of modulus equations is true.

As an example (and a small digression) let me explain how I was taught to find the LCM of the numbers 6 and 20 when I was about nine years old.

The first step was to ‘reduce’ the numbers to their prime factors. These are: 6 = 2 * 3 and 20 = 2 * 2 * 5. The LCM of 6 and 20 must contain all these factors. We start with all the factors of 6 and then look at the factors of 20 and see which factors of 20 we haven’t yet included. You will see that we have one of the 2s (so we need one more from the 20) and we need the 5 (from the 20). Hence the LCM of 6 and 20 is 2 * 2 * 3 * 5 = 60.

This method of complete factorization is one way of finding the LCM of a set of numbers. When there are only two numbers an easier method is to multiply the two numbers together and divide the product by their HCF. Finding the prime factors of a number is hard work—finding the HCF, thanks to Euclid, is much easier!

Continuing with my example, the HCF of 6 and 20 can be found using Euclid’s continued subtraction Algorithm. (20 − 6) = 14, then (14 − 6) = 8, next (8 − 6) = 2, penultimately (6 − 2) = 4 and finally (4 − 2) = 2. The HCF of 6 and 20 is 2.

On the Archive monthly disc in the sub directory DHM you will find the file [c_HCF] as a PipeDream and as a Fireworkz custom function; these custom functions utilise Euclid’s Algorithm of continued subtraction. The file [TestHCF] is the ‘driver’ for this custom function which will allow you find the HCF of a pair of numbers. If you enter the numbers 20 and 6 the custom function will return the HCF as the number 2 in milliseconds.

The LCM of 6 and 20 is the product of the numbers (6 * 20) divided by the HCF; the answer is best calculated as 6 * (20/2) = 60.

The Diffie–Hellman–Merkle (DHM) Protocol

In 1977 the RSA trio (Ronald Rivest, Adi Shamir and Leonard Adleman) published their now famous Public Key Encryption System in Scientific American. However, a year before that, using ideas put forward by Ralph Merkle, the duo Whitfield Diffie and Martin Hellman published what is now known as either the DH or DHM Protocol.

There is a great deal of evidence that an Englishman called James Ellis conceived the basic ideas in 1970 based on notes he found in a Bell Telephone report (written sometime between 1939 and 1946) by an anonymous author. Later, in 1973 and 1974, a couple of mathematicians working at the Government Communications Headquarters (GCHQ), Clifford Cocks and Malcolm Williamson, developed the ideas of James Ellis and came up with (what is now known as) the Diffie–Hellman–Merkle Protocol before Diffie, Hellman and Merkle thought of it. This evidence is presented in a book called “The Code Book” by the (now famous and popular) mathematical author Simon Singh. I find it interesting that these Englishmen seek no fame nor publicity for their prior discovery.

In 1976 computing power was not what it is now. When the authors of the DHM protocol described the prospective use of their discovery they stated that its use should be limited to the exchange of a Secret Shared Key. They pointed out that using their rather laborious process for the exchange of bulky messages would be far too slow for practical use.

They suggested that their Protocol should be used to exchange a relatively small amount of information such as a Shared Secret Key. The Secret Key would ‘unlock’ a much simpler encryption system which would be used for the bulk transfer. After describing the DHM Protocol I shall give two examples of such a simpler system which require Secret Keys.

Alice and Bob

I don't know who invented Alice and Bob (and Clare, Denis, etc) but the intention is to persuade the reader that Alice and Bob are much more friendly than the inhospitable 'Person A' and 'Person B'. This set of A, B, C, D, etc characters seems to permeate current encryption literature. I shall follow this convention and describe the DHM protocol for generating a secret key between Alice and Bob rather than between Persons A and B.

Stage 1. Two relatively prime numbers, g and m are selected and exchanged openly by Alice and Bob. Relatively prime means that the HCF of g and m is 1.

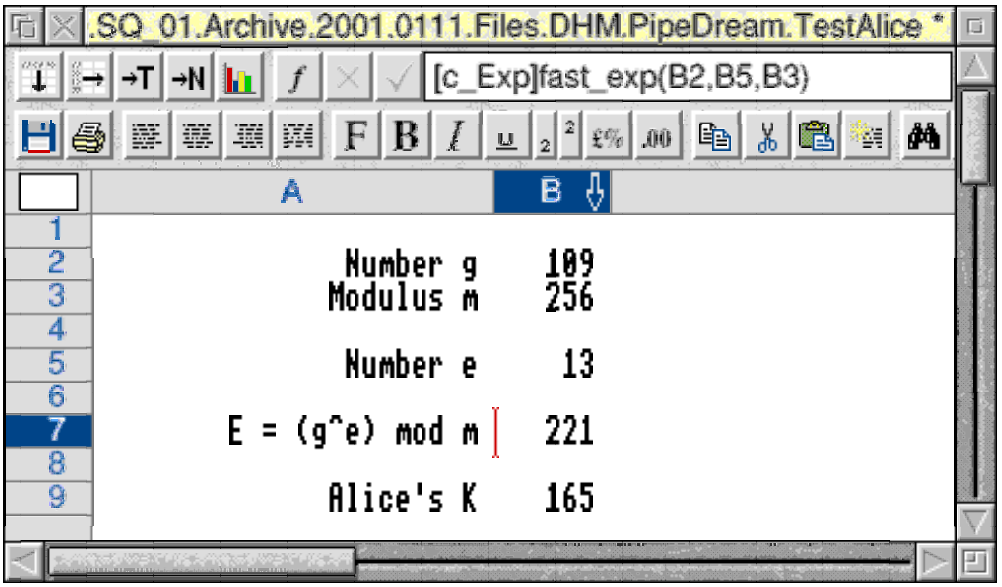
Stage 2. Alice chooses a number, e, and calculates E = (g^e) mod m. Alice sends the value of E openly to Bob but keeps e a secret. At the same time Bob chooses d and calculates D = (g^d) mod m and then sends D openly to Alice. Bob keeps his d secret.

Stage 3. Bob calculates K = (E^d) mod m. Alice calculates K = (D^e) mod m.

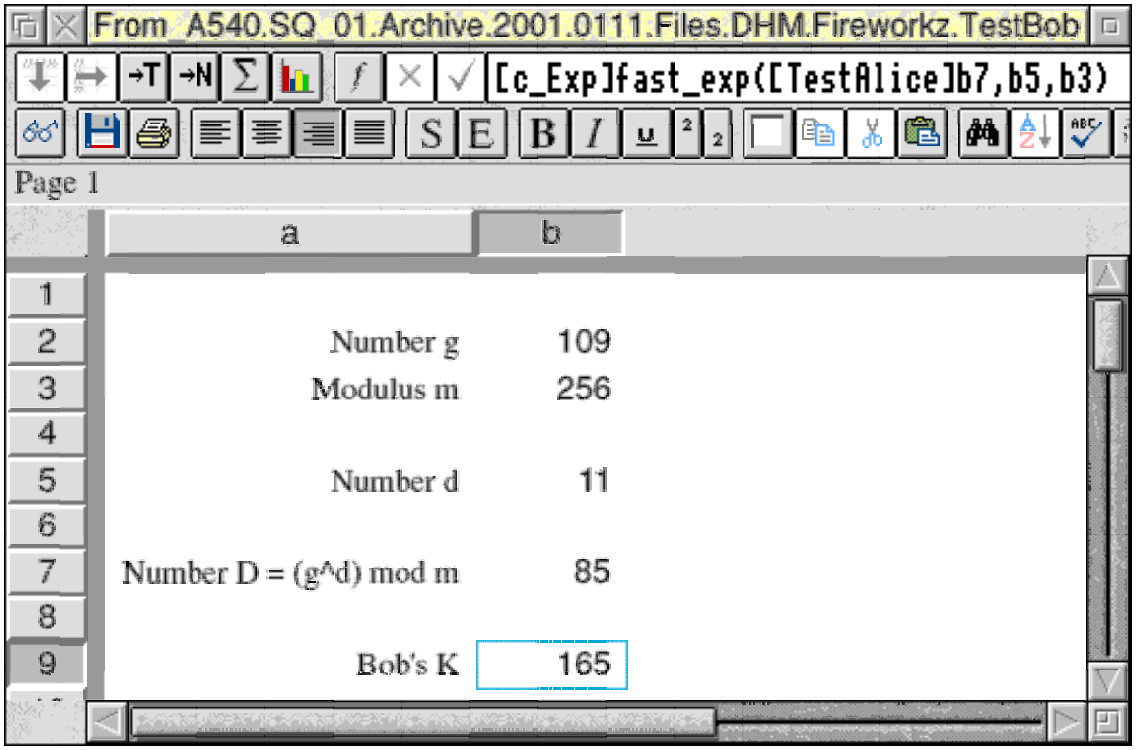
If g and m are relatively prime, the values of K calculated by Alice and Bob are both non zero. Can you see that both values of K = (g^(e*d)) mod m? Remember that the order of exponentiation doesn't matter. This shared value of K is the shared secret key.

On the Archive Monthly disc in the DHM subdirectory you will find three files in both PipeDream and Fireworkz format. The custom function file [c_Exp] executes the 'fast power' algorithm which I described last month. The other two are spreadsheets used by Alice and Bob to calculate K. Double click on [TestBob] to load the set of three files.

The spreadsheet shown in the screenshot below, [TestAlice], contains the numbers g and m which are agreed on and exchanged openly between Alice and Bob. They must be relatively prime−this means that the HCF of g and m must be 1. The spreadsheet doesn't check this. If the HCF is not 1 then you will find that the spreadsheets return K = 0. The number e is chosen by Alice. There is no point in choosing a number for e which is greater than m since the results repeat. For example 257 gives the same result as 1 and so on.



Alice calculates the value of E using the formula which is shown in the screenshot of her spreadsheet. This formula uses the 'fast exponentiation' algorithm. Alice sends E to Bob.



Similarly Bob chooses d. He calculates D and sends it to Alice. When Bob receives E from Alice he calculates K using the formula shown in the screenshot of his spreadsheet.

You will see (in the screenshots) that both Bob and Alice have found the same value of K despite the fact that Alice is using PipeDream and Bob is using Fireworkz.

So far as secrecy goes, if you were the 'code cracker' then you would know g, m, E and D but you would not know e, d and K. Although K is the Secretly Shared Key, your code breaking efforts should be targeted at finding d (and e) when you know D, (E), g and m.

Of course I know that (in my example with m = 256) you could use a Trial and Error method to run through every value of K in a few seconds—this is not what I am asking you to do. Essentially my question to you is this. Can you find a formula for d when you are given D, g and m and you know that D = (g^d) mod m?

If you have been following this series with sufficient understanding then you will know the sort of difficulties which arise in solving this modulus equation for d. So that I can gauge the extent of your understanding (and interest) perhaps you would drop me an email or letter if you think you know the sort of thing which you might try. Please do this even if you can't find the numerical solution but have a 'sort of' method.

A few points about K. K is less than m. For security and for a large 'Key Space', m must be a large number. The number m does not have to be prime but if it is then g and m are bound to be relatively prime. Usually m is chosen to be prime.

It is important that you appreciate that (using the classic method of defining the DHM key) neither Alice nor Bob can choose the value of K; it emerges from the calculations.

Exclusive OR

Five correspondents have written to me describing an Exclusive OR (EOR) system of encryption. Some are more sophisticated than others but what they all have in common is that the numerical form of the message is put through an EOR operation with a sequence of random numbers. Such an EOR operation is much faster then, say, a DHM exchange because it requires much less computing power.

Let me try to clarify how these EOR systems work. As a simple example the random number sequence is chosen as 8-bit random numbers in the range 0 to 255 inclusive. The numerical version of the message consists of the ASCII codes of the characters which form the text. Each character of the message is subjected in turn to an EOR operation with one number taken in its corresponding turn from the sequence of random numbers.

The message can be decoded as easily as it was encoded provided that the same sequence of random numbers is available to both Alice and Bob. All Bob has to do is to execute an EOR operation on the encoded message using the identical sequence of random numbers. The sequence of random numbers can be selected from a long list of random numbers made available on, say, a CD, or on a website—or they can be generated using a pseudo random number sequence generator such as those available in PipeDream and Fireworkz. Note that the length of the random number sequence is the same length as the message.

The security of the system relies on a secret exchange of the position in the long list of random numbers from which you start the EOR process. The ‘secret pointer’ to this position emerges from, say, a DHM Exchange.

The level of security can be increased by exchanging a set of Shared Secret Keys and using them to select more than one sequence (all of the same length—the length of the message) from the long list of shared random numbers. All these multiple sequences are EORed together before executing the final EOR on the message.

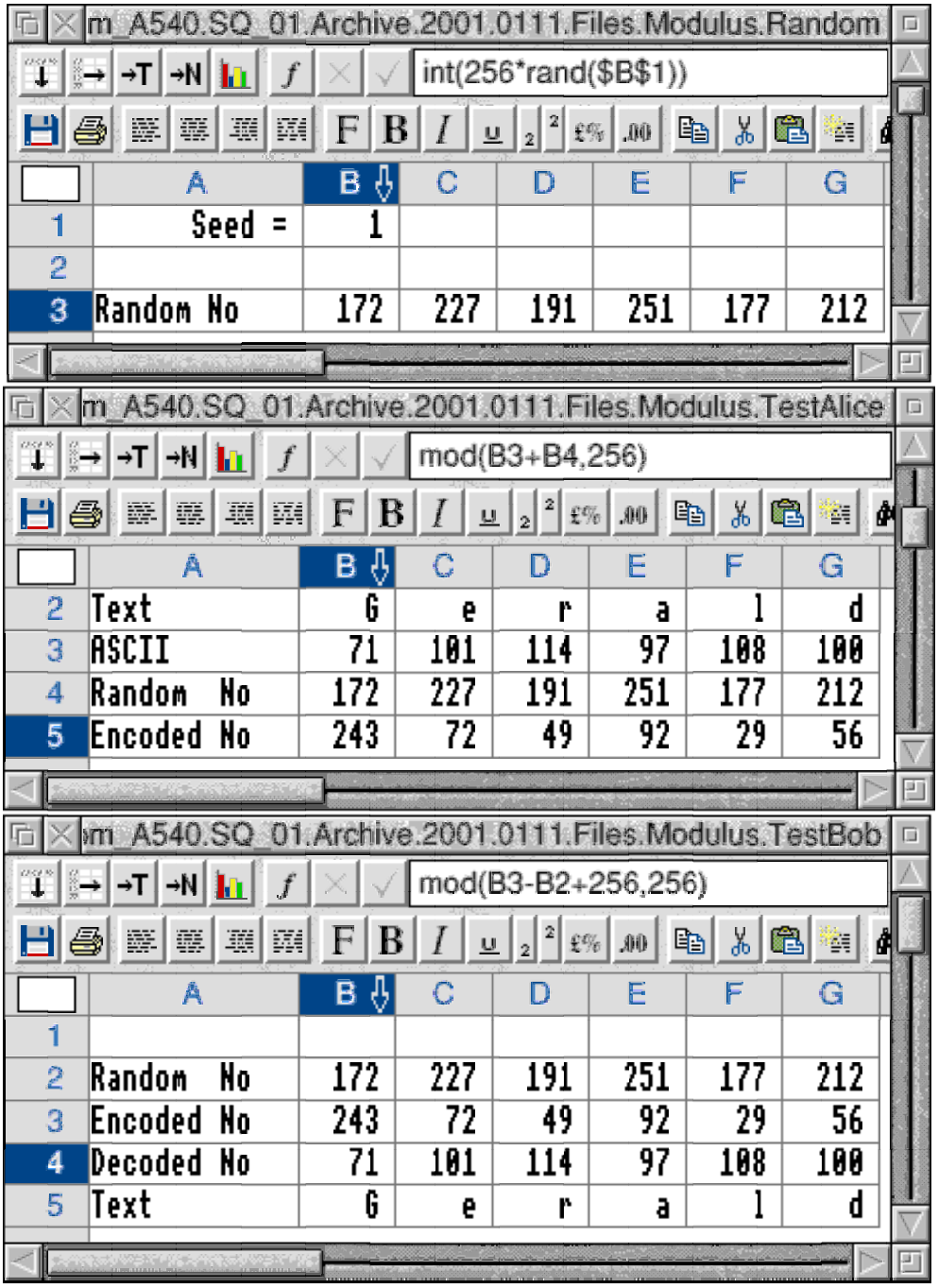
On the Archive monthly disc in the GLF_Eor directory you will find an example of a simple EOR encryption system in both PipeDream and Fireworkz format. Click on the [TestBob] file and all necessary files will load and run. I have written an EOR custom function which I’m sure could be made to run much more quickly but I regret that I have run out of time this month. I suggest that, before you experiment with the EOR files you have a look at the next section, Modulus.

Modulus

My EOR custom function uses the modulus(,) function. Indeed it is possible to regard an EOR operation is a ‘sort of’ multiple modulus operation which is carried out on the binary representation of the number. In this section I show how you can use some variant of the Modulus function as an alternative to the EOR operation.

For my simple example I have added the random number to the character’s ASCII code and found the modulus using m = 256. Addition is a relatively easy modulus operation to reverse by using subtraction. Of course you could multiply the ASCII code by the random number and decode using the Inverse Multiplier Algorithm [c_Inv].

My simple (addition) example comprises the three spreadsheets [TestAlice], [TestBob] and [Random] which you'll find in the GLF_Mod sub directory. To run this example double click on the file [TestBob]; all three files shown in the screenshot below will load. Before describing the operation of this system I must tell you something about the particular random number sequence I am using.



The File [Random]

The PipeDream file [Random] generates a sequence of pseudo random numbers using rand(seed). There is something about the function rand(seed) which you might not know. If you Quit PipeDream from the icon bar and then load the file [Random] then the sequence of pseudo random numbers will always be the same for the same 'seed'. If you change the 'seed' then the sequence will be different. If you do not Quit first (or if you have already generated random numbers) then the sequence will not be the same.

So you will see that (if we both have PipeDream) both you and I can generate the same random number sequence provided we exchange (secretly) the value of the ‘seed’. For this simple example I have chosen the ‘seed’ to be 1 but it could be any number. To use such a (hypothetical) system we would have to agree on a Secret Key for this ‘seed’. One way in which we could generate such a Secret Key is by using the DHM Protocol.

Incidentally, Fireworkz has the same facility but a ‘seed’ of 1 produces 21 as first random number whereas PipeDream produces 172. This means that you can not use Fireworkz if I use PipeDream for the ‘seed’ and vice versa!

The Files [TestAlice] and [TestBob]

The file [TestAlice] encodes and the file [TestBob] decodes the message “Gerald Fitton”.

Of course the modulus function I have chosen (the addition of a random number) is deliberately simple so that I can demonstrate the principle. Furthermore the Key Space I have chosen is small enough to be vulnerable to a Trial and Error attack. Please don’t ‘have a go’ at me for making it so simple—I am not trying to devise an encryption system but only trying to demonstrate principles. What I am trying to do is to keep you amused and, at the same time, encourage your interest in Modulus Arithmetic and Algebra.

Summary

The Diffie–Hellman–Merkle Protocol is a method of creating a Shared Secret Key. The intention is not to use that Secret Key to encode bulky messages but to provide a pointer used by a simpler and faster encryption algorithm. The DHM Protocol uses the modulus function as the ‘nearly one way’ operator. It requires a fast exponentiation algorithm such as the one which I have included as PipeDream and Fireworkz custom functions. I would remind you that these custom functions are limited by the precision of the Floating Point Emulator and that numbers of that size are vulnerable to attack because it is possible to factorize numbers of this magnitude in a relatively short time.

Primes

Almost as a ‘by-the-way’ I have improved the custom function which returns the largest prime factor of the number which you enter into the spreadsheet. It now runs about twice as fast as previously. You will find the improved versions in the DHM subdirectory.

I was surprised to discover that Fireworkz runs this custom function faster than PipeDream because I was always under the impression that PipeDream was faster than Fireworkz! This deserves further investigation (but I don’t have the time for right now).

Finally

You can email or write to me at the addresses given in Paul’s Fact File.

Please note that by the time this is published we shall no longer have a telephone or fax.