Polynomials Graze in Galois Fields

The Weird Arithmetic Behind Error Correction

The best rules are those you make yourself

Andy Talbot G4JNT

Introduction

Anyone who has tried to understand the intricacies of error correction and coding will very soon come across references to Galois Fields and polynomials. Both of these are used in calculating the extra bits or symbols that are added to source data before transmission in the more advanced Forward Error Correction techniques. Once the basics of simple error correction, parity checks and Hamming codes have been covered nearly all text books and papers then seem to become very esoteric and talk about these mysterious entities. After becoming confused at this stage so often in the past, I was determined to try to make sense of it. The light bulb moment seemed to happen obliquely, after studying the source code for the various modes within the WSJT-X suite of datamode software, in particular that for the latest Q65 mode, then going back and looking at a couple of textbooks .

Here I hope to be able to explain these two concepts in a way such that they can become just a tool rather than a mental block when looking further into error correction encoding and techniques. Galois Fields (GFs) and polynomials are not in themselves intimately connected, and the latter is distorted in seemingly-strange ways to work in binary logic. First of all we'll look at the definition of GFs in a simple arithmetical way and how they can be generated using prime numbers. Then we'll diverge to look at conventional polynomials, and then show how the term is used obliquely to use a wonderful, seemingly invented, technique of manipulating binary data using a mathematical notation that is broadly related to more conventional polynomials.

Then the two ideas are merged to show how what always seems to be made so complicated in the text books can be reduced to nothing more than shift registers and lookup tables. Something any microprocessor or gate array can handle. To keep things as simple as possible, only the one polynomial used for Q65 is used in the descriptions that follow. This has been selected as just one of an almost infinite number that can be chosen to work with.

Galois Fields and Prime Numbers

If you look up "Galois Field"(GF) using any search engine a description will appear; something along the lines of the Wikipedia entry "*In mathematics, a finite field or Galois field is a field that contains a finite number of elements. As with any field, a finite field is a set on which the operations of multiplication, addition, subtraction and division are defined and satisfy certain basic rules.*" Multiplication and addition certainly apply to the set of all real numbers, but that is not a closed set – one in which only a set of finite unique values ever occur. Real numbers go on to plus and minus infinity, so the rule about " *finite number of elements* " doesn't hold. What does work to give us a valid GF is modulo arithmetic, to the base of a prime number.

Modulo arithmetic is what is sometimes referred to as clock arithmetic, where you count from 0 to 23 hours then back to 0. (We'll use the 24 hour clock as the 12 hour clock inconveniently goes from 1:00 to 12:00). Adding 14 hours onto 12:00 ends up at 02:00 (early the next day). Staying with this 24 hour clock notation, 15:00 + 23 = 14:00 , so 23 behaves effectively as if it were minus 1 hour. We can say in this notation that -1 = 23 (mod 24) and 15 + 20 = 11 (mod 24). Multiplication follows similar rules.

Normally 12 * 2 = 24, but using modulo 24 arithmetic, 12 * 2 = 0 (mod 24), which can be interpreted as two twelve hour stints taking a whole 24 hour day. Also 15 * 3 = 21 (mod 24)

24 is not a prime number, it can be divided by 2 and 3, so let's now do the arithmetic modulo 13 which is prime. We start with a finite field that fulfils one of the requirements for a GF, the whole numbers 0,1,2,3,4,5,6,7,8,9,10,11,12. If we try adding and multiplying any elements of this set of numbers modulo 13, we always end up with another number within the set. So we have a closed set of numbers, 0 – 12, upon which the arithmetic modulo 13 always gives an answer within the same set. This fulfils the other requirements for a GF. A multiplication table modulo 13 appears in **Table 1** for illustration, where it can be seen that all numbers from 0 to 12 appear as a product somewhere. The same applies to addition.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	2	4	6	8	10	12	1	3	5	7	9	11
3	0	3	6	9	12	2	5	8	11	1	4	7	10
4	0	4	8	12	3	7	11	2	6	10	1	5	9
5	0	5	10	2	7	12	4	9	1	6	11	3	8
6	0	6	12	5	11	4	10	3	9	2	8	1	7
7	0	7	1	8	2	9	3	10	4	11	5	12	6
8	0	8	3	11	6	1	9	4	12	7	2	10	5
9	0	9	5	1	10	6	2	11	7	3	12	8	4
10	0	10	7	4	1	11	8	5	2	12	9	6	3
11	0	11	9	7	5	3	1	12	10	8	6	4	2
12	0	12	11	10	9	8	7	6	5	4	3	2	1

Table 1 Multiplication (mod 13)

Any number multiplied by zero gives zero, and any number multiplied by 1 remains unchanged. Any number added to zero gives itself. These identities are absolutely fundamental to arithmetic and must be catered for, even if the mechanics behind the calculations means they have to be separately dealt with. We will come back to this odd-sounding statement later, where it matters.

Had the modulo not been prime, multiplication would not give a closed set. For example, using MOD 24 any number multiplied by 2 would give an even result. 1, 3, 5 etc. would then never appear in that row or column since 2 is a factor of 24.

Forward Error Correction Using Primes

A simple FEC process can be visualised if we consider a variation on the basic Hamming code, replacing single parity bits with values taken from a GF generated by modulo arithmetic. In the simple (7,4) binary Hamming code, three parity bits are used to check over four data bits, with each of the parity bits checking a different subset of two or three of the data bits. Any transmission error that affects a single bit out of the group of seven can be identified by testing the received parity. If the error is in one of the subsets checked by a particular parity bit, that check of the received corrupted data will show an error. By looking at which parity checks fail, and which remain valid the position of the bit error can then be determined.

Parity checking is simply modulo 2 addition, so it doesn't take much of a leap in imagination to visualise the process being replaced by symbols taken from a GF(N), with parity symbols calculated modulo N across various subsets of source symbols. This, in a very crude way, constitutes Reed-Solomon encoding which is capable of correcting errors in complete transmitted symbols consisting of several bits at a time. The process just described uses only modulo addition, so the need for a GF is not essential; this simple FEC works with any modulo value, not just primes, although more complex error correction does use multiplication, so the prime requirement is needed.

Modulo arithmetic restricted to prime numbers in not very convenient when it comes to binary and digital computing, so having described what a GF can be in practice, we will diverge into another area altogether that does eventually allow a similar closed set with more useable numbers.

Polynomials

First of all, we need to look at what a conventional polynomial is in normal arithmetic, and how they can be manipulated, before looking at a special adaptation used for binary digital working.

A polynomial consist of a sum of powers of a variable, X is used here, and usually is equated to zero. It is conventionally used to represent a particular shape or curve on a graph for example. If we take a variable, X, it can be multiplied by itself, squared, cubed, raised to the fourth or higher powers to give terms of the form X^2, X^3 , X^4 etc. Each power of X can be multiplied by a coefficient and the polynomial consists of the terms of the different powers added together. We'll keep it simple here and choose an almost-arbitrary one with a set of coefficients and powers of X just up to X squared. So we might choose $2.X^2 - 16.X + 30 = 0$ as an example. It is conventional to specify polynomials to equate to zero as shown, but that equation could equally well be written ($2.X^2 - 16.X = -30$) or ($2.X^2 = 16.X – 30$). They all mean the same thing.

This particular equation works if we choose values for X equal to 5 or 3, since $2 * 5^2 – 16 * 5 + 30 = 0$ and $2 * 3^2 – 16 * 3 + 30 = 0$. To save remembering how to solve quadratic equations it is easier to see the result if this 'second order' polynomial is expressed as $(2.X – 10) . (X – 3)$. Which introduces the concept of factoring polynomials; here we have $2.X^2 - 16.X + 30 = (2.X – 10).(X – 3)$ and it is easy to see how plugging in X = 3 or X = 5 will both give a result of zero.

Polynomials are multiplied by separately multiplying-out each term in one by each term in the other, then summing the results for each power. For example, multiplying the two polynomials $(X^2 + X + 1)$ and $(X – 1)$ is done by multiplying each term within each set of brackets separately and summing each component that results in equal powers. Multiplying the contents of the first set of brackets term by term by the first term of the second, X, we get $X^2.X + X.X + 1.X$, then by the second term which is -1 to get $-X^2 – X – 1$. Summing the two results breaks up into $X^2.X = X^3$; $X.X = X^2$ in the first term which cancels the $-X^2$ in the second one, and 1.X cancels the -X, so the result simplifies to $(X^3– 1)$

So we can say that the polynomial $(X^3– 1)$ has the factors $(X^2 + X + 1)$ and $(X – 1)$. Neither of these two factors can themselves be broken into smaller polynomials – at least they can't if we want to work with real numbers and stay away from complex roots. So in a sense we can regard these two factors as prime polynomials, whereas $(X^3 - 1)$ is not prime.

We can also have powers of polynomials, for example

$(X + 1)^3 = (X + 1).(X + 1).(X + 1) = X^3 + 3.X^2 + 3.X + 1$

And we add them by summing coefficients of each power in turn, eg.

$(X^2 + X + 1) + (X – 1) = X^2 + X + 1 + X – 1 = X^2 + 2.X = X(X + 2)$.

So we see how combining polynomials by addition and multiplication will generate new polynomials that may, or may not, be capable of being factorised.

Binary Arithmetic

The polynomials just described use real numbers, in the real World, where addition and multiplication follow the normal rules we love and understand. But we now are going to enter the somewhat contrived World of binary mathematics where the rules are bent to work modulo 2 and we have just the digits 0 and 1 to work with. So 0 + 0 = 0, 0 + 1 = 1 and 1 + 1 = 0. (Imagine a clock face with just two digits on it, opposite each other, so a single step goes from 0 to 1, then a second step, adding one, goes back to zero). There are also only three options for multiplication, 0 * 0 = 0, 0 * 1 = 0 and 1

* 1 = 1. Since we are working modulo 2, subtraction and addition are equivalent, 1 + 1 = 0 but also 1 – 1 = 0 and 0 – 1 = 1

In logic gate terms these are equivalent to an Exclusive-OR (XOR) operation for addition and an AND operation for multiplication.

Here we are not just using binary representation of real numbers as would be done in a standard computer chip doing calculations for us, so we do not perform conventional binary arithmetic with carry from one digit to the next. Instead, we treat a string of binary digits like '**1000011**' as if they were coefficients of a polynomial , then do all our arithmetic modulo 2, digit by digit with no carry just as we did above when adding polynomials. The right-hand-most digit takes the power zero, ie X^0 which is just the number 1 or 0.

The binary string just quoted is equivalent to $1.X^6 +0.X^5 +0.X^4 +0.X^3 +0.X^2 +1.X^1 +1 = (X^6 + X + 1) = 0$ which as illustrated we represent by '**1000011'**. Recall that addition and subtraction are exactly the same operation MOD 2, so therefore -X = X and -1 = 1 in modulo 2 arithmetic. This means that we can rewrite $X^6 + X + 1 = 0$ as $X^6 = X + 1$, since adding X + 1 is exactly the same as subtracting it when working modulo 2.

Shift Register Representation

We have said that a polynomial can be represented as a string of binary digits corresponding to the power of each term, so let's store those digits representing that polynomial in a shift register; seven flip-flops (for now) holding X^6 to X^0 and whose contents can shift right or left.

1	0	0	0	0	1	1

The rightmost digit corresponds to X^0 and the leftmost to X^6 so we can say that a shift left is equivalent to multiplying by X, or incrementing the power of each term by one. X^0 becomes X^1 , X^1 becomes X^2 etc. Now look at the statement above where we show that $X^6 = X + 1$. This tells us that the leftmost term (X^6) is equivalent to the two right hand cells, (X + 1) and that they correctly cancel each other out to give the zero sum for the polynomial.

Now, we'll make a declaration that our shift register is only 6 bits long, and we will throw away the left most digit. It is of no further use since we know it to be equal to the two '1's on the right hand side and can therefore be replaced by adding them in.

Now we'll start from scratch and preload the shift register with a starting value of '**000001**', unity, otherwise known as the identity element X^0 which when multiplied by anything does not change it. We will shift this left repeatedly, increasing the power by one, or multiplying by X each time, and take a note of each digit as it pops out on the left-hand side. Each digit that pops out is X^6 and we know that for this polynomial it is equal to X + 1. So for every '1' that appears on the left, we add in its equivalent '000011' or X+1 to the shift register, remembering that adding is the equivalent of the XOR operation. The table below shows the first 13 operations with a '1' popping out on the left shift represented by an arrow. Its equivalent, X+1 or '000011', is then added modulo 2 to the shift register contents. For example, if the contents before shifting are '**100011**' (as in the penultimate line of the table), then one left shift gives a '1' shifted out and '000110' remaining in the shift register. The '1' shifted out (X^6) means we then need to XOR '000011' (corresponding to X + 1) with the new contents of '000110' A bit-by-bit XOR operation results in '**000101**'. As in the final row

Note how a unique binary pattern builds up in the six registers after each shift. The adding-in of '000011' for each new occurrence of X^6 popping out is usually referred to as dividing the shift register contents by the polynomial '1000011'. Text books frequently refer to polynomial division and this is all it really means. In many areas the polynomial being divided is not just based on the left-most result after a shift, but can be made up of certain registers of the contents XORed together. This is the basis of Cyclic Redundancy Checks (CRCs) and other processes, but we digress.

	X^5	X^4	X^3	X^2	X^1	X^0
	0	0	0	0	0	1
	0	0	0	0	1	0
	0	0	0	1	0	0
	0	0	1	0	0	0
	0	1	0	0	0	0
	1	0	0	0	0	0
←	0	0	0	0	1	1
	0	0	0	1	1	0
	0	0	1	1	0	0
	0	1	1	0	0	0
	1	1	0	0	0	0
←	1	0	0	0	1	1
←	0	0	0	1	0	1

It is usual to adopt a shorthand representation of the shift register contents instead of writing out the patterns of ‘1’s and ’0’s. Although not being used in any conventional numerical manner it is convenient to represent, or refer to the six bits in the register, as if they were a binary number from 0

01, 02, 04, 08, 16, 32, 03, 06, 12, 24, 48, 35, 05, 10, 20,
40, 19, 38, 15, 30, 60, 59, 53, 41, 17, 34, 07, 14, 28, 56,
51, 37, 09, 18, 36, 11, 22, 44, 27, 54, 47, 29, 58, 55, 45,
25, 50, 39, 13, 26, 52, 43, 21, 42, 23, 46, 31, 62, 63, 61,
57, 49, 33

Which then repeats. The first 13 values underlined can be seen to be the bit patterns shown in the table. Note that the all-zeros case cannot be allowed to be part of the set as this would never change for any number of shifts.

Remember that these are just numbers, in decimal, representing the contents of the binary string and that their actual numerical value has no direct meaning for calculation.

A Full Set, and a GF

Apart from the all-zeros case we appear to have generated a unique set of 63 numbers that when manipulated by addition and shifting keep within the same set. We have already defined addition to be a straight XOR operation for each of the six bits with no carry, so it is clear that we can never exceed a pattern represented by 63 as that would then need a seventh bit. Two identical values when added, or XORed, will give the all zeros pattern, so we know that ‘000000’ must also be part of the set, even if it cannot come from the generating process that gives the rest of the numbers. We now have a closed set of results for addition, with 64 different binary values represented by 0 – 63.

But what about multiplication? We have seen that polynomial multiplication is a tedious, long-winded process multiplying out each term, gathering together similar weighted values and summing them. Even in binary, with values of just ‘0’ and ‘1’, multiplication is going to be a tricky process. Furthermore, as we’ve restricted ourselves to using just the lowest six bits of any result, a multiplication process that could potentially give terms up to X^{10} from the product of two X^5 terms needs to be catered for. This is done by noting that every occurrence of X^6, or ‘1000000’ can be replaced by ‘000011’, X^7 by ‘000110’ and so on. So as each bit higher than X^6 shifts out and is lost, we can add in the appropriate power of ‘000011’ to replace it. As we’ve chosen a ‘prime’ polynomial to work with, we once again end up getting each and every one of our 63 values (excluding zero). We’ve already declared that multiplication of any value by zero gives zero is an absolute rule, so provided

this is catered for explicitly in the multiplication process we now have a closed set of numbers under multiplication and addition. Which meets the requirements for a Galois Field. But, while addition is straightforward as a 6-bit XOR operation, how can we do multiplication efficiently? The answer is 'by logs'.

Logarithms of Polynomials

We return to the shift register generation process and start with the '000001' value preloaded. One shift left gives, '000010', two shifts '000100', 12 shifts gives '000101' (the last row shown in the table) and so on up to 63 left shifts generating binary '100001', or 33. What this means in practice is that for the first shift, the polynomial is raised to the power 1, for the second to power 2 and so on up to $poly^{63}$ = 33 as the last term.

If any value is raised to a power, then that power can be considered to be the logarithm of the result. To illustrate, in conventional decimal arithmetic, 10^5 = 10000 so the log (to the base 10) of 10000 is therefore 5. Adding the logarithms of two numbers, then taking the antilog of the sum, is the equivalent of multiplying those two numbers. A process familiar to scientists, engineers and mathematicians before the era of electronic calculators.

A bit of lateral thinking suggests there is no reason the process cannot be applied to polynomial representations, so there is no real reason we can't use the same terminology for polynomials restricted to a truncated width. LOG($poly^N$) = N.

We can generate a set of values of LOG(N) for each value of N by rearranging the sequence generated above, repeated here for convenience. We will refer to this as the ANTILOG table as it lists the result of successively raising the order of the polynomial by one each time.

Antilog Table for $X^6 = X+1$

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
01,	02,	04,	08,	16,	32,	03,	06,	12,	24,
48,	35,	05,	10,	20,	40,	19,	38,	15,	30,
60,	59,	53,	41,	17,	34,	07,	14,	28,	56,
51,	37,	09,	18,	36,	11,	22,	44,	27,	54,
47,	29,	58,	55,	45,	25,	50,	39,	13,	26,
52,	43,	21,	42,	23,	46,	31,	62,	63,	61,
57,	49,	33							

We need to generate the inverse of this, the LOG table. To generate the LOG table, step through the ANTILOG Table, starting with zero as the index of the first entry. For each digit note its value V, and its position in the table P. In the LOG table, place P in the position given by the corresponding V, again starting with the first index corresponding to zero. The generating ANTILOG table starts at position 0 which contains 01, so place a zero (it's position) at position 1 (its value) in the LOG table. Position 4 in the ANTILOG table contains the value 16, so the 16^{th} entry in the LOG table contains 4. Position 19 in the source contains the value 30, so the 30^{th} position of the new table has to contain the value 19, and so on for all 63 entries. Since zero does not occur anywhere in the ANTILOG table, the first or zeroth term of the LOG table cannot exist, so here it is shown as -1 just to act as a sort of filler. This is a parallel to real life where LOG(0) corresponds to minus infinity and is undefined. The reference position can never be called up so its actual value listed in the table is immaterial, but note that because of this there are only 63 'genuine' table lookup values to be had.

Log Table for $X^6 = X+1$

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
-1,	00,	01,	06,	02,	12,	07,	26,	03,	32,
13,	35,	08,	48,	27,	18,	04,	24,	33,	16,
14,	52,	36,	54,	09,	45,	49,	38,	28,	41,
19,	56,	05,	62,	25,	11,	34,	31,	17,	47,
15,	23,	53,	51,	37,	44,	55,	40,	10,	61,
46,	30,	50,	22,	39,	43,	29,	60,	42,	21,
20,	59,	57,	58						

To multiply two polynomials expressed by their equivalent reference values, we first of all check to see if one of them is zero. If so, no further action is required and we set the result of the multiplication to be zero. Likewise, if one of the input values is 1 the result can quickly be set to be the other value. For other numbers, the procedure is as follows.

Look up in the Log Table, the values at the location for each of the two inputs. These are the logarithms of the two polynomials to be multiplied. For example, multiply the two polynomials represented by the numbers 42 and 13. The 42nd position in the LOG table contains 53, and the 13th position contains 48. Add these to get 101. Because the LOG table only contains 63 genuine values, due to zero being undefined, this sum needs to be taken modulo 63. 101 MOD 63 = 38. In the antilog table lookup the value at position 38 which is 27. As we're adding two logs to get the log of a product by progressing though a table, the sum of the two logs is formed using normal arithmetical addition rather than the XOR process specific to adding polynomials.

We now have the answer that the product of the two polynomials represented by 42 and 13 = 27. Or '101010' * '001101' = '011011'

The result can be confirmed using software adapted from the WSJT-X encoding process for Q65 **[3],** or directly by long multiplication, term by term.

Multiplying From First Principles

Using the two numbers from the previous section, the second one, 13 or binary '001101' corresponds to $X^3 + X^2 + 1$ so to do the multiplication term by term (as in any classic long multiplication) the first number 42, or '101010', needs to be raised to its second and third powers as defined in bits 2 and 3 of the second number, ie. two and three left shifts, the results added modulo 2 and then added again to itself for the X^0 or unity bit. The table below, generated in the same way as before, replacing every shifted out X^6 by adding in '000011' shows how the three terms are formed; the three rows to be summed are highlighted.

	1	**0**	**1**	**0**	**1**	**0**	* X^0
←	0	1	0	1	1	1	* X^1
	1	**0**	**1**	**1**	**1**	**0**	* X^2
←	**0**	**1**	**1**	**1**	**1**	**1**	* X^3

* X^0	=	'101010'	
* X^2	=	'101110'	
* X^3	=	'011111'	
Sum	=	'011011'	= 27 - The same value as obtained using logs.

Based on the polynomial $X^6 = X + 1$ in GF(64) we have illustrated two ways in which the product [42] * [13] = [27]. Square brackets are used here to indicate the numbers are representations of a binary string rather than numbers in their own right,

Choice of Polynomial.

At the start, the irreducible (prime) polynomial $X^6 + X + 1 = 0$ was selected as our choice of generator, and because we are working in modulo 2 this is the same as writing $X^6 = X + 1$. There is a good reason for this particular one to have been selected, although any other irreducible polynomial of the same order could have been used. Had another been selected, the result would still be a field of 64 values in the source or antilog table, but they would have been in a completely different order. They would have been isomorphs of each other and all results and intermediate values would have been different. An isomorph is where the same set of numbers all occur in a totally different order, but still a perfectly valid Galois field, just as useable as the one chosen. Because the result is a set of 64 binary values, represented by the numbers 0 – 63, this is called a Galois Field of 64, or GF(64).

The particular polynomial is the one selected for the error correction employed in the Q65 mode within the WSJT-X suite of datamode software. It is simple, there are only three terms in it to think about. In the encoding process for Q65 groups of six bits at a time are assembled into 6-bit symbols which are manipulated using a lookup table. Six bit symbols means 64 possible values so a Galois field of 64 entities is adopted. GF(64) multiplications and additions of the source symbols give more 6 bit parity symbols. These are combined to generate the transmitted information symbols.

The detailed encoding process for Q65 that gives a very powerful Forward Error Correction (FEC) is not relevant here, but a description of it can be found at **[1]**. The equations used make use of both addition and multiplication in GF(64) and the Fortran-90 software contains the two tables listed here, with names of 'GF64Log' and 'GF64Antilog'. How the actual FEC works to optimally correct received errors on a noisy link is a highly complex process and interested readers should refer to **[2]**.

Summary

We have described what is meant by a Galois Field, a set of values, entities or numbers, that is closed for the operations of addition and multiplication. Arithmetic modulo a prime number fulfils these criteria, but for numbers other than primes a more complicated solution has to be found to obtain a GF.

We looked at polynomials, how these are multiplied and otherwise manipulated and defined a 'prime', or irreducible polynomial. We then showed that when using binary values, modulo 2 polynomials could be rewritten and simplified by virtue of addition and subtraction modulo 2 being the same operation. By representing binary polynomials as bits stored in a shift register, we illustrated how repeated shifting left is the same as raising the power of the polynomial by one element. By adopting a prime polynomial we then showed how each time the left shift caused an overflow it was replaced by lower order bits for a simplified version. The overall result was to generate a unique set of values, that when their bit patterns were represented as digits they formed a closed set.

Finally we showed how that generation process of raising the polynomial power by one each time is akin to adding one to the logarithm, meaning the generated table becomes a table of powers, or an antilog table. By rearrangement this can be turned into a log table, where the effective logarithm of a particular set of coefficients for the given polynomial can be looked up.

By adding logs then looking up the antilog of the result we have effectively multiplied the two polynomials defined by their coefficients expressed as binary digits. The resulting closed set of values under multiplication and addition as defined allow it to be declared as a Galois Field in 2^N. Meaning that they may be used wherever arithmetic in such a closed field is needed. Typically in the areas of error correction and coding, CRC generation and checking and in cryptography.

We have restricted ourselves to just one polynomial for demonstration, the sixth-order one used for Q65 encoding. Other irreducible polynomials are equally valid, and there is no need to stay with GF(64) symbols. Any value, GF(2^N) is possible using binary arithmetic under the same type of operation.

References

[1] http://g4jnt.com/Q65_CodingProcess.pdf

[2] http://www.eme2016.org/wp-content/uploads/2016/08/EME-2016-IV3NWV-Presentation.pdf

[3] Software '***GF64MultTest***', PowerBasic source code, and as a compiled .EXE for Windows, is available by request.