Error Detection and Correction Using the BCH Code

Hank Wallace

The Information Environment

The information revolution is in full swing, having matured over the last thirty or so years. It is estimated that there are hundreds of millions to several billion web pages on computers connected to the Internet, depending on whom you ask and the time of day. As of this writing, the google.com search engine listed 1,346,966,000 web pages in its database. Add to that email, FTP transactions, virtual private networks and other traffic, and the data volume swells to many terabits per day. We have all learned new standard multiplier prefixes as a result of this explosion. These days, the prefix “mega,” or million, when used in advertising (“Mega Cola”) actually makes the product seem trite.

With all these terabits per second flying around the world on copper cable, optical fiber, and microwave links, the question arises: How reliable are these networks? If we lose only 0.001% of the data on a one gigabit per second network link, that amounts to ten thousand bits per second! A typical newspaper story contains 1,000 words, or about 42,000 bits of information.
Imagine losing a newspaper story on the wires every four seconds, or 900 stories per hour. Even a small error rate becomes impractical as data rates increase to stratospheric levels as they have over the last decade.

Given the fact that all networks corrupt the data sent through them to some extent, is there something that we can do to ensure good data gets through poor networks intact?

Enter Claude Shannon

In 1948, Dr. Claude Shannon of the Bell Telephone Laboratories published a groundbreaking work entitled “The Mathematical Theory of Communication.” The meat of this work, which I shall summarize in a moment, has allowed the development of communications systems that transmit data effectively with zero errors. This is remarkable considering the intuitive understanding we have of the difficulty of communication, even in speech. Noise, distractions, simultaneous information sources, conflicting information, and other problems degrade our communications with each other on a personal level. The proposition that there could be, in some sense, perfect communication is considered absurd, especially between genders.

To examine Shannon's ideas and conclusions, we need some very basic definitions. First, a *communications channel* is a pipe of some sort through which we send information. If the information is smoke signals, then the channel is visual. If the information is audio, then the channel may be wired, as with the telephone. If the information is text and pictures, then the channel may be a computer network, like the Internet.

Now to anyone using a dial-up Internet connection, the notion of *channel capacity* is a native one as well. Channel capacity is the maximum amount of data that can be pumped through the channel in a fixed period of time. I have conversations regularly with non-technical friends about the speeds of our modem links, something that did not happen ten years ago. We even use the correct terminology, saying that “my service provider routinely runs at 40 kilobits per second.”

To that let's add the notion of *information source*. For example, every web page viewed comes from a web server, an information source. That web server can produce data at some rate, say, ten megabits per second. Unfortunately, most of us don't receive data that fast because the communications pipe is just too small, in terms of bits per second. However, every source has some information rate, measured in bits per second.

We also intuitively understand the effects of *noise* on communication. Whether several people speaking at once, or rushing wind in an automobile, noise makes communication more difficult, causing us to pause to consider and decode the sounds we hear, or even to ask for a repetition from the speaker.

Noise is present on communication networks, too, and comes in several forms. It can be man-made, but noise is generated even by the jostling of atoms in matter, at least all matter above absolute zero (-273.15 °C). Therefore, noise is a fundamental fact of life that we must deal with.

Thanks to the Internet, most high school graduates understand these basic definitions. However, Dr. Shannon discovered long before the Internet that information transmission and noise and communications channel capacity have a very special relationship. Quoting his central result:

Let a discrete channel have the capacity C and a discrete source the entropy per second H. If $H \leq C$ there exists a coding system such that the output of the source can be transmitted over the channel with an arbitrarily small frequency of errors.[1]

For the sake of our discussion, we will consider entropy per second to be bits per second of information transmitted. The word “discrete” refers to information transmitted in symbols, such as letters via teletype, or dots and dashes via Morse Code; digital information.

Shannon’s is rather a remarkable conclusion. It says in common terms that if our information source is sending data at a rate less than what the communications channel can handle, then we can add some extra bits to the data stream to push the error rate down to an arbitrarily low level!

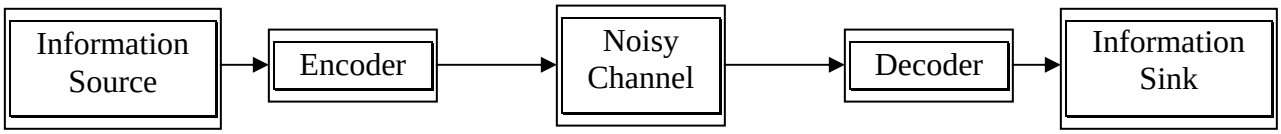
What is the down side? What is the trade-off? It is that communications delay is increased as we achieve lower and lower data error rates. However, for every situation, there are enough choices of channel coding that there is likely some satisfactory compromise between delay and error performance.

Since this theory is only a little over 50 years old, it is relatively new in the scheme of great discoveries. We think of Newton's Laws, Maxwell's Equations, and Einstein's Relativity as groundbreaking, enabling sorts of advances. This theory of communication is of that order.

Shortly after the publication of Shannon's work, many engineers and mathematicians got to work finding out how to create arbitrarily good communications links. Some of those techniques are considered in this paper.

The Basic Structure of a Communications System

As described briefly in the previous part, the basic structure of a communications system is presented in this diagram.



The information can of course be anything representable in symbols. For our purposes, this information should be representable as a string of binary digits. The purpose of the communications system is to convey the information from one point to another with no degradation.

The noisy channel adds noise to the information without our consent, corrupting the information to a degree related to the character and strength of the noise. The detailed behavior of communications systems in the presence of noise is a lengthy study in itself, and will not be pursued further in this short paper, except to state the limits of our coding methods.

We are concerned mostly with the encoder and decoder blocks. These blocks implement the “coding system” spoken of by Shannon, adding some extra bits in the encoder, and removing them again in the decoder. The decoder detects errors, corrects errors, or a combination of both. If we have a good design, the information on the input and output will be identical.

Not surprisingly, there are numerous coding systems. Each has a suitable area of service. This investigation targets only the most basic of coding systems, so our assumptions must match. We are assuming that we are transmitting symbols of only binary bits; that the channel adds a purely randomly determined amount of noise to each bit transmitted; and that each bit is independent of the others (the channel is memoryless). This is called a binary symmetric channel. Lin and Costello have an excellent introduction to this concept [2].

We will develop a random error correcting code here called the BCH code. This code handles randomly located errors in a data stream according to its inherent limitations. Other coding systems are optimized to correct bursts of errors, such as the Reed-Solomon code used in compact discs. Errors induced in a compact disc are more likely to damage a bunch of bits together, as a scratch or finger smudge would do.

On top of these coding systems, other techniques are used to ensure data accuracy. These are only mentioned here, as they are full studies alone. Typically, if data errors are detected on a message in a network, the receiving station requests a retransmission from the sender. This is called automatic repeat request, or ARQ. Even if error correction codes are used, usually another detection-only code is used on data messages to check the results of the correction. If that test fails, a retransmission is requested. On top of that, systems may use a combination of burst and random

error correcting codes, gaining benefits from each method at the cost of added complexity. From this brief overview, the reader can see that we are only touching the surface.

To the Math

We are slowly working our way into the details of the BCH error detection and correction code. Now we start on the fun part, the math. I suggest that you review or obtain a basic text on abstract algebra, also called modern algebra, as it is the foundation of our work here.

Regarding a coding system to reduce errors, Shannon says that it can be done, but not how. That is the curse of existence theorems. However, many have gone before us and laid the foundations of error control coding. Let's explore those foundations.

Transmission on computer and radio networks uses binary symbols. For example, the American Standard Code for Information Interchange defines the letter "A" as corresponding to a binary value of 1000001. The letter "B" is 100010, and so on. Each letter, number, and symbol on the typical personal computer keyboard has its own numeric value. A sentence is then represented by a string of these values: "ABC" = 1000001, 1000010, 1000011. If we desire to detect errors in the transmission of such a string of ones and zeros, then it is a good design decision to make a coding system that works with binary symbols.

However, what type of system shall we choose? We anticipate that encoding and decoding data will not be trivial, so we will need the most capable system available. From our study of abstract algebra (see any basic text on the subject), we know that the most flexible, yet still basic system of rules is that of the field. In a field, we can add, subtract, multiply and divide.

What field do we choose? Our binary symbols are strings of ones and zeros taken from $\mathbf{Z}_2$ (the field of binary numbers). Perhaps we could use it. However, it only has two digits, so if each character is to be represented in the field, we will need a bigger field.

Well, the order of (number of elements in) every finite field is p^m for some prime p and integer $m > 0$, so we are stuck with the power of a prime number of elements in our field. That is coincidentally good, however, because digital data is transmitted typically in multiples of eight bits (2^3), so each message can be considered to have values from a field of 2^m bits, a power of a prime. Therefore, we conclude that $\mathbf{Z}_{2^m}$ is a good candidate for our field, with m to be determined.

Unfortunately, we soon learn that ordinary arithmetic in $\mathbf{Z}_{2^m}$ does not meet the criterion for inverses. For example, in $\mathbf{Z}_{2^4}$ (containing the integers 0…15), 2 has no inverse. That is, $2b \bmod 16 = 1$ has no solution b. (Remember that two field elements are multiplicative inverses if we get 1 when they are multiplied together.) Therefore, $\mathbf{Z}_{2^4}$ is not a field with standard multiplication as $\mathbf{Z}_p$ is.

Ordinary arithmetic fails because $\mathbf{Z}_{16} \cong \mathbf{Z}/\langle 16 \rangle$ ('≅' denotes isomorphism), and the generator of $\langle 16 \rangle$ is a composite number. $\mathbf{Z}_p \cong \mathbf{Z}/\langle p \rangle$ is a field for p a prime. We need a different operation, at least for multiplication.

That different arithmetic is polynomial arithmetic. We know that $\mathbf{Z}_2(\alpha) \cong \mathbf{Z}_2[x]/\langle p(x)\rangle$ where $p(x)$ is a minimal polynomial with root α. Thus, we have replaced the non-prime 2^m with a "prime polynomial," irreducible $p(x)$.

The isomorphism above tells us again that the required field exists. Now we have to build an example and see how it works. The following is a standard example of the construction of $\mathbf{Z}_{2^4}$ from the references. I will not go through all the details, but this is the basic outline.

We are constructing the extension field of $\mathbf{Z}_2[x]$, $\mathbf{Z}_2(\alpha)$. The isomorphism suggests that we need a minimal polynomial with a root in $\mathbf{Z}_{2^4}$. Since we want 16 elements in $Z_2(\alpha)$, the degree of the polynomial should be 4, so that $2^4 = 16$.

However, there is an additional condition. The polynomial must be primitive in $GF(2^4)$. That means that every element in $\mathbf{Z}_{2^4}$ must be expressible as some power of $p(x)$. With this limitation, checking the minimal polynomials of all the nonzero elements in $GF(2^4)$, we find two polynomials, $x^4 + x^3 + 1$ and $x^4 + x + 1$, that are primitive. Either may be used.

To generate the elements, we start with three initial elements of the extension field and perform all arithmetic modulo $p(x) = x^4 + x^3 + 1$. The initial elements are 0, 1, and α. Raising α to powers successively identifies α^2 and α^3 as members of the extension field. When we get to α^4, we realize that it does not represent an element of $\mathbf{Z}_{2^4}$, but we know that $p(\alpha) = 0 = x^4 + x^3 + 1$, so $\alpha^4 = \alpha^3 + 1$ (in a binary field addition is equivalent to subtraction). Thus, we reduce each power greater than three using the identity $\alpha^4 = \alpha^3 + 1$.

For example, when multiplying 1011 by 1101, the polynomial representation is

$$(x^3 + x + 1)(x^3 + x^2 + 1) = x^6 + x^5 + x^4 + 3x^3 + x^2 + x + 1$$

Reducing this using the identity, finally we see that $1011 \times 1101 = 0010$, or

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

By computing successive powers of α and reducing using the identity, we can build the entire field $\mathbf{Z}_{2^4}$. I have written a program to do the busy work, and it appears in Appendix A. Now that we have a field with all the proper operations, we may call it $GF(2^4)$ or GF(16), the Galois field with 16 elements.

Below appear the elements generated by both primitive polynomials. There are three forms of each element. The power form expresses the reality that each field element except zero is the power of the generating element, α, and is useful when multiplying field elements. The polynomial form is useful for adding field elements, and the binary values in the centers of the tables are merely the coefficient of these polynomials, with the highest power of α on the left. Notice that the two fields contain the same elements in different order.

Field of 16 elements generated by $x^4 + x + 1$		
Power Form	n-Tuple Form	Polynomial Form
0	0000	0
1	0001	1
α	0010	α
α^2	0100	α^2
α^3	1000	α^3
α^4	0011	$\alpha + 1$
α^5	0110	$\alpha^2 + \alpha$
α^6	1100	$\alpha^3 + \alpha^2$
α^7	1011	$\alpha^3 + \alpha + 1$
α^8	0101	$\alpha^2 + 1$
α^9	1010	$\alpha^3 + \alpha$
α^{10}	0111	$\alpha^2 + \alpha + 1$
α^{11}	1110	$\alpha^3 + \alpha^2 + \alpha$
α^{12}	1111	$\alpha^3 + \alpha^2 + \alpha + 1$
α^{13}	1101	$\alpha^3 + \alpha^2 + 1$
α^{14}	1001	$\alpha^3 + 1$

Field of 16 elements generated by $x^4 + x^3 + 1$		
Power Form	n-Tuple Form	Polynomial Form
0	0000	0
1	0001	1
α	0010	α
α^2	0100	α^2
α^3	1000	α^3
α^4	1001	$\alpha^3 + 1$
α^5	1011	$\alpha^3 + \alpha + 1$
α^6	1111	$\alpha^3 + \alpha^2 + \alpha + 1$
α^7	0111	$\alpha^2 + \alpha + 1$
α^8	1110	$\alpha^3 + \alpha^2 + 1$
α^9	0101	$\alpha^2 + 1$
α^{10}	1010	$\alpha^3 + \alpha$
α^{11}	1101	$\alpha^3 + \alpha^2 + 1$
α^{12}	0011	$\alpha + 1$
α^{13}	0110	$\alpha^2 + \alpha$
α^{14}	1100	$\alpha^3 + \alpha^2$

To examine why a primitive polynomial is needed to generate all the field elements, I ran the program using a non-primitive polynomial. Notice repetition in the elements. This is undesirable because not all the values in GF(16) are represented by the computations, and our ability to encode and decode messages using arithmetic in this set would be impaired or destroyed. (However, some families of codes do use non-primitive polynomials because it allows a greater range of selection of code lengths; the code can be tailored to the application.)

Field of 16 elements generated by $x^4 + x^3 + x^2 + x + 1$		
Power Form	n-Tuple Form	Polynomial Form
0	0000	0
1	0001	1
α	0010	α
α^2	0100	α^2
α^3	1000	α^3
α^4	1111	$\alpha^3 + \alpha^2 + \alpha + 1$
α^5	0001	1
α^6	0010	α
α^7	0100	α^2
α^8	1000	α^3
α^9	1111	$\alpha^3 + \alpha^2 + \alpha + 1$
α^{10}	0001	1
α^{11}	0010	α
α^{12}	0100	α^2
α^{13}	1000	α^3
α^{14}	1111	$\alpha^3 + \alpha^2 + \alpha + 1$

Mechanically, these field elements are easy to generate. Multiplication by element α results in a left shift of the previous binary value. If a one is shifted out of the fourth position (in GF(16)), that

constitutes a polynomial of fourth degree and the result is reduced by subtracting the generator polynomial from the result. However, subtraction in binary arithmetic is merely the exclusive OR function, so an encoder for field elements is easy to build in logic circuits.

Next time, we will look at various types of codes that arose over the years as a result of the investigations I have reproduced above.

Types of Codes

We are looking at the BCH error detection and correction code, going first through some preliminary background on the mathematical basis of the code.

Now that we have a field in which to do computations, what we need next is a philosophy for encoding and decoding data in order to detect and possibly correct errors. Here we are taking advantage of results discovered by the hard work of perhaps a hundred individuals just in the last 50 years.

There are several types of codes. The first major classification is linear vs. nonlinear. Linear codes in which we are interested may be encoded using the methods of linear algebra and polynomial arithmetic. Then we have block codes vs. convolutional codes. Convolutional codes operate on streams of data bits continuously, inserting redundant bits used to detect and correct errors.

Our area of investigation here will be linear block codes. Block codes differ from convolutional codes in that the data is encoded in discrete blocks, not continuously. The basic idea is to break our information into chunks, appending redundant check bits to each block, these bits being used to detect and correct errors. Each data + check bits block is called a codeword. A code is linear when each codeword is a linear combination of one or more other codewords. This is a concept from linear algebra and often the codewords are referred to as vectors for that reason.

Another characteristic of some block codes is a cyclic nature. That means any cyclic shift of a codeword is also a codeword. So linear, cyclic, block code codewords can be added to each other and shifted circularly in any way, and the result is still a codeword. You might expect that it takes some finesse to design a set of binary words to have these properties.

Since the sets of codewords may be considered a vector space, and also may be generated through polynomial division (the shifting algorithm, above), there are two methods of performing computations: linear algebra and polynomial arithmetic. We will dwell on polynomial arithmetic methods later in this paper.

Assume that we have a code that can detect *t* errors in a codeword. That means up to *t* errors can occur and the receiver will say with 100% certainty that the codeword contains errors. How does this work? What is the intuitive structure of these codewords in the field? (See the previous installment for a development of the mathematical field we are using for computations.)

Let us say that we transmitted one of the codewords generated previously in GF(16) by the polynomial $\alpha^4 + \alpha + 1$. If an error occurs, the result will be another codeword in this field. We have

no way of knowing exactly which bits were changed. That is because every possible bit pattern is a codeword.

However, if we used a larger field, say, GF(32), and then transmitted our four bit information words plus one check bit, half of the 32 codewords would be valid and the other half would not. If we received one of the invalid codewords, we could request a retransmission. That is exactly how parity check bits work, the simplest error detection system.

With the example above of the character “A”, binary 1000001, only seven bits are needed to represent the entire alphabet and several punctuation marks, 128 values in total. Adding an eighth bit, which is a parity sum of the other seven, enlarges the required field to 256 elements, with 128 of them representing valid information.

Then our letter “A” would be represented as 01000001, with the leftmost bit being an even parity bit. This bit is set or reset to give the entire binary word an even number of one bits. For the letter “C”, the value is 11000011, and the parity bit is 1 because there are three ones in the rest of the word, four total. If a character is received with a parity error, it is discarded. The odd parity scheme is equivalent in performance and effectively identical.

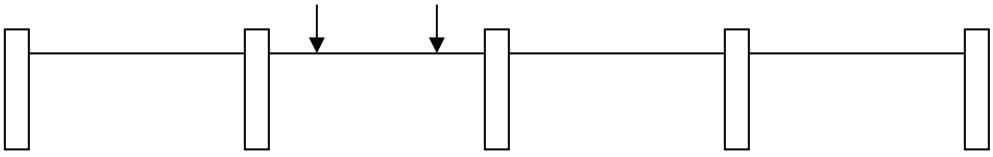
Now the parity bit is used in communications links today, and many chips have the capability of encoding and decoding it in hardware, flagging errant data words. However, it can only detect errors, and only those which change an odd number of bits. One may suspect that adding more check bits increases the error detection capability of the code, and that is right. Much effort has been expended to find powerful codes that can detect and correct a significant number of errors while still being easy to encode and decode.

Next time, we will look intuitively at how codewords are selected to allow error correction.

Codeword Selection Criteria

We suspect, however, that the selection of codewords in a more error tolerant coding system would have to be done by a deeper method of which the parity example above is only a special case. What criterion do we use to select codewords?

The criterion is related to the relative distances between objects in space, but a space of perhaps many dimensions. For example, when placing fenceposts on a one-dimensional line, the farmer spaces them evenly to maximize and equalize the support given to each part of the fence. Consider our codewords spaced along the fence.



The subset of our field elements which are valid codewords are the fenceposts. Two other invalid codewords are noted by arrows. Errors occur during communication change the codewords, moving them along the fence line. To maximize our chances of guessing the correct codeword in spite of

the errors, we need to likewise space our codewords evenly, as in the diagram. For that we use a concept called Hamming distance.

Hamming distance (or simply *distance*) is a very simple tool. To find the distance between two binary words, we just count the number of bits differing. For example, the distance between binary 01010011 and 01011100 is four, because the four rightmost bits differ. The distance between 10100 and 11001 is three bits.

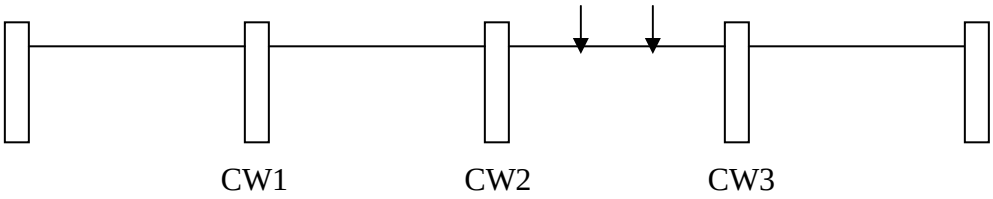
This computation can be performed easily by using the logical exclusive OR function:

```
    10100
xor 11001
    -----
    01101
```

The result has three ones, or we say a *weight* of three.

Given a requirement for 256 codewords in a larger set of 1024, for example, our task is to find a method of selecting the 256 valid codewords that maximizes the distance between them. Then, when errors occur, we can make a good estimate as to what correct codeword was transmitted. Each code has a minimum distance, called *d* that represents this value.

Now just from the minimum distance between the codewords we can draw a powerful conclusion. Consider the codewords arrayed on the fence again. If the distance between the codewords is three bits, then how many errors can we correct and be sure that we have not over corrected?



Looking at the diagram above, assume that codeword two has been transmitted. if one error occurs, it takes us one bit closer to codeword three, but we can see from the diagram that the obvious selection of the corrected codeword is still codeword two. If two errors occur, then the received codeword is closer now to codeword three than codeword two, and the decoder will select codeword three as the right one, which is a mistake.

In real life, this situation is multidimensional, with each codeword having many close neighbors in its field, distance-wise. But even this simple example suggests that the number of errors that we can correct is $t = (d\text{-}1)/2$, or half the distance, not including the middle bit for odd values of d. For $d = 3$, $t = 1$. For $d = 4$, $t = 1$ still.

Note that if we are not correcting errors, we can detect more than t. In the example, we could detect as many as two bit errors in either direction from codeword two. The number of detectable errors is in general d-1, because d errors would transform one codeword into another.

That brings up an important notion. A large number of errors in a codeword (*d* or more) can possibly transform a codeword into another valid, but unintended codeword. This situation is called an undetectable error. For example, if two bit errors occurred in the transmission of the letter "A" with parity bit (01000001), it could be mistaken for a "C" (11000011). To guard against this, communications engineers sometimes use an additional overall check code that tests the entire message for validity.

Two Linear Block Codes

The first code developed was the Hamming code, in 1950. It actually consists of a whole class of codes with the following characteristics[3]:

Block Length: $n = 2^m - 1$
Information Bits: $k = 2^m - m - 1$
Parity Check Bits: $n - k = m$
Correctable Errors: $t = 1$

These conditions are true for $m > 2$. For example, with $m = 4$, there are $n = 15$ total bits per block or codeword, $k = 11$ information bits, $n - k = 4$ parity check bits, and the code can correct $t = 1$ error. A representative codeword would be

10010100101 0010

where the four bits on the right (0010) are the parity checkbits. By choosing the value of *m*, we can create a single error correcting code that fits our block length and correction requirements. This one is customarily denoted a (15, 4) code, telling us the total number of bits in a codeword (15) and the number of information bits (4).

We omit the details of encoding and decoding the Hamming code here because such will be covered in detail for the BCH code, later.

The Golay code is another code, more powerful than the Hamming code, and geometrically interesting. This (23, 12) code was discovered by Marcel J. E. Golay in 1949 [4]. It may also be extended using an overall parity bit to make a (24, 12) code. The minimum distance is seven, so it can detect up to six errors, or correct up to $t = (7 - 1)/2 = 3$ errors.

The aspect of the Golay and Hamming codes that makes them interesting is the fact that they are *perfect*. With any code, the codewords can be considered to reside within spheres packed into a region of space. The entire space is $GF(2^m)$. Each sphere contains a valid codeword at its center and also all the invalid codewords that correct to the valid codeword, those being a distance of three or fewer bits from the center in the case of the Golay code ($t = 3$). If there are orphan binary words outside the spheres, then the code is termed *imperfect*.

Just how many codewords are in a sphere? With the Golay code, we first have the valid codeword. Then add the invalid codewords produced by introducing a single error in each of the 23 bits of the

valid codeword, $C(n, 1) = 23$. Add to that the invalid codewords produced by introducing two and three errors in the valid codeword, $C(n, 2) = 253$, and $C(n, 3) = 1771$. Adding these up, we see the sphere contains $2048 = 2^{11}$ words.

There are also $4096 = 2^{12}$ total valid codewords (and spheres) in the Golay code, so the sum of all the codewords in all the spheres is $2^{11}2^{12} = 2^{23}$, and that is the entire set of 23-bit binary words in $GF(2^{23})$. So there is no binary word in $GF(2^{23})$ that is not correctable to one of the 4096 valid codewords. That is a perfect code[5]. An imperfect code has some elements in $GF(2^m)$ outside of any such sphere, so a correction algorithm may not produce a useful result with such elements.

The abstract beauty of this structure is remarkable, but even more remarkable is the fact that perfect codes are rare. Pless and others have proven this fact, that the only nontrivial multiple error correcting perfect binary codes are equivalent to the binary Golay (23, 12) code [6]. This sweeping conclusion comes about from a result that states that a perfect binary (n, k) code that corrects t errors must have n, k, and t satisfy the following relationship:

$$2^k \sum_{i=0}^{t} \binom{n}{i} = 2^n.$$

The proof concludes that there are only a few values of n, k, and t that provide equality, indicating a perfect code. For the binary Golay code, the expression works out to:

$$2^{12} \left(\binom{23}{0} + \binom{23}{1} + \binom{23}{2} + \binom{23}{3} \right) = 2^{12} (1 + 23 + 253 + 1771) = 2^{23}$$

The binary Hamming codes are perfect as well, and there is a ternary Golay (11, 6) code with minimum distance 5 that is perfect. Aside from some other trivial codes that are of no practical interest (repetition codes decoded with majority logic gating), that is the extent of the perfect codes. One might suppose that there is some n-dimensional space where another perfect code exists, but that is not the case.

Main Event: The BCH Code

I have done some work in the past in real communications systems using the Golay code. It has worked well in those applications. Many other real-world systems use one of the BCH codes.

The BCH abbreviation stands for the discoverers, Bose and Chaudhuri (1960), and independently Hocquenghem (1959). These codes are multiple error correcting codes and a generalization of the Hamming codes. These are the possible BCH codes[7] for $m \geq 3$ and $t < 2^{m-1}$:

Block Length: $n = 2^m - 1$
Parity Check Bits: $n - k \leq mt$
Minimum distance: $d \geq 2t + 1$

The codewords are formed by taking the remainder after dividing a polynomial representing our information bits by a generator polynomial. The generator polynomial is selected to give the code its characteristics. All codewords are multiples of the generator polynomial.

Let us turn to the construction of a generator polynomial. It is not simply a minimal, primitive polynomial as in our example where we built GF(16). It is actually a combination of several polynomials corresponding to several powers of a primitive element in GF(2^m).

The discoverers of the BCH codes determined that if α is a primitive element of GF(2^m), the generator polynomial is the polynomial of lowest degree over GF(2) with α, α^2, α^3, … , α^{2t} as roots. The length of a codeword is $2^m - 1$ and t is the number of correctable errors. Lin concludes[8] that the generator is the least common multiple of the minimal polynomials of each α^i term. A simplification is possible because every even power of a primitive element has the same minimal polynomial as some odd power of the element, halving the number of factors in the polynomial. Then $g(x) = \text{lcm}(m_1(x), m_3(x), \ldots, m_{2t-1}(x))$.

These BCH codes are called primitive because they are built using a primitive element of GF(2^m). BCH codes can be built using nonprimitive elements, too, but the block length is typically less than $2^m - 1$.

As an example, let us construct a generator polynomial for BCH(31,16). Such a codeword structure would be useful in simple remote control applications where the information transmitted consists of a device identification number and a few control bits, such as “open door” or “start ignition.”

This code has 31 codeword bits, 15 check bits, corrects three errors ($t = 3$), and has a minimum distance between codewords of 7 bits or more. Therefore, at first glance we need $2t - 1 = 5$ minimal polynomials of the first five powers of a primitive element in GF(32). But the even powers’ minimal polynomials are duplicates of odd powers’ minimal polynomials, so we only use the first three minimal polynomials corresponding to odd powers of the primitive element.

The field we are working in is GF(32), shown below. This was generated using primitive polynomial $x^5 + x^2 + 1$ over GF(32).

Field of 32 elements generated by $x^5 + x^2 + 1$		
Power Form	n-Tuple Form	Polynomial Form
0	00000	0
1	00001	1
α	00010	α
α^2	00100	α^2
α^3	01000	α^3
α^4	10000	α^4
α^5	00101	$\alpha^2 + 1$
α^6	01010	$\alpha^3 + \alpha$
α^7	10100	$\alpha^4 + \alpha^2$
α^8	01101	$\alpha^3 + \alpha^2 + 1$
α^9	11010	$\alpha^4 + \alpha^3 + \alpha$
α^{10}	10001	$\alpha^4 + 1$
α^{11}	00111	$\alpha^2 + \alpha + 1$

α^{12}	01110	$\alpha^3 + \alpha^2 + \alpha$
α^{13}	11100	$\alpha^4 + \alpha^3 + \alpha^2$
α^{14}	11101	$\alpha^4 + \alpha^3 + \alpha^2 + 1$
α^{15}	11111	$\alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1$
α^{16}	11011	$\alpha^4 + \alpha^3 + \alpha + 1$
α^{17}	10011	$\alpha^4 + \alpha + 1$
α^{18}	00011	$\alpha + 1$
α^{19}	00110	$\alpha^2 + \alpha$
α^{20}	01100	$\alpha^3 + \alpha^2$
α^{21}	11000	$\alpha^4 + \alpha^3$
α^{22}	10101	$\alpha^4 + \alpha^2 + 1$
α^{23}	01111	$\alpha^3 + \alpha^2 + \alpha + 1$
α^{24}	11110	$\alpha^4 + \alpha^3 + \alpha^2 + \alpha$
α^{25}	11001	$\alpha^4 + \alpha^3 + 1$
α^{26}	10111	$\alpha^4 + \alpha^2 + \alpha + 1$
α^{27}	01011	$\alpha^3 + \alpha + 1$
α^{28}	10110	$\alpha^4 + \alpha^2 + \alpha$
α^{29}	01001	$\alpha^3 + 1$
α^{30}	10010	$\alpha^4 + \alpha$

We need first a primitive element. Well, α is a primitive element in GF(32). Next we need the minimal polynomials of the first three odd powers of α. Tables of minimal polynomials appear in most texts on error control coding. Lin and Costello, Pless, and Rorabaugh[9] exhibit algorithms for finding them using cyclotomic cosets. From Lin and Costello[10], the first three odd power of α minimal polynomials are:

α: $m_1(x) = x^5 + x^2 + 1$
α^3: $m_3(x) = x^5 + x^4 + x^3 + x^2 + 1$
α^5: $m_5(x) = x^5 + x^4 + x^2 + x + 1$

Therefore, $g(x) = \text{lcm}(m_1(x), m_3(x), m_5(x)) = m_1(x)\, m_3(x)\, m_5(x)$ (since these are irreducible).

So $g(x) = (x^5 + x^2 + 1)(x^5 + x^4 + x^3 + x^2 + 1)(x^5 + x^4 + x^2 + x + 1) = x^{15} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1$.

To encode a block of bits, let us first select as our information the binary word 1000001 for the letter "A" and call it $f(x)$, placing it in the 16-bit information field . Next, we append a number of zeros equal to the degree of the generator polynomial (fifteen in this case). This is the same as multiplying $f(x)$ by x^{15}. Then we divide by the generator polynomial using binary arithmetic (information bits are **bold**):

```
                                 ________________________________
1000111110101111)1000000010000010000000000000000
                 1000111110101111
                 ----------------
                     1101101011110000
                     1000111110101111
                     ----------------
                      1010101010111110
```

```
                  1000111110101111
                  ----------------
                    100101000100010
```

The quotient is not used and so we do not even write it down. The remainder is 100101000100010, or $x^{14} + x^{11} + x^9 + x^5 + x$ in polynomial form, and of course it has degree less than our generator polynomial, $g(x)$. Thus the completed codeword is

Information	Checkbits
0000000001000001	100101000100010

This method is called *systematic encoding* because the information and check bits are arranged together so that they can be recognized in the resulting codeword. Nonsystematic encoding scrambles the positions of the information and check bits. The effectiveness of each type of code is the same; the relative positions of the bits are of no matter as long as the encoder and decoder agree on those positions[11]. To test the codeword for errors, we divide it by the generator polynomial:

```
                  ________________________________
1000111110101111)0000000001000001100101000100010
                 1000111110101111
                 ----------------
                     1100100001111000
                     1000111110101111
                     ----------------
                      1000111110101111
                      1000111110101111
                      ----------------
                                     0
```

The remainder is zero if there are no errors. This makes sense because we computed the checkbits ($r(x)$) from the information bits ($f(x)$) in the following way:

$$f(x)\, x^n = q(x)\, g(x) + r(x)$$

The operation $f(x)\, x^n$ merely shifts $f(x)$ left n places. Concatenating the information bits $f(x)$ with the checkbits $r(x)$ and dividing by $g(x)$ again results in a remainder, $r'(x)$, of zero as expected because

$$f(x)\, x^n + r(x) = q(x)\, g(x) + r'(x)$$

If there are errors in the received codeword, the remainder, $r'(x)$, is nonzero, assuming that the errors have not transformed the received codeword into another valid codeword. The remainder is called the syndrome and is used in further algorithms to actually locate the errant bits and correct them, but that is not a trivial matter.

The BCH codes are also cyclic, and that means that any cyclic shift of our example codeword is also a valid codeword. For example, we could interchange the information and checkbits fields in the last division above (a cyclic shift of 15 bits) and the remainder would still be zero.

Decoding the BCH(31,16) Code

Determining where the errors are in a received codeword is a rather complicated process. (The concepts here are from the explanation of Lin and Costello[12].) Decoding involves three steps:

1. Compute the syndrome from the received codeword.
2. Find the error location polynomial from a set of equations derived from the syndrome.
3. Use the error location polynomial to identify errant bits and correct them.

We have seen that computing the syndrome is not difficult. However, with the BCH codes, to implement error correction we must compute several components which together comprise a syndrome vector. For a *t* error correcting code, there are 2*t* components in the vector, or six for our triple error correcting code. These are each formed easily using polynomial division, as above, however the divisor is the minimal polynomial of each successive power of the generating element, α.

Let $v(x)$ be our received codeword. Then $S_i = v(x) \bmod m_i(x)$, where $m_i(x)$ is the minimal polynomial of α^i. In our example,

$$S_1(x) = v(x) \bmod m_1(x)$$
$$S_2(x) = v(x) \bmod m_2(x)$$
$$S_3(x) = v(x) \bmod m_3(x)$$
$$S_4(x) = v(x) \bmod m_4(x)$$
$$S_5(x) = v(x) \bmod m_5(x)$$
$$S_6(x) = v(x) \bmod m_6(x)$$

Now in selecting the minimal polynomials, we take advantage of that property of field elements whereby several powers of the generating element have the same minimal polynomial. If $f(x)$ is a polynomial over GF(2) and α is an element of GF(2^m), then if $b = 2^i$, α^b is also a root of $f(x)$ for $i \geq 0$ [13]. These are called conjugate elements. From this we see that all powers of α such as α^2, α^4, α^8, α^{16}, … are roots of the minimal polynomial of α. In GF(32) which applies to our example, we must find the minimal polynomials for α through α^6. The six minimal polynomials are:

$$m_1(x) = m_2(x) = m_4(x) = x^5 + x^2 + 1$$
$$m_3(x) = m_6(x) = x^5 + x^4 + x^3 + x^2 + 1$$
$$m_5(x) = x^5 + x^4 + x^2 + x + 1$$

Next, we form a system of equations in α:

$$S_1(\alpha) = \alpha + \alpha^2 + \ldots + \alpha^n$$
$$S_2(\alpha^2) = (\alpha)^2 + (\alpha^2)^2 + \ldots + (\alpha^n)^2$$
$$S_3(\alpha^3) = (\alpha)^3 + (\alpha^2)^3 + \ldots + (\alpha^n)^3$$
$$S_4(\alpha^4) = (\alpha)^4 + (\alpha^2)^4 + \ldots + (\alpha^n)^4$$
$$S_5(\alpha^5) = (\alpha)^5 + (\alpha^2)^5 + \ldots + (\alpha^n)^5$$
$$S_6(\alpha^6) = (\alpha)^6 + (\alpha^2)^6 + \ldots + (\alpha^n)^6$$

It turns out that each syndrome equation is a function only of the errors in the received codeword. The α^i are the unknowns, and a solution to these equations yields information we use to construct

an error locator polynomial. One can see that this system is underconstrained, there being multiple solutions. The one we are looking for is the one that indicates the minimum number of errors in the received codeword (we are being optimistic).

The method of solution of this system involves elementary symmetric functions and Newton's identities and is beyond the scope of this paper. However, this method has been reduced to an algorithm by Berlekamp that builds the error locator polynomial iteratively. Using the notation of Lin and Costello[14], a $t + 2$ line table may be use to handle the bookkeeping details of the error correction procedure for binary BCH decoding. It is described next.

First, make a table (using BCH(31,16) as our example):

μ	$\sigma^{(\mu)}(x)$	d_μ	l_μ	$2\mu - l_\mu$
-½	1	1	0	-1
0	1	S_1	0	0
1				
2				
$t = 3$				

The BCH decoding algorithm follows.

1. Initialize the table as above. Set $\mu = 0$.
2. If $d_\mu = 0$, then $\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x)$. Let $L = l_{\mu+1}$.
3. If $d_\mu \neq 0$, then find a preceding row (row ρ) with the most positive $2\mu - l_\mu$ and $d_\rho \neq 0$. Then $\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x) + d_\mu d_\rho^{-1} x^{2(\mu-\rho)}\sigma^{(\rho)}(x)$. If $\mu = t - 1$, terminate the algorithm.
4. $l_{\mu+1} = \deg(\sigma^{(\mu+1)}(x))$.
5. $d_{\mu+1} = S_{2\mu+3} + \sigma_1^{(\mu+1)} S_{2\mu+2} + \sigma_2^{(\mu+1)} S_{2\mu+1} + \ldots + \sigma_L^{(\mu+1)} S_{2\mu+3-L}$. σ_i is the coefficient of the i-th term in $\sigma(x)$.
6. Increment μ and repeat from step 2.

At each step we are computing the next approximation to the error locator polynomial $\sigma^{(\mu)}(x)$. Depending upon the result of the previous step, we may be required to add a correction term, d_μ. to $\sigma^{(\mu)}(x)$. When we have completed step $t - 1$, $\sigma^{(\mu)}(x)$ is the final error locator polynomial if it has degree less than or equal to t. If the degree is greater than t, then the codeword cannot be corrected (there are more than t errors).

Let us work out an example. Given our sample codeword (0000000001000001100101000100010) we introduce three errors as if it were a corrupt received codeword,
$v(x)$ = 000**1**0000**1**10000011001**0**0000100010. Now we set to work computing syndrome components. Remember that the check polynomials are, with their binary equivalents,

$m_1(x) = m_2(x) = m_4(x) = x^5 + x^2 + 1$ (100101),
$m_3(x) = m_6(x) = x^5 + x^4 + x^3 + x^2 + 1$ (111101),
$m_5(x) = x^5 + x^4 + x^2 + x + 1$ (110111),

so we have three divisions to do to find six syndrome components. These are done by simple binary division, as above, details omitted.

$S_1(x) = v(x) \bmod m_1(x) = x^2$
$S_2(x) = v(x) \bmod m_2(x) = x^2$
$S_3(x) = v(x) \bmod m_3(x) = x^4 + x^3 + x + 1$
$S_4(x) = v(x) \bmod m_4(x) = x^2$
$S_5(x) = v(x) \bmod m_5(x) = x^4 + x$
$S_6(x) = v(x) \bmod m_6(x) = x^4 + x^3 + x + 1$

We find $S_i(\alpha^i)$ by substituting α^i into the equations above, reducing using the table we derived for GF(32) when necessary. Remember that $\alpha^5 = \alpha^2 + 1$.

$S_1(\alpha) = \alpha^2$
$S_2(\alpha^2) = \alpha^4$
$S_3(\alpha^3) = (\alpha^3)^4 + (\alpha^3)^3 + (\alpha^3) + 1 = \alpha^{14}$
$S_4(\alpha^4) = \alpha^8$
$S_5(\alpha^5) = (\alpha^5)^4 + (\alpha^5) = \alpha^{29}$
$S_6(\alpha^6) = (\alpha^6)^4 + (\alpha^6)^3 + (\alpha^6) + 1 = \alpha^{28}$

Using the algorithm, we fill in the table.

μ	$\sigma^{(\mu)}(x)$	d_μ	l_μ	$2\mu - l_\mu$
-½	1	1	0	-1
0	1	$S_1 = \alpha^2$	0	0
1	$\alpha^2 x + 1$	α^{26}	1	1
2	$\alpha^{24}x^2 + \alpha^2 x + 1$	α^{20}	2	2
$t = 3$	$\alpha^{27}x^3 + \alpha^{11}x^2 + \alpha^2 x + 1$	—	—	—

Set $\mu = 0$. We see that $d_\mu \neq 0$, so we choose $\rho = -1/2$, and
$\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x) + d_\mu\, d_\rho^{-1}\, x^{2(\mu-\rho)}\sigma^{(\rho)}(x) = \sigma^{(0)}(x) + d_0\, d_{-1/2}^{-1}\, x^{2(0+1/2)}\sigma^{(-1/2)}(x) =$
$1 + (\alpha^2)\,(1)\,(x)\,(1) = \alpha^2 x + 1$.
Then, $l_{\mu+1} = \deg(\sigma^{(\mu+1)}(x)) = \deg(\alpha^2 x + 1) = 1$.
Finally, $d_{\mu+1} = S_{2\mu+3} + \sigma_1^{(\mu+1)} S_{2\mu+2} + \sigma_2^{(\mu+1)} S_{2\mu+1} + \ldots + \sigma_L^{(\mu+1)} S_{2\mu+3-L} =$
$S_3 + \sigma_1^{(1)} S_2 = (\alpha^{14}) + \alpha^2(\alpha^4) = \alpha^{26}$.

Set $\mu = 1$. We see that $d_\mu \neq 0$, so we choose $\rho = 0$, and
$\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x) + d_\mu\, d_\rho^{-1}\, x^{2(\mu-\rho)}\sigma^{(\rho)}(x) = \sigma^{(1)}(x) + d_1\, d_0^{-1}\, x^{2(1-0)}\sigma^{(0)}(x) =$
$(\alpha^2 x + 1) + (\alpha^{26})\,(\alpha^2)^{-1}\, x^2\,(1) = (\alpha^2 x + 1) + (\alpha^{24})\, x^2 = \alpha^{24}x^2 + \alpha^2 x + 1$.
Then, $l_{\mu+1} = \deg(\sigma^{(\mu+1)}(x)) = \deg(\alpha^{24}x^2 + \alpha^2 x + 1) = 2$.
Finally, $d_{\mu+1} = S_{2\mu+3} + \sigma_1^{(\mu+1)} S_{2\mu+2} + \sigma_2^{(\mu+1)} S_{2\mu+1} + \ldots + \sigma_L^{(\mu+1)} S_{2\mu+3-L} =$
$S_5 + \sigma_1^{(2)} S_4 + \sigma_2^{(2)} S_3 = (\alpha^{29}) + \alpha^2(\alpha^8) + \alpha^{24}(\alpha^{14}) = \alpha^{20}$.

Set $\mu = 2$. We see that $d_\mu \neq 0$, so we choose $\rho = 1$, and
$\sigma^{(\mu+1)}(x) = \sigma^{(\mu)}(x) + d_\mu\, d_\rho^{-1}\, x^{2(\mu-\rho)}\sigma^{(\rho)}(x) = \sigma^{(2)}(x) + d_2\, d_1^{-1}\, x^{2(2-1)}\sigma^{(1)}(x) =$
$(\alpha^{24}x^2 + \alpha^2 x + 1) + (\alpha^{20})\,(\alpha^{26})^{-1}x^2\,(\alpha^2 x + 1) = (\alpha^{24}x^2 + \alpha^2 x + 1) + (\alpha^{25})\,x^2\,(\alpha^2 x + 1) =$
$\alpha^{24}x^2 + \alpha^2 x + 1 + \alpha^{27}x^3 + \alpha^{25}x^2 = \alpha^{27}x^3 + \alpha^{11}x^2 + \alpha^2 x + 1.$

The final error locator polynomial is $\sigma^{(\mu)}(x) = \alpha^{27}x^3 + \alpha^{11}x^2 + \alpha^2 x + 1$.

We next find the roots of $\sigma^{(\mu)}(x)$ in GF(32) by trial and error substitution. (There is a search algorithm due to Chen that is more efficient.) The roots are α^4, α^9, and α^{22}. The bit positions of the error locations correspond to the inverses of these roots, or α^{27}, α^{22}, and α^9, respectively. A polynomial corresponding to the error pattern would then be $e(x) = x^{27} + x^{22} + x^9$. Adding $e(x)$ to the received codeword corrects the errors. Examining the original corrupt codeword we created,

$v(x)$ = 000**1**0000**1**1000001100100**0**000100010
$\oplus$ $e(x)$ = 000**1**0000**1**0000000000001**0**00000000
$c(x)$ = 000**0**0000**0**1000001100101**0**00100010

and it is clear that the calculated error pattern matches the actual error pattern and $c(x)$ matches our original codeword.

If there are no errors, then the syndromes all work out to zero. One to three errors produce the corresponding number of bits in $e(x)$. More than three errors typically results in an error locator polynomial of degree greater than $t = 3$. However, it is again possible that seven bit errors could occur, resulting in a zero syndrome and a false conclusion that the message is correct. That is why most error correction systems take other steps to ensure data integrity, such as using an overall check code on the entire sequence of codewords comprising a message.

In practice, error correction is done in either software or hardware. The Berlekamp algorithm is complex and not too attractive when considered for high-speed communications systems, or operation on power limited microprocessors. One version of the BCH code is used in pagers and another in cell phones, so optimization of the algorithm for the application is important. A cursory scan of the literature shows efforts are being made to discover alternative methods of decoding BCH codes.

Biographical Note on Claude Shannon[15]

Only recently did Claude Shannon pass away, on February 24, 2001 at the age of 84. He earned an M.S. in electrical engineering and a Ph.D. in mathematics from Massachusetts Institute of Technology. “The Mathematical Theory of Communication” was published when he was only 32 years old. He is reported to have been interested in both the practical and theoretical aspects of the problems he tackled, and his theoretical *magnum opus* has had deep practical impact on all our lives through the communications revolution.

Appendix A. GF(2^m) Field Generator Computer Program

```
/* GENFIELD.C

This program generates fields of 2^m elements using polynomial
arithmetic.

Hank Wallace 09-Mar-01
Revision 09-Mar-01

*/

#define src_file 1

#include “process.h”
#include “string.h”
#include “conio.h”
#include “math.h”
#include “ctype.h”
#include “dos.h”
#include “stdio.h”
#include “stdlib.h”

typedef unsigned char   uchar;
typedef signed char     schar;
typedef unsigned int    uint;
typedef unsigned long   ulong;

/* ======================================================= */

char *tobinary(int number, int digits, char *s)
/* This function converts an integer to its binary
representation and places it in in string s. */
{
  int i;

  number<<=16-digits;
  *s=0;
  for (i=0; i<digits; i++)
    {
      if (number & 0x8000)
        strcat(s,”1”);
      else
        strcat(s,”0”);
      number<<=1;
    }
  return(s);
}

/* ======================================================= */

char *topoly(int number, int digits, char *s)
/* This function converts an integer to its polynomial
representation and places it in in string s. */
{
  int i;

  number<<=16-digits;
  *s=0;
  for (i=0; i<digits; i++)
    {
      if (number & 0x8000)
        sprintf(&s[strlen(s)],”a^%d “, digits-i-1);
      number<<=1;
    }
  return(s);
}

/* ======================================================= */
```

```
void main(int argument_count, char *argument[])
{
  int
    i,     // loop index
    order, // order of the generator polynomial
    x,     // the current field element
    n;     // number of elements in the field
  long
    poly,  // polynomial entered by the user
    l;     // scratch
  char
    s[100]; // for string operations

  // look for command line arguments
  if (argument_count != 2)
    {
      printf("GF(2^m) Field Element Generator\n\n");
      printf("Usage: GeneratorPoly(a^n...a^0)\n\n");
      exit(0);
    }

  // read polynomial coefficients
  // polynomial is assumed to have a root alpha not in GF(2)
  poly=atol(argument[1]);

  // determine order of polynomial
  order=31;
  l=poly;
  while ((l & 0x80000000L) == 0)
    {
      order--;
      l<<=1;
    }

  // compute number of elements in the field
  n=1 << order;

  // generate and print the field elements
  printf("Field of %d elements with generator polynomial %s\n",
    n,topoly(poly,order+1,s));

  // print the ever present zero and one elements
  printf("0     %s\n",tobinary(0,order,s));
  printf("1     %s\n",tobinary(1,order,s));

  x=1; // initialize the current field element
  for (i=0; i<n-2; i++)
    {
      x<<=1; // multiply by the root, alpha

      if (x & n) // arithmetic is modulo the polynomial
        {
          // subtract (exclusive OR) the generator polynomial
          x^=poly;
        }
      printf("a^%-2d  %s ",i+1,tobinary(x,order,s));
      printf("%s\n",topoly(x,order,s));
    }
}
```

References for *Error Detection and Correction Using the BCH Code*

[1] Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communication* (Chicago: University of Illinois Press, 1963), p.71.

[2] Shu Lin and Daniel J. Costello, Jr., *Error Control Coding* (Englewood Cliffs, New Jersey: Prentice Hall, 1983), Chapter 1.

[3] Lin and Costello, p. 79.

[4] Golay, Marcel J.E., "Notes on Digital Coding," *Proceedings of the IRE*, Vol. 37, June, 1949, p. 657.

[5] Oliver Pretzel, *Error Correcting Codes and Finite Fields*, (Oxford: Oxford University Press, 1992), p. 77.

[6] Vera Pless, *Introduction to the Theory of Error Correcting Codes*, 2nd ed. (New York: John Wiley & Sons, 1989), p. 23.

[7] Lin and Costello, p. 142.

[8] Ibid.

[9] C. Britton Rorabaugh, *Error Coding Cookbook*, (New York: McGraw-Hill, 1996), p. 36.

[10] Lin and Costello, p. 580.

[11] Pretzel, p.224.

[12] Lin and Costello, p. 151.

[13] Ibid, p. 34.

[14] Ibid, p. 158.

[15] James DeTar, "Mathematician Claude Shannon," *Investor's Business Daily*, March 22, 2001, p. A3.

Author Information

Hank Wallace is President of Atlantic Quality Design, Inc., an electronics R&D firm in Virginia. More publications and information about the company's products and services are available at www.aqdi.com.