

## Chapter 7: Galois-Based Cyclic Codes

### Chapter Contents.

#### 7.1 Overview of Linear Block Codes

The Basics.

Symbols in  $GF(q)$ , parity check symbols, data words  $d$  in  $D^k$ , code words  $c$  in  $C^k$ .

Rank, row space, generator matrix  $G$ , parity check matrix  $H$ , nullspace, dual code

**Fact 0:** If  $G = [P \mid I_k]$ , then  $H = [I_{n-k} \mid -P^T]$ .

The notion of distance between code words: Error Correction

minimum distance  $d$ , corresponding EC capability  $t$ , random errors, distance, weight

**Fact 1:** The minimum distance  $d$  of a code is the minimum weight of all code words.

**Fact 2:** If the minimum distance of a code is  $d$ , then the maximum number  $t$  of symbol errors per code word that the code can correct is given by:  $t = \text{Int}[(d-1)/2]$

**Fact 3:** There is an upper bound on  $d$  or  $t$  which applies to all block codes:  $d \leq n-k+1$  ;

**Definition:** maximum-distance-separable codes, R-S as example.

**Fact 4:** The minimum distance  $d$  of a code is equal to the minimum number of columns of the matrix  $H$  which are linearly dependent.

Encoders and Decoders: The Syndrome

Important Codes, Code History, convolution codes.

#### 7.2 Cyclic Codes

The Cyclic Basis: Definition of a Cyclic Code

The Systematic Basis

Implementation of Encoders and Decoders

Cyclic Redundancy Check (CRC)

#### 7.3 Cyclic vs Systematic Basis of a Cyclic Code: A Rearrangement

**Fact 5:** Given any data word polynomial  $d(x)$ , we can find  $D(x)$  of the systematic code.

**Fact 6:** Given any  $D(x)$  of the systematic code, we can find its data word polynomial  $d(x)$ .

**Fact 7:** The mapping between  $d(x)$  and  $D(x)$  is one-to-one. Thus, one can regard either set of polynomials as a rearrangement of the other set of polynomials.

**Fact 8:** The mapping between  $c(x)$  and  $C(x)$  is one-to-one. Thus, one can regard either set of polynomials as a rearrangement of the other set of polynomials.

**Corollary:** The set of code words of a code in the cyclic basis is a rearrangement of the set of code words of the same code in the systematic basis.

#### 7.4 Why Cyclic Codes are Cyclic

**Definition:** cyclic permutation  $f^{(m)}(x)$ .

**Math Lemma 1:** Claim that:  $f^{(1)}(x) = \text{Rem}[(x f(x))/(x^n - 1)]$ .

**Math Lemma 2:** Claim that:  $f^{(m)}(x) = \text{Rem}[(x^m f(x))/(x^n - 1)]$ .

**Fact 9:** For a cyclic code, all cyclic permutations of any code word are also code words.

(contents continues on next page)

## 7.5 Cyclic Code as an Ideal of the Ring $A_n$

**Fact 10:** A cyclic code with symbols in  $GF(q)$ , and which is generated by some  $g(x)$  which divides  $x^n - 1$ , forms an ideal  $(g(x))$  of the ring  $A_n = \text{Polys}[x, GF(q)] / (x^n - 1)$ . This ideal consists of the set of  $q^k$  code words generated by multiplying data polynomials  $d(x)$  by  $g(x)$ ,

$$\{c(x)\} = \{d(x)\} \bullet \{g(x)\}$$

$$\text{Standard Array} = A_n / (g(x))$$

## 7.6 The Connection between Cyclic Codes and Galois Fields

**Fact 11:** The parity check matrix  $H$  for a Galois cyclic code has a certain form.

## 7.7 The BCH Codes

The BCH Bound

BCH Bound Corollary

## 7.8 The Narrow-Sense BCH Codes

**Example:**  $GF(32 = 2^5)$

The  $H$  matrix for narrow-sense BCH codes.

Comments

## 7.9 The Hamming Codes

Hamming codes for  $p=2$

The  $H$  matrix for the Hamming Codes.

Why are Hamming codes with  $p > 2$  unable to correct single symbol errors?

Modified Hamming Codes

## 7.10 The Reed-Solomon Codes

Comparison to a Digital Filter

## Chapter 7: Galois-Based Cyclic Codes

There are many good texts on the theory of error-correcting codes, see References. It has been our experience that most of the difficulty encountered in reading these texts is not so much with the coding theory itself, but with the underlying theory of Galois Fields. Since we have just finished a long exposition of Galois Field theory, and before we continue on to certain specific applications, it seems appropriate to at least make the connection to coding theory.

### 7.1 Overview of Linear Block Codes

#### The basics.

**Definition:** For our purposes, a *symbol*, or *code symbol*, is an element of some finite field  $GF(q)$ . For normal binary codes, each code symbol lies in the field  $GF(2)$  and is therefore called a *bit*. We shall try to keep the discussion as general as possible by using the term *symbol* in place of *bit*.

The basic idea of a *block code* is to take *data words* of  $k$  symbols each, add  $n-k$  *parity check symbols* which are dependent on (computed from) the  $k$  data symbols, and end up with *code words* of  $n$  symbols each. Thus, data is encoded in a finite *blocks* consisting of  $n$  symbols. Such a code is denoted as  $(n,k)$ .

A subset of block codes are the *linear* block codes, where the code words are generated by the application of a linear operator onto the data words. Since we are limiting our interest to situations where  $k$  and  $n$  are finite numbers, such a linear operator is simply a *matrix*. If we think of data words and code words as row vectors (as opposed to column vectors), then a linear block code is defined by a matrix  $G$  such that

$$c = d \bullet G$$

The data word  $d$  is a  $k$ -component row vector, the code word  $c$  is an  $n$ -component row vector, and therefore  $G$  is a matrix with  $n$  columns and  $k$  rows.  $G$ , known as the *generator matrix* of the code, must be a matrix whose *rank* is  $k$ , which means it has  $k$  linearly independent rows (or columns). These  $k$  linearly independent rows span a vector space of dimension  $k$  known as the *row space* of  $G$ . By applying the matrix  $G$  to unit-vector data words, such as  $d = (1,0,0\dots)$ , one sees that the  $k$  rows of  $G$  are in fact some of the code words of the code. These particular code words form a  $k$ -dimensional basis for the whole code -- any code word can be written as a linear combination of the  $k$  code words which are the  $k$  rows of  $G$ . In other words,

$$\text{linear block code } (n,k) = \text{the } k\text{-dimensional row space of } G.$$

As a matter of notation, we have chosen the convention of representing the vectors  $c$  and  $d$  as row-vectors. Coding theory texts often do this as a space-saving measure, since one wastes less paper writing a row vector than a column vector. One can of course also think in terms of column vectors  $d^T$  and  $c^T$ . The superscript  $T$  means transpose, which when applied to an arbitrary  $n \times m$  matrix means that you interchange the rows and the columns. That is,  $(M^T)_{ij} = M_{ji}$ . Thus, we can reexpress our code as

$$c^T = G^T \bullet d^T$$

In general, as noted above, the components of all vectors  $d$  and  $c$ , and the components of the matrix  $G$  are *symbols*, that is, they are elements of some field  $GF(q)$ . Thus, whenever one has a  $+$  or  $\bullet$  operation, this operation is that defined by  $GF(q)$ . Notice that there is always an implied  $+$  operation when we multiply matrices.

Since  $GF(q)$  has  $q$  elements, we realize that there are  $q^k$  possible data words in an  $(n,k)$  code -- since all  $k$ -tuples are "legal" data words. Let us call this  $k$ -dimensional data space by the name  $D^k$ .

Similarly, the set of all possible  $n$ -tuples forms a space of  $q^n$  elements which we shall call  $V^n$ . However, not all  $n$ -tuples in this space are code words. The code space, which is the space in which the code words (= code vectors) live, and which is the rowspace of  $G$ , is a *subspace* of  $V^n$  of dimension  $k$ , which we will call  $C^k$ .

So we see that all data vectors in our data space  $D^k$  get mapped by  $G$  into the code space  $C^k$ , which is a subspace of  $V^n$ . Within  $V^n$  there exists another useful subspace besides  $C^k$ . It is the space formed by the set of all  $n$ -tuples in  $V^n$  which are *orthogonal* to the code vectors of  $C^k$ . In other words, an  $n$ -tuple vector  $h$  is in this new space as long as,

$$c \bullet h^T = 0$$

Here we have written  $h^T$  to get a column vector, which we can dot with the row-vector  $c$  to get 0. Note, by the way, that this 0 is a symbol, an element of  $GF(q)$ .

It is not hard to show that the set of such orthogonal vectors  $h$  does in fact form a vector space, and that the dimension of this vector space is  $n-k$ . Within this space, we can therefore find  $(n-k)$   $n$ -tuples  $h$  which satisfy the above equation for all code vectors  $c$ . We can then consider these  $h$ 's to be the *rows* of a matrix  $H$  which has  $n-k$  rows and  $n$  columns. Then we can write:

$$c \bullet H^T = 0 \quad \text{or alternatively, applied to column vectors } c^T, \quad H \bullet c^T = 0$$

This matrix  $H$  has rank  $n-k$ , which means its  $n-k$  rows form a vector space of dimension  $n-k$  within  $V^n$ . This is just the space we are talking about, the space within  $V^n$  all of whose elements are orthogonal to the code vectors of  $C^k$ . This space has a traditional name in matrix mathematics. It is called the *nullspace* of  $C^k$  and we can denote it as  $N^k$ . It has dimension  $n-k$ , not  $k$ .

In coding theory, the matrix  $H$  is called the *parity check matrix* of the  $(n,k)$  code. You can "check" that a vector  $c$  really is a code word by applying  $H$  as shown above to see if the result is 0. This relates to the topic of syndromes, which we defer till later.

Since the rows of  $G$  are a set of code vectors, it trivially follows from the above that:

$$G \bullet H^T = 0 \quad \text{or} \quad H \bullet G^T = 0$$

Since matrix  $H$  has rank  $n-k$ , it can *itself* be considered as the generator matrix of a new code having data words containing  $n-k$  symbols. That is, we can define a code:

$$c' = d' \bullet H$$

where  $d'$  is a  $(n-k)$ -tuple and  $c'$  is an  $n$ -tuple. This code is  $(n, n-k)$  and is known as the *dual code* to  $(n, k)$ . All code vectors  $c$  of the code  $(n, k)$  are orthogonal to all code vectors  $c'$  of the dual code  $(n, n-k)$ . Here is a little proof:

$$c' \bullet c^T = (d' \bullet H) \bullet (d \bullet G)^T = (d' \bullet H) \bullet G^T \bullet d^T = d' \bullet (H \bullet G^T) \bullet d^T = d' \bullet (0) \bullet d^T = 0$$

The normal manner of expressing the  $G$  matrix is

$$G = [P \mid I_k] .$$

Recall that  $G$  has  $n$  columns and  $k$  rows. The first  $n-k$  columns contain some submatrix  $P$ , and the last  $k$  columns contain a  $k \times k$  identity matrix. When this  $G$  is applied to a data vector  $d$ , as in  $c = d \bullet G$ , the first  $n-k$  components of  $c$  are the parity check symbols obtained by multiplying the vector  $d$  by the first  $n-k$  columns of  $G$ , where the  $P$  matrix data is located. The last  $k$  components of  $c$  are exactly the components of  $d$ . This form of matrix  $G$  is called the *systematic basis*, and we shall refer to it again in reference to cyclic codes.

One might wonder if, given the above matrix  $G$ , one can construct  $H$ . The answer is yes.

**Fact 0:** If  $G = [P \mid I_k]$ , then  $H = [I_{n-k} \mid -P^T]$ .

**Proof:** Notice that  $H$  has all the right dimensions:  $n$  columns and  $n-k$  rows. To prove the above, we need to show that  $G \bullet H^T = H \bullet G^T = 0$ . In other words, we need to show that

$$\sum_{j=0}^{n-1} G_{ij} H_{sj} = 0$$

Remember that the first index is rows, the second is columns. The proof is easy once we have a good way to codify the matrices. Here is a method that seems to work. Let  $x = n-k$  as a shorthand, and use the Figures shown in Appendix 7.1. Then:

$$G_{ij} = P_{ij} \theta(j < x) + \delta_{i, j-x} \theta(j \geq x)$$

$$H_{sj} = \delta_{sj} \theta(j < x) - P_{j-x, s} \theta(j \geq x)$$

The function  $\theta(\text{boolean})$  is 1 for  $\text{boolean} = \text{true}$  and 0 for  $\text{boolean} = \text{false}$ . This is just a trick to restrict the range of  $j$  summation as needed. If you now multiply these things together and sum over  $j$ , there are four terms. Two vanish at once due to the boolean  $\theta$ 's. For example,  $\theta(j \geq x) \theta(j < x) = 0$ . You end up with the other two terms:

$$P_{is} \theta(s < x) - P_{is} \theta(i \geq 0)$$

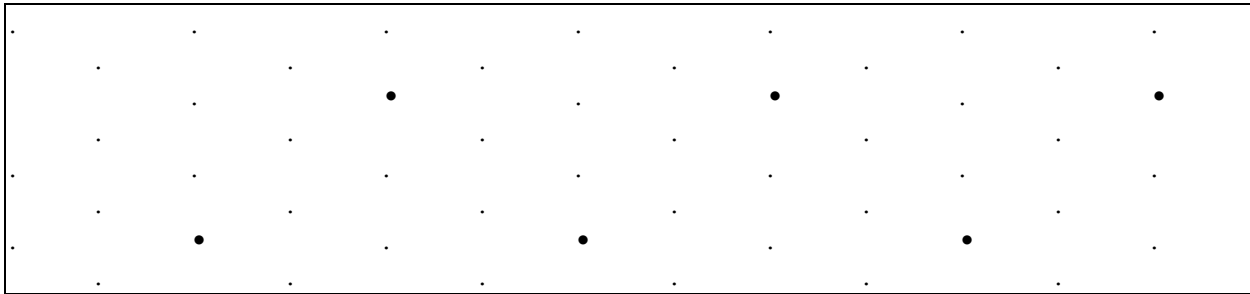
But  $\theta(i \geq 0) = 1$ , since we never deal with negative indices. Similarly, as shown in the Appendix 7.1 figure,  $P$  is only defined for its column index  $s < x$ . It can be regarded as zero beyond this range, so we can set  $\theta(s < x) = 1$  without changing anything. The result is then

$$P_{is} - P_{is} = 0$$

Thus, we have shown that the  $G = [P \mid I_k]$ ,  $H = [I_{n-k} \mid -P^T]$  results in  $G \bullet H^T = H \bullet G^T = 0$ .

### **The notion of distance between code words: Error Correction**

We have noted that the code space  $C^k$  is a subspace of  $V^n$ . There are many  $n$ -tuples in  $V^n$  which are not code words. Here is a suggestive picture, where the large dots are code words in  $C^k$ , and the little dots are other vectors in  $V^n$ :



Suppose an *error* occurs during the transmission of a code word (•), and a code word is converted into an unused vector (·). It seems reasonable to assume that such an erroneous code word is a mutation of the "nearest" good code word (•). By "error" we mean that one or more symbols of the code word got hammered during transmission. If the number of such symbol alterations in a code word is  $t$ , we say that there was a  $t$ -symbol random error in that code word.

As the picture above suggests, if the bad code word (·) is an immediate neighbor of a good word (•), there seems little doubt as to how to do the correction. You correct to that immediate neighbor.

In the above picture, the *distance* between code words is 3. That is, 3 symbols must be changed to get from one (•) to another (•). Thus, if we get some bad code word (·), it is more likely that it represents a 1-symbol error than a 2-symbol error, so the best thing to do is correct it to the nearest neighboring (•).

In general, a code word (•) has some "private region" surrounding it, and all bad code words (·) in this region most likely are mutations of the (•). The larger  $n$  is relative to  $k$ , and the better "quality" the code has, the larger is this private region surrounding each code word, so more erroneous symbols can be corrected.

However, not all code words necessarily have the same size private region. The *smallest* distance between *any* two code words of a code is called  $d$ .

$$d = \text{smallest distance between any two code words of a code}$$

Suppose you wish to be able to "correct" all random errors of  $t$  or fewer symbols.

$$t = \text{number of random errors (in symbols per codeword) that a code can "correct"}$$

We use the term *random error* to indicate errors that occur independently on each symbol of a code word during transmission. Another class of errors are *burst errors*, where there is a coherent destruction of many symbols by a single event. Burst errors are important in coding theory, but we shall have nothing to say about them other than that they exist, and one can calculate their probability and analyze the quality of a code in terms of how well it can correct burst errors.

If  $d \geq 2t+1$ , then one can make a clear presumption that a non-codeword vector that results from some random error is closer to one codeword than to any other, and it would best be corrected to this code word. For example, if there were a 2-symbol error, and if the distance between code words were at least

5 symbols (meaning 5 symbols must be changed to convert one codeword into another), then you would have a choice of making a 2-symbol correction to get to codeword A, or a 3-symbol correction to get to codeword B. Probability favors codeword A. Thus, "error correction" is really a probabilistic process, not a foolproof one. Moreover, it is possible that an error ( a 5-symbol error in this example) could convert one code word into another. This would be an undetectable error.

If the probability of a random error in a symbol of a code word is very small, like  $10^{-6}$ , then it is very reasonable to assume that the a (t+1)-symbol random error is less likely than a t-symbol error:

$$\text{Probability of (t+1)-symbol error} = (10^{-6})^{(t+1)}$$

$$\text{Probability of (t)-symbol error} = (10^{-6})^{(t)}$$

Thus, in this case, the t+1 symbol error is a million times less likely than a t symbol error.

**Definition:** The *distance* between any two code words of a code is the number of non-zero symbols in the *difference* between these two code words.

**Definition:** The *weight* of a code word is the number of non-zero symbols in the code word. [ This is the Hamming weight.]

**Fact 1:** The minimum distance d of a code is the minimum weight of all code words.

**Proof:** This *distance* between any two code words is the *weight* of the different code word. As you consider all possible distances, you encounter all possible weights. QED.

**Fact 2:** If the minimum distance of a code is d, then the maximum number t of symbol errors per code word that the code can correct is given by:

$$t = \text{Int}[(d-1)/2] \quad (\text{Int} = \text{Integer Part})$$

**Proof by example:** If  $d = 5$ , then you have  $(\bullet \cdots \bullet \cdots \bullet)$ , and any bad code word can be unambiguously mapped to a good code word if the error was less than  $t=2$  symbols. If there were a  $t \geq 3$  symbol error, your correction would fix it as if it were  $t \leq 2$  symbol error, and the result would be wrong. If  $d=6$ , then you have  $(\bullet \cdots \bullet \cdots \bullet)$ , and you can still only unambiguously correct  $t \leq 2$  bit errors. An error of 3 symbols which puts you at a middle (·) cannot be corrected without a 50% chance ( in our schematic model) of doing the wrong thing.

**Corollary:** The relation between t and d is  $d = 2t+1$  if d is odd, and  $d=2t+2$  if d is even.

**Fact 3:** There is an upper bound on d or t which applies to all block codes. It is this:

$$d \leq n-k+1 \quad \text{or} \quad t \leq \text{Int}[(n-k)/2]$$

**Proof:** Consider the data unit vector  $d = (10\dots)$ . The weight of the corresponding code vector is at most  $1 + (n-k)$ , because at most you might have all n-k parity check symbols non-zero. The minimum weight over all code vectors must certainly be  $\leq$  the maximum weight of this one code vector. Thus, we arrive

that the fact that the minimum weight  $\leq n-k+1$ . According to Fact 1, we set this equal to the code distance  $d$ :

$$d \leq (n-k+1) \quad t \leq \text{Int}[(n-k)/2]$$

The rightmost inequality follows from Fact 2 above.

Obviously, the closer you can come to this bound, the happier you are, since you can correct more errors  $t$  for a given number  $n-k$  of parity check symbols. Amazingly, certain high quality codes hit this bound exactly.

**Definition:** A code which satisfies the above bound as an equality is called *maximum-distance-separable*. The Reed-Solomon codes are an example.

**Fact 4:** The minimum distance  $d$  of a code is equal to the minimum number of columns of the matrix  $H$  which are linearly dependent. This should not be confused with the rank of  $H$ , which is  $n-k$ , and which is the maximum number of columns which are linearly independent.

**Proof:** Suppose there are  $n$  columns of  $H$  which are linearly dependent. Linear dependence of  $n$  columns means that there is a linear combination of these columns which adds up to 0. According to our known fact that  $H \cdot c^T = 0$ , we can interpret the coefficients in any such linear combination as the coefficients of some code word  $c$ . Thus, we have a code word of weight  $n$  for every set of  $n$  dependent columns of  $H$ . According to Fact 1,  $d$  is the minimum weight of all code words. If there were a set of  $n$  dependent columns with  $n < d$ , then we would have some code word with weight  $= n < d$ , which is a contradiction. Thus,  $n \geq d$ . So  $d$  is then the minimum number of columns that are linearly dependent.

**Corollary:** For a given  $H$  of rank  $n-k$ ,  $d$  can be as small as 1. This occurs if some column is a multiple of another column, something that can easily happen when symbols are in some field  $GF(q)$ . On the other hand, from the definition of "rank", we know that *all* sets of  $n-k+1$  columns are linearly dependent, so we cannot have  $d > n-k+1$ . Thus, we have a second proof of the bound mentioned above.

### Encoders and Decoders: The Syndrome

An "encoder" is a piece of hardware or software which in effect applies the matrix  $G$  and generates  $n$ -symbol code words from  $k$ -symbol data words. It does this by adding the  $n-k$  parity check symbols.

A "decoder" is a piece of hardware or software which receives the code words, and tries to determine if there have been errors, and may try to correct some errors.

When a bad code word is received, ie, some  $c' \neq c$  because an error occurred in transmission, the so-called *syndrome*  $s$  is the result of the application of  $H^T$  to  $c'$ . Recall from the above discussion that  $c \cdot H^T = 0$  for good code words. Thus, the syndrome is defined as

$$s = c' \cdot H^T$$

One can express the bad code word as a good code word plus an error code word, so  $c' = c + e$ . Then

$$\mathbf{s} = \mathbf{c}' \cdot \mathbf{H}^T = (\mathbf{c} + \mathbf{e}) \cdot \mathbf{H}^T = \mathbf{c} \cdot \mathbf{H}^T$$

Since  $\mathbf{H}^T$  has  $n-k$  columns, the syndrome vector  $\mathbf{s}$  has  $n-k$  components. A non-vanishing syndrome vector *detects* an error. Good code words always have zero syndrome. An error correcting decoder attempts to look up the most likely  $\mathbf{e}$  for a given  $\mathbf{s}$ , and then corrects the error by doing  $\mathbf{c}' = \mathbf{c} - \mathbf{e}$ .

### Important Codes, Code History, and other kinds of codes.

Linear block codes are characterized by their values of  $n$  and  $k$ , by their  $d$  (which is really their ability to correct  $t$ -symbol random errors), and by the complexity of the required encoders and decoders. Another factor is ability to correct burst errors. All these things are measures of the *performance* of a code. Some of the names of famous block codes are Hamming, Golay(1949), Hadamard, Reed-Muller (1954), Fire(1959), BCH(1960), and Reed-Solomon(1960). There are many more.

The study of error correcting codes is a relatively recent development, on the time scale of the supporting math. Although Galois died in 1832, Shannon's famous paper was published in 1948. The Hamming codes were discovered in 1950. Fire codes came in 1959, especially formulated to correct long burst errors. In 1960 came the large class of BCH codes, which we will discuss below, that could correct multiple random symbol errors. The Reed-Solomon codes appeared in 1960. These are BCH codes applied to code symbols in  $\text{GF}(2^m)$  and are thus finding much current use. They have good burst correcting capabilities, and are used, for example, in most modern digital audio equipment.

*Convolution codes* are in a completely different class from block codes, and were developed in the 1960's. You produce  $n$  code symbols based on  $k$  data symbols, but also based on some number  $m$  of *previous* data symbols, so there is no clearly defined "block" that is the basic encoded unit. These systems use encoders which look somewhat like polynomial multipliers in that they store some number of previous data symbols in flipflops and have lots of feed-forward adders. There is a definite state-machine flavor here, due to the memory of previous symbols. Convolution codes are much more complex than linear block codes, and topological methods, such as tree diagrams, are used to understand them. In contrast, the block codes and especially the cyclic block codes have a strong "algebraic" flavor. For example, they might involve the algebra of Galois field elements.

According to Rhee, the 1970's were spent mostly finding codes of longer length and better performance, while the 1980's concentrated on practical applications.

## 7.2 Cyclic Codes

Cyclic codes are a subset of the linear block codes. The name arises from the fact that, in a cyclic code, any cyclic permutation of a code word is also a code word. This fact will be proven below.

### The Cyclic Basis: Definition of a Cyclic Code

A cyclic code is constructed (and therefore defined) by the sequence of instructions given below. All polynomials referred to have coefficients and  $x$  in some field  $GF(q) = GF(p^m)$ .

(a) Select a integer value for  $n$ , the desired length of the code words.

(b) Find a pair of polynomials  $g(x)$  and  $h(x)$  such that

$$g(x)h(x) = x^n - 1.$$

The degree of  $h(x)$  is  $k$ , the degree of  $g(x)$  is  $n-k$ . The polynomial  $g(x)$  is the *generator* of the  $(n,k)$  code, and  $h(x)$  is the generator of the dual code  $(n,n-k)$ .

(c) Let a set of  $k$  data symbols (in  $GF(q)$ ) be the coefficients of a polynomial  $d(x)$  of degree  $k-1$ .

(d) The code words are the coefficients of polynomials  $c(x)$  where

$$c(x) = d(x) g(x).$$

Since  $d(x)$  is of degree  $k-1$ , and  $g(x)$  is of degree  $n-k$ ,  $c(x)$  is of degree  $n-1$ , which means it has  $n$  coefficients or symbols in  $GF(q)$ .

Now it happens that when you compute  $c(x) = d(x)g(x)$ , the  $n$  coefficients of  $c(x)$  are a convolution of the  $k$  data coefficients of  $d(x)$  with the  $n-k$  coefficients of  $g(x)$ . In the second section of Chapter 4 we had just this situation and we are able to conclude here that

$$c_i = \sum_{j=0}^i d_{i-j} g_j$$

It is possible to untangle this convolution and write the code generating matrix  $G$  in a so-called *systematic* form. In this form, or basis, the  $n$ -symbol code words have their first  $n-k$  symbols being parity check symbols, and their last  $k$  symbols being the data symbols. In the cyclic form, as was just noted, these symbols are all convoluted together in the  $n$ -bit code words. More on this below.

For the moment, imagine working in the cyclic basis. It is not hard to build a piece of hardware that computes  $c(x) = d(x)g(x)$ , and we shall have much to say about this in a later chapter. The circuit is a *polynomial multiplier*.

The code word  $c(x)$  is then transmitted over a channel. The receiver, to nobody's great surprise, tries to recover  $d(x)$  by *dividing*  $c(x)$  by  $g(x)$ :  $d(x) = c(x)/g(x)$ . The hardware for doing this is called a *polynomial divider*.

**Example:** In a Reed-Solomon code, the code symbols are parallel groups of bits, so the R-S polynomial multipliers and dividers have data paths that are multiple bits wide. They look very much like digital filters, with one significant difference which we will describe below.

## The Systematic Basis

In practice, one tends to use hardware which works in the systematic basis rather than the cyclic basis, but the principle is as described above. The reason is that error detection and correction is much easier in the systematic basis, since you know where the data and parity check bits are in your code words.

According to our polynomial Division Algorithm of Chapter 3, we can write the following expansion:

$$x^{n-k} d(x) = D(x) g(x) - \gamma(x)$$

In other words, we multiply  $d(x)$  [ which carries our  $k$  data symbols as coefficients ] by a power, and then divide this product by the generator polynomial  $g(x)$  which, recall, has degree  $n-k$ . Then  $D(x)$  is the quotient polynomial, and  $-\gamma(x)$  is the remainder. Whatever  $\gamma(x)$  is, it is of degree less than  $n-k$  and thus has  $n-k$  coefficients. Since the LHS has degree  $(n-k) + (k-1) = (n-1)$ , and since  $g(x)$  has degree  $(n-k)$ ,  $D(x)$  must have degree  $(k-1)$ , the same as  $d(x)$ .

Suppose we now define our code word to be

$$\begin{aligned} C(x) &= D(x) g(x) \\ &= \gamma(x) + x^{n-k} d(x) \end{aligned}$$

If you write out the coefficients of  $C(x)$ , you find from the second line above that the first  $n-k$  of them are the coefficients of  $\gamma(x)$ , and the last  $k$  of them are the data bits of  $d(x)$ . Therefore, the coefficients of  $\gamma(x)$  are precisely the  $n-k$  parity check symbols which you combine with the  $k$  data symbols to get an  $n$ -symbol code word. Thus, we are now in the systematic basis. In the cyclic basis, we had code words  $c(x) = d(x)g(x)$ , whereas in the systematic basis, we have  $C(x) = D(x)g(x)$ .

We should note that the coefficients of a polynomial are usually processed *most significant symbol first*. We certainly know from our pre-calculator experience with long division that we start with the most significant digit of the dividend. A hardware polynomial divider is no different. Thus, in terms of time ordering, when we think of  $C(x)$  as described above, the most significant symbols are the data symbols and come first, then the least significant symbols are the parity check symbols, and they come last in a data stream of coefficients which represents  $C(x)$ .

In either cyclic or systematic basis, the code word is the product of some sort of data polynomial of degree  $k-1$  with the generator polynomial of degree  $n-k$  to give a code word polynomial of degree  $n-1$ . The construction which led to  $D(x)$  is what one needs to do to untangle the convolution contained in our earlier formulation  $c(x) = d(x)g(x)$ .

## Implementation of Encoders and Decoders

If  $C(x)$  is sent through a channel, how do we recover  $d(x)$  at the other end? We just grab the first  $k$  symbols of  $C(x)$  and these make up  $d(x)$ .

How do we know if there was an error? We set up a polynomial divider to compute  $C(x)/g(x)$ . The quotient is  $D(x)$ , and the remainder is supposed to be zero! That is, true code words  $C(x)$  are multiples of  $g(x)$ , so there should be no remainder. If there is a non-zero remainder, then there has been an error. This remainder is a form of the *syndrome* referred to above. Thus, at the receiver we have:

$$C'(x) = D(x) g(x) + s(x)$$

Here we indicate by  $C'(x)$  a received word that has an error. If there is no error so  $C' = C$ , a code word, then  $s(x) = 0$ . If  $s(x) \neq 0$ , then we have *detected* an error.

If  $s(x) \neq 0$ , then the coefficients of  $s(x)$  can be used to *correct* the error, provided that the size of the error was less than  $t$  symbols, and the selected code can correct  $t$ -symbol errors. A large fraction of any book on error-correcting codes deals with the details of how this is done for various codes, and this is where implementations can become quite complex.

Obviously, it is the fact that there are extra parity check symbols in the code which allows an error to be detected and possibly corrected.

Let us take one more look at the encoding and decoding equations for a cyclic code in the systematic basis:

$$\text{encode} \quad C(x) = D(x) g(x) = \gamma(x) + x^{n-k} d(x)$$

$$\text{decode and detect error } C'(x) = D(x) g(x) + s(x)$$

We wish to stress the extreme simplicity of implementing these two functions in hardware, apart from the error correction.

The encoder is a two-stage process. In the first stage, we just bypass through the  $k$  data symbols into the channel, since these are the higher coefficients of  $C(x)$ . At the same time, we run these same symbols into a polynomial divider set up to compute  $[x^{n-k} d(x)] / g(x)$ . As we shall later see, the factor  $x^{n-k}$  is a trivial complication in the implementation of such a divider. We then throw out the quotient  $D(x)$ ; it goes into a bit bucket. When the division is done, the flipflops of the divider contain the remainder  $\gamma(x)$ . These coefficients are then shifted out into the channel and we are done.  $C(x)$  is built and sent.

A non-correcting decoder is also a two-stage process. The first  $k$  symbols of  $C'(k)$  are the data symbols we want. They are parked somewhere. At the same time, all  $n$  symbols of  $C'(x)$  are run through a polynomial divider which computes  $C'(x)/g(x)$ . The quotient is again thrown out, and the remainder is the syndrome  $s(x)$ . If  $s(x) \neq 0$ , an error has been detected. Error correction logic can then attempt to correct the code word that has been temporarily parked, before it is sent out of the decoder.

### Cyclic Redundancy Check (CRC)

If we ignore error correction, the circuit described above is exactly how a CRC system works. There is some generator polynomial  $g(x)$ . The parity check symbols are computed as shown above and are tacked onto the end of the data stream to form  $C(x)$ . At the receiver, the data symbols are siphoned off as

needed. The entire incoming stream  $C'(x)$  is run through a divider and the syndrome remainder is computed. If it is not zero, there was a "CRC error".

Of course the number of data symbols  $k$  can be somewhat larger than one usually thinks of in a "code". If the generator polynomial  $g(x)$  has degree  $m = n - k$ , then the output stream has  $n = k + m$  symbols. Since we do not care if the code words are cyclic under cyclic permutations, we do not require that  $g(x)$  divide  $x^n - 1$ . Thus, one can do CRC with an arbitrary  $k$ . The probability of detecting an error increases with the degree of  $g(x)$  which is the number of check symbols  $n - k$ . A standard number often used is  $n - k = 16$ . Normally, "symbols" means "bits, and  $GF(q) = GF(2)$  for a CRC system.

It is interesting that neither Rhee nor Peterson and Weldon mention CRC in their respective 500-page books. The presumed reason is that CRC does not relate to "error correction".

### 7.3 Cyclic vs Systematic Basis of a Cyclic Code: A Rearrangement

In the cyclic basis, the code words of a cyclic code are formed as  $c(x) = d(x)g(x)$ , whereas in the systematic basis, the code words are  $C(x) = D(x)g(x)$ .

**Fact 5:** Given any data word polynomial  $d(x)$ , we can find  $D(x)$  of the systematic code.

**Proof:** The connection between data polynomial  $d(x)$  and  $D(x)$  is given by a Division Algorithm expansion over  $g(x)$  [which has degree  $n - k$ ]:

$$x^{n-k} d(x) = D(x)g(x) - \gamma(x).$$

For any  $d(x)$  of degree  $k - 1$ , there exists a unique  $D(x)$  of degree  $k - 1$  which is the quotient of the above Division Algorithm expansion over  $g(x)$ . The remainder  $-\gamma(x)$  is also unique, and of degree  $< n - k$ .

**Fact 6:** Given any  $D(x)$  of the systematic code, we can find its data word polynomial  $d(x)$ .

**Proof:** Form the product  $D(x)g(x)$ . This product is of degree  $(k - 1) + (n - k) = (n - 1)$ . Expand it using the Division Algorithm over the function  $x^{n-k}$ ; this yields unique  $q(x)$  and  $r(x)$ :

$$D(x)g(x) = q(x)x^{n-k} + r(x) \quad \begin{array}{l} q(x) \text{ has degree } k-1 \\ r(x) \text{ has degree less than } n-k \end{array}$$

Rearrange as

$$x^{n-k} q(x) = D(x)g(x) - r(x)$$

Compare this to

$$x^{n-k} d(x) = D(x)g(x) - \gamma(x).$$

Thus, we have found candidate polynomials for  $d(x)$  and  $\gamma(x)$ :

$$d(x) = q(x) \quad \gamma(x) = r(x)$$

**Fact 7:** The mapping between  $d(x)$  and  $D(x)$  is one-to-one. Thus, one can regard either set of polynomials as a rearrangement of the other set of polynomials.

**Proof:** Suppose different  $d_1(x)$  and  $d_2(x)$  both map into the same  $D(x)$ . Then

$$\begin{aligned} x^{n-k} d_1(x) &= D(x) g(x) + \gamma_1(x) \\ x^{n-k} d_2(x) &= D(x) g(x) + \gamma_2(x) \end{aligned}$$

Subtract to get:

$$x^{n-k} [d_1(x) - d_2(x)] = \gamma_1(x) - \gamma_2(x)$$

The LHS is a polynomial of degree  $\geq n-k$ . The RHS is a polynomial of degree  $< n-k$ . This is a contradiction, so you cannot have two different  $d$ 's which map into the same  $D$ .

Similarly, suppose different  $D_1(x)$  and  $D_2(x)$  map into the same  $d(x)$ . Then write

$$\begin{aligned} x^{n-k} d(x) &= D_1(x) g(x) + \gamma_1(x) \\ x^{n-k} d(x) &= D_2(x) g(x) + \gamma_2(x) \end{aligned}$$

Subtract to get:

$$[D_2(x) - D_1(x)] g(x) = \gamma_1(x) - \gamma_2(x)$$

We make the same argument as above: The LHS is a polynomial of degree  $\geq n-k$ . The RHS is a polynomial of degree  $< n-k$ . This is a contradiction, so you cannot have two different  $D$ 's which map into the same  $d$ .

Thus, the mapping between the  $d(x)$  and  $D(x)$  is a 1-to-1 mapping.

**Fact 8:** The mapping between  $c(x)$  and  $C(x)$  is one-to-one. Thus, one can regard either set of polynomials as a rearrangement of the other set of polynomials.

**Corollary:** The set of code words of a code in the cyclic basis is a rearrangement of the set of code words of the same code in the systematic basis.

**Proof:** According to Fact 7, the mapping between the  $d(x)$  and the  $D(x)$  is one-to-one. Then, since

$$c(x) = g(x)d(x) \quad C(x) = g(x)D(x)$$

we can make a corresponding one-to-one mapping between the  $c(x)$  and the  $C(x)$ .

**Comment:** The Corollary is true whether or not the generator  $g(x)$  divides  $x^n - 1$ . Nowhere in this section have we made use of this fact. However, this fact is the basis of the next section.

## 7.4 Why Cyclic Codes are Cyclic

In the above discussion of cyclic codes, it was noted that the name arises because all cyclic permutations of code words are also code words. Here we will prove this is so.

**Definition:** If we take the coefficients of a polynomial  $f(x)$  and "rotate them  $m$  places to the right" to form a new polynomial, this new polynomial is called a *cyclic permutation* of the original polynomial by  $m$  places, and is denoted by  $f^{(m)}(x)$ .

**Example:**  $f(x) = f_0 + f_1 x + f_2 x^2 + \dots + f_{n-1} x^{n-1}$

$$f^{(1)}(x) = f_{n-1} + f_0 x + f_1 x^2 + \dots + f_{n-2} x^{n-1}$$

**Math Lemma 1:** Claim that:  $f^{(1)}(x) = \text{Rem}[(x f(x)) / (x^n - 1)]$ .

**Proof:** Let  $f(x) = f_0 + f_1 x + f_2 x^2 + \dots + f_{n-1} x^{n-1}$ . Then

$$\begin{aligned} x f(x) &= f_0 x + f_1 x^2 + f_2 x^3 + \dots + f_{n-1} x^n \\ &= [f_{n-1} + f_0 x + f_1 x^2 + \dots + f_{n-2} x^{n-1}] + f_{n-1} (x^n - 1) = [f^{(1)}(x)] + f_{n-1} (x^n - 1). \end{aligned}$$

If we were to expand  $x f(x)$  over  $(x^n - 1)$  using the Division Algorithm, we would recognize the constant  $f_{n-1}$  as the quotient polynomial, and the thing in  $[ ]$  as the remainder polynomial. Q.E.D.

**Math Lemma 2:** Claim that:  $f^{(m)}(x) = \text{Rem}[(x^m f(x)) / (x^n - 1)]$ . Equivalently,

$$x^m f(x) = q(x) (x^n - 1) + f^{(m)}(x) \quad \text{for some quotient polynomial } q(x)$$

**Proof:** Math Lemma 1 shows this is true for  $m=1$ . Then do an induction proof. Assume true for  $m=k$ . Then true for  $m=k+1$  by applying Math Lemma 1. To get from  $m=k$  to  $m=k+1$  you apply a single power of  $x$ , and this is what Math Lemma 1 is all about. Thus, Math Lemma 2 is true for all integer  $m$ .

**Fact 9:** For a cyclic code, all cyclic permutations of any code word are also code words. Thus, the  $k$ -dimensional vector space  $C^k$  spanned by the code words is a *cyclic* subspace of  $V^n$ .

**Proof:** We do this proof in the cyclic basis. It then also applies to the systematic basis, since we have just shown above (Corollary of Fact 8) that the set of code words is the same in either basis.

Apply Math Lemma 2 to  $f(x) = c(x)$ , a polynomial of degree  $n-1$  corresponding to a legal codeword,

$$x^m c(x) = q(x) (x^n - 1) + c^{(m)}(x)$$

Make the replacement  $c(x) = d(x)g(x)$  to get

$$c^{(m)}(x) = x^m d(x)g(x) - q(x)(x^n - 1)$$

Now make the further replacement  $(x^n - 1) = h(x)g(x)$ ,

$$c^{(m)}(x) = [ x^m d(x) - q(x)h(x) ] g(x).$$

Since  $g(x)$  has degree  $n-k$ , and since  $c^{(m)}(x)$  has degree  $n-1$ , the item in brackets  $[ ]$  must have degree  $k-1$ . It is just some data polynomial we will define as  $d_m(x)$ ,

$$d_m(x) = [ x^m d(x) - q(x)h(x) ]$$

Thus,

$$c^{(m)}(x) = d_m(x) g(x).$$

So, the rotated (cyclically permuted) code word  $c^{(m)}(x)$  results from encoding the data word  $d_m(x)$ , so  $c^{(m)}(x)$  must therefore be a code word. Q.E.D.

## 7.5 Cyclic Code as an Ideal of the Ring $A_n$

Consider the following residue class ring:

$$A_n = \text{Polys}[x, \text{GF}(q)] / (x^n - 1)$$

We studied this type of structure in detail in Chapter 3. The elements of  $A_n$  are the rows of a chart, and each row can be labelled as  $\{ f(x) \}$ , where  $f(x)$  is a polynomial of degree  $n-1$  or less.

Consider the set of code words of a cyclic code generated by some generator  $g(x)$ . Since the codewords are by definition multiples of  $g(x)$ , we make the corresponding statement about elements of  $A_n$ ,

$$\{ c(x) \} = \{ d(x) \} \bullet \{ g(x) \}.$$

**Fact 10:** The  $\{ c(x) \}$  form an *ideal* within  $A_n$ . In other words, the code words of a cyclic code comprise an ideal within the set of polynomials of degree  $n-1$ . We shall denote this ideal as  $( g(x) )$ .

**Proof:** We have to show that  $r \bullet I = I$ . This means that  $\{ r(x) \} \bullet \{ i(x) \} = \{ i'(x) \}$  for any  $\{ r(x) \} \in A_n$  and any  $\{ i(x) \} \in ( g(x) )$ . According to the definition of the above ideal, we have

$$\{ i(x) \} = \{ d(x) \} \bullet \{ g(x) \} \quad \text{for some } d(x)$$

Thus,

$$\{ r(x) \} \bullet \{ i(x) \} = \{ r(x) \} \bullet ( \{ d(x) \} \bullet \{ g(x) \} ) = ( \{ r(x) \} \bullet \{ d(x) \} ) \bullet \{ g(x) \}$$

But  $(\{r(x)\} \bullet \{d(x)\})$  is some  $\{d'(x)\}$  in  $A_n$  so we have

$$\{r(x)\} \bullet \{i(x)\} = \{d'(x)\} \bullet \{g(x)\} = \{i'(x)\}$$

where  $\{i'(x)\}$  is some element of the ideal  $(g(x))$ , Q.E.D.

**Fact 9 Revisited:** If  $g(x)$  divides  $x^n - 1$ , then any cyclic permutation of a code word of the cyclic code generated by  $g(x)$  is also a code word.

**Proof:** Here we are giving an "alternate proof". Actually, it is the same proof as above, only here it is expressed in the language of rings and ideals.

A code word  $c(x)$  is a polynomial of degree  $n-1$ . Thus, according to Math Lemma 2 above,

$$c^{(m)}(x) = \text{Rem}[(x^m c(x)) / (x^n - 1)]$$

where  $c^{(m)}(x)$  is a cyclic permutation of  $c(x)$  by  $m$  places. Since  $c(x)$  is of degree  $n-1$ , we know that  $\{c(x)\} \in A_n$ . If  $m < n$ , then  $\{x^m\} \in A_n$ . Thus, we can rewrite the above equation as

$$\{c^{(m)}(x)\} = \{x^m\} \bullet \{c(x)\}$$

Because  $\{c(x)\}$  is an element of the ideal  $(g(x))$ , and because  $\{x^m\}$  is in  $A_n$ , it follows from the definition of an ideal that  $\{c^{(m)}(x)\}$  is also in the ideal  $(g(x))$ . Thus,  $\{c^{(m)}(x)\}$  must be some code word.

So, we have an interesting new way to view a cyclic code:

**Fact 10':** A cyclic code with symbols in  $GF(q)$ , and which is generated by some  $g(x)$  which divides  $x^n - 1$ , forms an ideal  $(g(x))$  of the ring  $A_n = \text{Polys}[x, GF(q)] / (x^n - 1)$ . This ideal consists of the set of  $q^k$  code words generated by multiplying data polynomials  $d(x)$  by  $g(x)$ ,

$$\{c(x)\} = \{d(x)\} \bullet \{g(x)\}$$

### The Standard Array.

We can now take our ring  $A_n$ , and do a coset (residue class) decomposition of it using the ideal  $(g(x))$  as the top row of the chart. The ring so obtained is called the standard array for the code:

$$\text{Standard Array} = A_n / (g(x)) = \{ (\text{Polys}[x, GF(q)] / (x^n - 1)) / (g(x))$$

Since the ideal  $(g(x))$  contains the  $q^k$  code vectors, every subsequent row of the chart also has  $q^k$  elements. The total number of elements in all rows is  $q^n$ , which is the total number of  $n$ -tuples. Thus, the total number of rows including the ideal is  $q^n/q^k = q^{n-k}$ .

We know that any element of a subsequent row can be written as

$$\{ f(x) \} = \{ \text{coset leader of this } j^{\text{th}} \text{ row} \} + \{ c_i(x) = \text{code word along top of chart} \}$$

or more symbolically

$$f = k_j + c_i$$

If we now apply the matrix  $H^T$ , we find that all elements of a given row have the same syndrome:

$$(k_j + c_i) H^T = k_j H^T = s$$

So this is what all elements of a row of the standard array have in common: they have the same syndrome. One usually chooses as coset leaders the vectors which have a minimum possible number of non-zero components. These then represent the most probable error patterns. We can then arrange some kind of lookup table so that each syndrome pattern looks up its error pattern, and this is then added to a stored erroneous code word to produce the corrected code word. As just noted above, there are  $q^{n-k} - 1$  subsequent rows in the standard array, so this many error patterns must be selected.

## 7.6 The Connection between Cyclic Codes and Galois Fields

So far in this Chapter we have already made a modest connection between cyclic codes and Galois Fields  $GF(q)$ . We have suggested the the code symbols -- the coefficients of the polynomials  $c(x)$ ,  $d(x)$ , and  $g(x)$  -- can be thought of as elements of  $GF(q)$ .

Now we are going to make a much stronger connection between a certain very powerful class of cyclic codes and the Galois Fields  $GF(q)$ . This connection is motivated by the following two observations:

(1) According to the definition given in Section 7.2, a cyclic code  $(n,k)$  is defined by any polynomial  $g(x)$  (order  $n-k$ ) which divides evenly into  $x^n - 1$ .

(2) According to Big Theorem 2 of Chapter 4, the polynomial  $x^{q-1} - 1$  can be fully factored into linear factors each of which contains a root of Galois Field  $GF(q)$ . We write this as:

$$(x^{q-1} - 1) = (x - \alpha_1) \bullet (x - \alpha_2) \bullet (x - \alpha_3) \bullet (x - \alpha_4) \bullet \dots \bullet (x - \alpha_{q-1})$$

These two observations scream out the following suggestion:

**Suggestion:** Why not take  $g(x)$  to be any subset of  $n-k$  of these factors. Such a  $g(x)$  will then generate a cyclic code  $(n,k) = (q-1,k) = (p^m - 1,k)$ . We shall call this a Galois-induced cyclic code.

Let  $g(x)$  then have the following form:

$$g(x) = (x - \alpha_1) \bullet (x - \alpha_2) \bullet (x - \alpha_3) \dots (x - \alpha_{n-k})$$

What can we say about this particular  $g(x)$ ?

- (1) it has  $n-k$  roots in  $GF(q)$ , that is,  $g(\alpha_1) = g(\alpha_2) = g(\alpha_3) \dots = 0$
- (2) it has degree  $n-k$
- (3) the coefficients of  $g(x)$  lie in  $GF(q)$

According to the definition of a cyclic code, the code word polynomials will be formed as

$$c_i(x) = d_i(x)g(x) \quad i = 1, 2, 3, \dots, q^k$$

It therefore follows that all the code words of such a cyclic code will have at least the same  $GF(q)$  roots  $\alpha_i$  that  $g(x)$  has. It also follows that the code symbols, being coefficients of  $c_i(x)$ , will lie in  $GF(q)$

If we happen to want these code symbols to lie in the ground field  $GF(p)$ , we would have to take combinations of factors  $(x - \alpha_i)$  that form the "minimum polynomials" discussed in Chapter 5.

**Fact 11:** The parity check matrix  $H$  for such a cyclic code has the following form:

$$H = \begin{bmatrix} 1 & (\alpha_1) & (\alpha_1)^2 & (\alpha_1)^3 & \dots & (\alpha_1)^{n-1} \\ 1 & (\alpha_2) & (\alpha_2)^2 & (\alpha_2)^3 & \dots & (\alpha_2)^{n-1} \\ 1 & (\alpha_3) & (\alpha_3)^2 & (\alpha_3)^3 & \dots & (\alpha_3)^{n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & (\alpha_{n-k}) & (\alpha_{n-k})^2 & (\alpha_{n-k})^3 & \dots & (\alpha_{n-k})^{n-1} \end{bmatrix}$$

**Proof:** By definition,  $H$  is the matrix which kills all code vectors,  $H \bullet c^T = 0$ . Think of the components of  $c^T$  as the coefficients of the polynomial  $c(x)$ . If we multiply the  $i^{\text{th}}$  row of the above  $H$  matrix by  $c^T$ , we get:

$$c_0 + c_1 (\alpha_i) + c_2 (\alpha_i)^2 + c_3 (\alpha_i)^3 + \dots + c_{n-1} (\alpha_i)^{n-1}$$

But this is exactly  $c(\alpha_i)$  which we know is 0, since  $\alpha_i$  is a root of  $g(x)$ . Thus, the above matrix  $H$  has the right property that  $H \bullet c^T = 0$ . Also, it has the right number of rows ( $n-k$ ) and columns ( $n$ ). The elements of  $H$  are symbols, ie, elements of  $GF(q)$ . And  $H$  is the generator of the dual code  $(n, n-k)$ .

We now have a *huge* number of possible Galois-induced cyclic codes on our hands. It turns out that some of these codes are much more interesting than others because they have more "performance". In particular, as we shall see in the next section, there is a theorem which lets us pick out the codes which have the best random error correction capability. These codes form the so-called BCH code family, which happens to contain the narrow-sense BCH codes as well as the Reed-Solomon codes. A subset of the narrow-sense BCH codes consists of the Hamming Codes. Here is the general relationship:

BCH codes

|                            |            |         |
|----------------------------|------------|---------|
| The narrow-sense BCH Codes | $t \geq 0$ | $GF(q)$ |
| The Hamming Codes          | $t = 0, 1$ | $GF(q)$ |
| The Reed-Solomon codes     | $t \geq 0$ | $GF(q)$ |

The  $t$  column indicates the amount of random error correction possible, and the last column shows the space of the code symbols. Most useful cases have  $p=2$ , since this is what computers deal with.

## 7.7 The BCH Codes

BCH are the initials of Bose and Ray-Chaudhuri (1960) and independently Hocquenghem (1959).

Let us return to the general proposal described above. In order to have all code words of a  $GF(q)$ -induced cyclic code vanish at a set of roots  $\alpha_1, \alpha_2, \dots, \alpha_N$ , we should arrange to have these be the roots of the generator polynomial  $g(x)$ , since the code words are obtained as  $c(x) = d(x)g(x)$ . According to Big Theorem 1 of Chapter 4, we know that  $GF(q)$  is cyclic, and we can therefore represent all its non-zero elements as powers  $\alpha^k$  of a primitive element  $\alpha$ . Thus, we can rewrite the above list of roots as:

$$\{\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_N\} = \{\alpha^{e_1}, \alpha^{e_2}, \alpha^{e_3}, \dots, \alpha^{e_N}\}$$

where the  $e$ 's are whatever exponents we need to get  $\alpha_1, \alpha_2, \dots, \alpha_N$ .

We now wish to quote a very strange theorem about the minimum distance, hence error correcting ability of, this Galois-induced cyclic code. You will have to read this very carefully:

**The BCH Bound :** The minimum distance  $d$  of the cyclic code whose generator  $g(x)$  has some set of roots in  $GF(q)$ , namely  $\{\alpha^{e_1}, \alpha^{e_2}, \alpha^{e_3}, \dots, \alpha^{e_N}\}$ , is *greater than* the largest number of *consecutive* integers (modulo the order of  $\alpha$ ) in the power list:  $\{e_1, e_2, e_3, \dots, e_N\}$ .

**Comment:** This theorem, known as *The BCH Bound*, is providing a *lower* bound on the minimum distance of the code, that is, a lower limit on the number of errors per codeword that can be corrected. Recall that  $d \sim 2t+1$ . This suggests that we might be able to find some codes that will correct large numbers of errors in a code word.

One proof of the BCH Bound involves examining the  $H$  matrix shown above in Fact 11, and showing that  $d =$  (the minimum number of linearly dependent columns) exceeds the strange number noted in the theorem. Recall Fact 4 above. For a proof, see Peterson and Weldon Chapter 9.

From this bound, we are immediately motivated to select powers that are in sequence, rather than some set of random powers of  $\alpha$ , due to the word "consecutive" appearing in the theorem. Also, we might like to have the "order of  $\alpha$ " be as large as possible, since our  $d$  is going to be *modulo* this number. The optimal solution would seem to let  $\alpha$  be a primitive element, since then its order is the entire size of the group  $\{GF(q) - 0, \bullet\}$ , which is  $q-1$ . In general,  $\alpha$  could be any element of  $GF(q)$ .

So following this line of attack, we can make a simpler version of the above theorem:

**BCH Bound Corollary :** If we select the roots of  $g(x)$  to be  $\{\alpha^a, \alpha^{a+1}, \alpha^{a+2}, \dots, \alpha^{a+k}\}$ , this list has  $k+1$  consecutive powers, so the minimum distance of the implied cyclic code is  $d > k+1$ , provided  $k+1$  does not exceed the order of  $\alpha$ . It will be more convenient to restate this as  $d \geq k+2$ .

**Definition:** Such codes, characterized by some  $\alpha \in GF(q)$  and integers  $a, k$  are known as *the BCH codes*.

## 7.8 The Narrow-Sense BCH Codes

**Definition:** The *narrow-sense BCH codes* are those BCH codes for which  $a=1$ ,  $\alpha$  is a primitive element of  $GF(q)$ ,  $g(x)$  contains the roots  $\{ \alpha, \alpha^2, \alpha^3, \dots, \alpha^N \}$ , and  $g(x)$  has coefficients in  $GF(p)$ .

This last requirement usually means that there are many more roots as well, as we shall see below. Nevertheless, the BCH Bound Corollary above still applies. According to the Corollary, this code has  $d \geq N+1$ . Thus, we can define a so-called design minimum distance of the code  $d_{\text{design}} = N+1$ , and then  $d \geq d_{\text{design}}$ . This says that the true minimum distance of one of the BCH codes is at least as large as  $d_{\text{design}} = N+1$ . To summarize,

Narrow-Sense BCH Codes:

roots of  $g(x)$  include  $\{ \alpha, \alpha^2, \alpha^3, \dots, \alpha^N \}$ ,  $\alpha$  = a primitive element of  $GF(q)$

$g(x)$  includes whatever *other* roots are needed such that  $g(x)$  has coefficients in  $GF(p)$

$n = q-1 = p^m - 1$        $n-k$  = order of  $g(x)$

$d \geq d_{\text{design}}$       where       $d_{\text{design}} = N+1$

$t \geq t_{\text{design}}$       where       $t_{\text{design}} = \text{Int}(N/2)$       [from Fact 2]

From our work in Chapter 5, we know immediately how to construct a generator  $g(x)$  which has coefficients in  $GF(p)$  and which has the roots  $\{ \alpha, \alpha^2, \alpha^3, \dots, \alpha^N \}$  in  $GF(q)$ . Recall that the minimum polynomial  $m(x)$  of  $\alpha$  has  $\alpha$  as a root, and has coefficients in  $GF(p)$ . Of course it also has all those conjugate roots as well. Therefore, an obvious candidate for  $g(x)$  is to take the product of the minimum polynomials of all the roots in the list. Due to the conjugates, it may happen that several roots have the same  $m(x)$ . In this case, we want to keep only one copy of this  $m(x)$ , because we must make sure that  $g(x)$  divides into  $x^n - 1$ , where  $n = q-1$  for  $GF(q)$ . Recall as noted above that we are just gathering up a subset of factors  $(x-\alpha_i)$ , and that  $x^n-1$  contains all these factors (Big Theorem 2, Chapter 4). Thus, we propose

$$g_N(x) = \text{LCM}[m_1(x)m_2(x)m_3(x)m_4(x) \dots m_N(x)] \text{ arbitrary } p$$

where LCM means least common multiple, just a way of saying throw out duplicate  $m_i(x)$ . Note that the order of  $g(x)$  is highly dependent on the roots.

The case  $N = 1$  gives  $g(x) = m_1(x)$  all by itself. These are the Hamming Codes which we discuss in a separate section below.

In Chapter 5 we noted that  $m_1(x)$  contains all roots of the conjugate set of  $\alpha$ , which consists of the distinct elements of this set:

$$\{ \alpha, \alpha^p, \alpha^{p^2}, \alpha^{p^3} \dots \alpha^{p^{m-1}} \} \quad \alpha^{p^m} = \alpha^q = \alpha \quad m \text{ elements}$$

Therefore, as you build up the  $g_N(x)$ , there will be repeats. In general, every  $p^{\text{th}}$   $m_i(x)$  will be a duplicate of some earlier  $m_i(x)$ , so every  $p^{\text{th}}$  one can be thrown out. For example,  $m_p(x)$  contains the root  $\alpha^p$ , but so does  $m_1(x)$ , as shown above.

For the special case  $p=2$ , this means that every *even*  $m_i(x)$  can be thrown out, so we are left with:

$$g_N(x) = g_{N+1}(x) = \text{LCM}[m_1(x)m_3(x)m_5(x)m_7(x) \dots m_N(x)] \quad N \text{ odd, } p = 2 \text{ only}$$

The general procedure for developing BCH codes is as follows.

- (1) Select  $p$  and  $m$ , so you have  $n = q-1 = p^m - 1$ . You have now selected a particular Galois Field  $GF(p^m)$ , as described in much detail in Chapter 4.
- (2) Second, construct the "Table" for  $GF(p^m)$  exactly as described in Chapter 6. As noted there, this table implicitly contains the entire algebra of the field, that is, it implies the complete  $\bullet$  and  $+$  field operation tables. You then know how to multiply or add any two elements of the field  $GF(p^m)$ .
- (3) for each power  $\alpha^k$  construct the minimal polynomial  $m_k(x)$ . An example of doing this was given in Chapter 6, with more details in Chapter 5.
- (4) Then for each value of  $N = 1, 2, 3, \dots$  compute  $g(x)$  exactly as shown above.

$$g_N(x) = \text{LCM}[m_1(x)m_2(x)m_3(x)m_4(x) \dots m_N(x)]$$

For each  $N$ , you find that  $g(x)$  has some degree  $n-k$ , so for each  $N$ , you get a value of  $k$ .

There are some implied limits on  $N$ ,  $t_{\text{design}}$  and  $k$ .

If the field  $GF(q)$  is enumerated as powers of a primitive element  $\alpha$ , the highest power is  $\alpha^{q-2}$ , as shown in Chapter 4 after Big Theorem 1. This power is  $\alpha^{n-1}$ . Thus, according to the above expression for  $g(x)$ , we must have  $N < n$ . Thus,  $t_{\text{design}} < \text{Int}(n/2)$  [ $\leq$  if  $n$  odd].

Next, we know that each minimum polynomial  $m_i(x)$  has degree  $\leq m$ . The degree is  $m$  if  $\alpha^i$  is a primitive element of  $GF(q)$ . Since  $n-k$  is the degree of  $g(x)$ , all we can say is that  $n-k \leq mt$ . If any minimum polynomials have fewer than  $m$  distinct roots, or if any minimum polynomials repeat, as in the example below,  $n-k$  will be less than this upper bound of  $mt$ .

Here is a summary of these limits:

$$N < n \quad t_{\text{design}} \leq \text{Int}(n/2) \quad n-k \leq mt$$

**Example:**  $GF(32 = 2^5)$ . We take advantage of the details presented in Rhee. The Table (see Chapter 6 of this report) for  $GF(32)$  is depicted on Rhee page 171, we will not duplicate it here. Here is a list of the minimum polynomials:

$$\begin{aligned}
 m_1(x) &= (x - \alpha) (x - \alpha^2) (x - \alpha^4) (x - \alpha^8) (x - \alpha^{16}) \\
 m_3(x) &= (x - \alpha^3) (x - \alpha^6) (x - \alpha^{12}) (x - \alpha^{24}) (x - \alpha^{17}) \\
 m_5(x) &= (x - \alpha^5) (x - \alpha^{10}) (x - \alpha^{20}) (x - \alpha^9) (x - \alpha^{18}) \\
 m_7(x) &= (x - \alpha^7) (x - \alpha^{14}) (x - \alpha^{28}) (x - \alpha^{25}) (x - \alpha^{19}) \\
 m_9(x) &= m_5(x) \\
 m_{11}(x) &= (x - \alpha^{11}) (x - \alpha^{22}) (x - \alpha^{13}) (x - \alpha^{26}) (x - \alpha^{21}) \\
 m_{13}(x) &= m_{11}(x) \\
 m_{15}(x) &= (x - \alpha^{15}) (x - \alpha^{30}) (x - \alpha^{29}) (x - \alpha^{27}) (x - \alpha^{23})
 \end{aligned}$$

Based on the above, we can now list off the orders of the various  $g(x)$ . Note for example that since  $m_9(x)$  is a duplicate of  $m_5(x)$ ,  $g_5(x)$  is the same as  $g_4(x)$ , and similarly  $g_7(x) = g_6(x)$ . Note that  $n = 2^5 - 1 = 31$ , and that  $p=2$ , so we can use the simplified LCM formula given above for  $g(x)$ .

| <u>N</u>  | <u>t(design)</u> | <u>generator</u>                          | <u>n-k = order(g)</u> | <u>k</u>  | <u>BCH code</u> |
|-----------|------------------|-------------------------------------------|-----------------------|-----------|-----------------|
| 1         | 0                | $g_1(x) = m_1(x)$                         | 5                     | 26        | (31,26)         |
| <b>2</b>  | <b>1</b>         | <b><math>g_2(x) = g_1(x)</math></b>       | <b>5</b>              | <b>26</b> | <b>(31,26)</b>  |
| 3         | 1                | $g_3(x) = g_1(x)m_3(x)$                   | 10                    | 21        | (31,21)         |
| <b>4</b>  | <b>2</b>         | <b><math>g_4(x) = g_3(x)</math></b>       | <b>10</b>             | <b>21</b> | <b>(31,21)</b>  |
| 5         | 2                | $g_5(x) = g_3(x)m_5(x)$                   | 15                    | 16        | (31,16)         |
| <b>6</b>  | <b>3</b>         | <b><math>g_6(x) = g_5(x)</math></b>       | <b>15</b>             | <b>16</b> | <b>(31,16)</b>  |
| 7         | 3                | $g_7(x) = g_5(x)m_7(x)$                   | 20                    | 11        | (31,11)         |
| 8         | 4                | $g_8(x) = g_7(x)$                         | 20                    | 11        | (31,11)         |
| 9         | 4                | $g_9(x) = g_7(x)$                         | 20                    | 11        | (31,11)         |
| <b>10</b> | <b>5</b>         | <b><math>g_{10}(x) = g_9(x)</math></b>    | <b>20</b>             | <b>11</b> | <b>(31,11)</b>  |
| 11        | 5                | $g_{11}(x) = g_{10}(x)m_{11}(x)$          | 25                    | 6         | (31,6)          |
| 12        | 6                | $g_{12}(x) = g_{11}(x)$                   | 25                    | 6         | (31,6)          |
| 13        | 6                | $g_{13}(x) = g_{11}(x)$                   | 25                    | 6         | (31,6)          |
| <b>14</b> | <b>7</b>         | <b><math>g_{14}(x) = g_{13}(x)</math></b> | <b>25</b>             | <b>6</b>  | <b>(31,6)</b>   |
| 15        | 7                | $g_{15}(x) = g_{13}(x)m_{15}(x)$          | 30                    | 1         | (31,1)          |
| 16        | 8                | $g_{16}(x) = g_{15}(x)$                   | 30                    | 1         | (31,1)          |
| 17        | 8                | $g_{17}(x) = g_{15}(x)$                   | 30                    | 1         | (31,1)          |
| .....     |                  |                                           |                       |           |                 |
| <b>31</b> | <b>15</b>        | <b><math>g_{31}(x) = g_{29}(x)</math></b> | <b>30</b>             | <b>1</b>  | <b>(31,1)</b>   |

In general, we see that several sequential values of  $N$  all generate the same code  $(n,k)$ . We put in bold the bottommost line in each group, since this line displays the maximum possible design  $t$ . Clearly, the other codes in each group must have had  $t > t_{\text{design}}$ . We can therefore compress the above table:

| <u>N</u> | <u>t(design)</u> | <u>generator</u>        | <u>n-k = order(g)</u> | <u>k</u> | <u>BCH code</u> |
|----------|------------------|-------------------------|-----------------------|----------|-----------------|
| 1-2      | 1                | $g_2(x) = g_1(x)$       | 5                     | 26       | (31,26)         |
| 3-4      | 2                | $g_4(x) = g_3(x)$       | 10                    | 21       | (31,21)         |
| 5-6      | 3                | $g_6(x) = g_5(x)$       | 15                    | 16       | (31,16)         |
| 7-10     | 5                | $g_{10}(x) = g_9(x)$    | 20                    | 11       | (31,11)         |
| 11-14    | 7                | $g_{14}(x) = g_{13}(x)$ | 25                    | 6        | (31,6)          |
| 15-31    | 15               | $g_{31}(x) = g_{29}(x)$ | 30                    | 1        | (31,1)          |

Thus, for  $p=2$ ,  $m=5$ , we can identify a set of six BCH narrow-sense codes of length 31. They are able to correct at least 1,2,3,5,7 and 15 errors. The last code with  $k=1$  is rather boring and is usually omitted from any listing of BCH codes. It has only two code words, each is 31 bits. Since the code is cyclic, these two code words must be

010101010101010101010101010101010 and its complement

As already noted, the  $t=1$  code is a Hamming Code.

#### The H matrix for narrow-sense BCH codes.

From Fact 11, the H matrix for a narrow-sense BCH code is the following:

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \dots & \alpha^{n-1} \\ 1 & (\alpha^2) & (\alpha^2)^2 & (\alpha^2)^3 & \dots & (\alpha^2)^{n-1} \\ 1 & (\alpha^3) & (\alpha^3)^2 & (\alpha^3)^3 & \dots & (\alpha^3)^{n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & (\alpha^N) & (\alpha^N)^2 & (\alpha^N)^3 & \dots & (\alpha^N)^{n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & (\alpha_{n-k}) & (\alpha_{n-k})^2 & (\alpha_{n-k})^3 & \dots & (\alpha_{n-k})^{n-1} \end{pmatrix}$$

There are  $n-k$  rows,  $n$  columns, and each matrix element is a symbol lying in  $GF(q)$ . As you go down the second column, we encounter first the roots  $\alpha, \alpha^2, \alpha^3$  up to  $\alpha^N$ , and then we encounter whatever other roots are needed to make the code words come out with coefficients in  $GF(p)$ . We label the last root  $n-k$ . It of course can be written as a power like the other roots are.

If we represent each element of this matrix as a vertical vector having  $m$  components, each of which is an element of  $GF(p)$ , then we can think of the above matrix as having  $m \times (n-k)$  rows of numbers belonging to  $GF(p)$ .

**Comment:** The significance of the BCH codes is that they provide clearcut codes for correcting multi-symbol errors. The Hamming codes for  $p=2$  only correct single bit errors.

We have seen the the BCH codes are completely based on Galois Field theory. The code word polynomials are selected to vanish at  $N$  roots  $(\alpha_1, \alpha_2, \dots, \alpha_N)$  which are elements of  $GF(q)$ . The resulting code can correct at least  $t$ -symbol errors per received code word.

## 7.9 The Hamming Codes

The Hamming Codes are those narrow-sense BCH codes having  $N = 1$ . The generator  $g(x)$  for a Hamming Code is therefore given by:

$$g(x) = m_1(x) = p(x)$$

where  $m_1(x)$  is the minimum polynomial of  $\alpha$ . Since  $\alpha$  is a primitive element of  $GF(q)$ , this polynomial is a primitive polynomial of  $GF(q)$ , which we call  $p(x)$ . (See Chapters 5 and 6.) The order of  $p(x)$  is  $m$ , according to Fact 10 of Chapter 5.

As already noted,  $n = q-1$ . Thus, we have a set of cyclic codes for which (arbitrary  $p$ ):

$$\begin{aligned} n &= p^m - 1 & (p = \text{prime}, m = \text{integer}) \\ n-k &= m \\ k &= n - (n-k) = p^m - 1 - m \end{aligned}$$

According to the narrow-sense BCH codes bound for  $N=1$ , we conclude that  $t_{\text{design}} = 0$ , which is not very impressive.

### Hamming codes for $p=2$ :

For  $p=2$ , the above formulas reduce to:

$$\begin{aligned} n &= 2^m - 1 \\ n-k &= m \\ k &= n - (n-k) = n-m = 2^m - 1 - m \end{aligned}$$

In this case, we know from the previous section that  $N = 2$  yields the *same code* as  $N=1$ , due to the "throw out the evens" rule for  $g_N(x)$ . And we know from the BCH bound that an  $N=2$  code has  $t_{\text{design}} = 1$ . Thus, all  $p=2$  Hamming codes can correct single-bit errors. In contrast, as we saw above, the Hamming codes for  $p>2$  are likely to have no error correction capability at all.

These codes with  $p=2$  are the classic Hamming Codes discovered by Hamming in 1950. They all correct single bit errors.

In the Standard Array for such a Hamming code, there are  $2^n/2^k = 2^{n-k} = 2^m = n+1$  rows. Thus, there are  $n$  rows or cosets below the ideal. Since code words are  $n$  bits, *all* coset leaders can be exactly chosen as the  $n$  single-bit error patterns. Thus, single bit error correction is very easy to do. Codes which have this property are very few indeed, and are known as *perfect codes*.

**Comment:** When one first sees the Hamming Codes, the fact that  $n = 2^m - 1$  (and  $n-k = m$ ) seems very, very strange. One has no clue as to where these peculiar numbers are coming from. In retrospect, these numbers are only too familiar from our study of Galois Fields. The number  $n = q-1$  is the number of non-zero elements in  $GF(q=2^m)$ , and  $m$  is just the Galois power. The Hamming cyclic code generator  $g(x)$  is a

primitive polynomial of  $GF(2^m)$  which we know has degree  $m$ . The powers appearing in the  $H$  matrix are powers of a primitive element  $\alpha$  of  $GF(2^m)$ , and we have  $g(\alpha) = 0$ .

### The $H$ matrix for the Hamming Codes.

The  $H$  matrix is given by:

$$H = \begin{matrix} & \begin{matrix} 1 & \alpha & \alpha^2 & \alpha^3 & \dots & \alpha^{n-1} \end{matrix} \\ \begin{matrix} 1 \\ 1 \\ 1 \\ 1 \\ \dots \\ 1 \end{matrix} & \begin{pmatrix} (\alpha^p) & (\alpha^p)^2 & (\alpha^p)^3 & \dots & (\alpha^p)^{n-1} \\ (\alpha^{p^2}) & (\alpha^{p^2})^2 & (\alpha^{p^2})^3 & \dots & (\alpha^{p^2})^{n-1} \\ (\alpha^{p^3}) & (\alpha^{p^3})^2 & (\alpha^{p^3})^3 & \dots & (\alpha^{p^3})^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ (\alpha^{p^{m-1}}) & (\alpha^{p^{m-1}})^2 & (\alpha^{p^{m-1}})^3 & \dots & (\alpha^{p^{m-1}})^{n-1} \end{pmatrix} \end{matrix}$$

As usual, there are  $n-k$  rows and  $n$  columns, and the matrix elements are symbols in  $GF(q=p^m)$ . If we represent each element of this matrix as a vertical vector having  $m$  components, each of which is an element of  $GF(p)$ , then we can think of the above matrix as having  $m^2$  rows of numbers belonging to  $GF(p)$ .

Each row in the above matrix corresponds to a code word  $c(x)$  vanishing at some root  $\alpha$ . We can consider just the first row of the above matrix, corresponding to  $c(x)$  vanishing only at the primitive element  $\alpha$ . Then we would write this subset of the rows of matrix  $H$  as:

$$H_1 = [1, \alpha, \alpha^2, \alpha^3, \dots, \alpha^{n-1}]$$

If we represent the elements as vertical  $m$ -tuples, we then have a matrix of numbers in  $GF(p)$  that has  $n-k = m$  rows and  $n$  columns. It is certainly true that  $H_1 \bullet c^T = 0$ . Thus, we can think of  $H_1$  as one of a set of  $m$  similar matrices (the rows of the large matrix shown above) each of which has elements in  $GF(p)$ , and has  $n-k = m$  rows and  $n$  columns. Each such matrix corresponds to  $c(\alpha_i) = 0$  for one root of the conjugate root set containing  $\alpha$ .

In the case that  $c(x)$  has coefficients in  $GF(p)$ , as is the case with all narrow-sense BCH codes, one can think of the vector  $c$  as being collapsed down from a vector of  $n$  symbols in  $GF(q)$ , to a vector of  $n$  symbols in  $GF(p)$ , which are just the numbers of  $Z_p$ . In other words, in  $m$ -tuple notation, an element of  $GF(q)$  which lies in  $GF(p)$  has the format  $a(1,0,0,\dots)$ . It is a multiple of the identity of  $GF(q)$ .

In this collapsed sense, we can apply each of the  $m$  rows of  $H_1$  (which contain numbers in  $Z_p$ ) to the collapsed code vectors, each of which contains  $n$  number in  $Z_p$ . Thus, we get a collapsed version of the larger picture.

**Full Picture:** Matrix  $H$  has  $n-k$  rows of symbols in  $GF(q)$  and acts on  $n$ -dimensional code vectors  $c$ , each component of which is a symbol in  $GF(q)$ .

**Collapsed Picture:** Matrix  $H_1$  has  $n-k$  rows of symbols in  $GF(p)$  and acts on  $n$ -dimensional code vectors  $c$ , each component of which is a symbol in  $GF(p)$ .

In the collapsed world, we can think of  $H_1$  as the parity check matrix, and we can think of it as generating the dual code  $(n, n-k)$ . But the same can be said of the full scale  $H$ .

This fact that there exists a collapsed form of the  $H$  matrix that mimics the full form seems to be a quirk of the Hamming codes. It happens to work because  $n-k = m$ . That is, the size of the  $m$ -tuples in  $GF(q)$  equals  $n-k$ . The other requirement is that  $c(x)$  have coefficients in  $GF(p)$ . This requirement is met by all narrow-sense BCH codes, but the first is only met by the Hamming codes.

### Why are Hamming codes with $p > 2$ unable to correct single symbol errors?

The collapsed version the parity check matrix  $H_1$  gives us some insight as to why  $t=0$  when  $p>2$ . We know that there are  $p^m$  different  $m$ -tuple elements of  $GF(q)$ . If we exclude those that are multiples of each other, there are then only  $p^m/(p-1)$   $m$ -tuples. In the construction of  $H_1$ , if we have to use any  $m$ -tuples which are multiples of each other, we will have two columns which are linearly dependent. From Fact 4, this means that  $d = 1$ , so  $t = 0$ .

When  $p > 3$ , there are simply not enough "no-multiples"  $m$ -tuples available to fill out all  $n$  columns of  $H_1$ . This is because

$$p^m/(p-1) < n = p^m \quad \text{that is} \quad 1 < p-1$$

In the case  $p=2$ , there are exactly enough  $m$ -tuples to fill out the columns, so all are used, and  $t = 1$ .

### Modified Hamming Codes

To improve on the  $p>2$  Hamming codes and their poor EC capability, one can do the following trick, as noted by Peterson and Welson, page 221. Instead of using a primitive element  $\alpha$  as the defining root, use instead the element  $\beta = \alpha^{p-1}$ . Whereas  $\alpha$  has order  $p^m - 1$ ,  $\beta$  has order  $(p^m - 1)/(p-1)$ . It turns out that there are still  $m$  distinct elements in the conjugate set of  $\beta$ , so  $n-k = m$ .  $H_1$  can then be constructed as

$$H_1 = [ 1, \beta, \beta^2, \beta^3, \dots, \beta^{n-1} ] \quad n = (p^m - 1)/(p-1).$$

Now the number of columns has been reduced so that it exactly matches the number of "no multiples"  $m$ -tuples. Since there are no direct multiples,  $d = 3$ , and  $t = 1$  for these codes. Thus, we have a form of the Hamming codes that have the same EC capability as the  $p=2$  Hamming codes.

### Modified Hamming Codes

$g(x)$  = minimum polynomial of  $\beta = \alpha^{p-1}$ ,  $g(\beta) = 0$ ,  $\alpha$  = primitive element of  $GF(p^m)$

$$\begin{aligned} n &= (p^m - 1)/(p-1) & \beta &= \alpha^{p-1} H_1 = [1, \beta, \beta^2, \beta^3, \dots, \beta^{n-1}] \\ n-k &= m \\ k &= n - (n-k) = n-m \end{aligned}$$

For  $p=2$ , these reduce to the normal  $p=2$  Hamming codes.

## 7.10 The Reed-Solomon Codes

We have seen that the narrow-sense BCH codes are a special case of the general class of BCH codes. The Reed-Solomon codes are another special case.

The reason we chose the minimum polynomials for the narrow-sense BCH codes was to make sure  $g(x)$  had coefficients in the field  $GF(p)$ . Suppose we consider generators  $g(x)$  that have coefficients in  $GF(q) = GF(p^m)$ . In this case, there is no need to fool around with the minimum polynomials, and we have a much simpler expression for a generator containing the desired roots. As with the narrow sense BCH codes, we set  $a = 1$  and select  $\alpha$  to be a primitive element of  $GF(q)$ . Thus, our candidate  $g(x)$  in  $GF(q)$  is simply this:

$$g_N(x) = (x - \alpha)(x - \alpha^2)(x - \alpha^3) \dots (x - \alpha^N) \quad g(x) \text{ has coefficients in } GF(p^m)$$

Notice that there is no uncertainty about the degree of  $g_N(x)$  as there is in the narrow-sense BCH codes. The degree here is  $n-k = N$ .

If  $g(x)$  has coefficients in  $GF(p^m)$ , then so do the code polynomials  $c(x)$ . We can assume the same for the data polynomials  $d(x)$ .

The error correcting capability of the above code is exactly the same as for the corresponding narrow sense BCH code, and for the same reason: an application of the BCH Bound Corollary. Thus we have,

$$\begin{aligned} d &\geq d_{\text{design}} & \text{where} & & d_{\text{design}} &= N+1 \\ t &\geq t_{\text{design}} & \text{where} & & t_{\text{design}} &= \text{Int}(N/2) \quad [\text{from Fact 2}] \end{aligned}$$

**Example:**  $p=2$  and  $m=8$ ,  $n = 2^8 = 256$ . The data symbols and code symbols are then *bytes*. Each code words contain 256 bytes. If  $t_{\text{design}} = 3$  for some code  $N$ , then up to 3 erroneous bytes in such a code words can be corrected.

**Definition:** Codes with code symbols in  $p^m$  and with  $g(x)$  as shown above are known as the Reed-Solomon codes (1960).  $\alpha$  is a primitive element of  $GF(p^m)$  and the code corrects  $t$  errors.

For the special case  $p=2$ , the parameters of a Reed Solomon code are as follows:

$$\begin{aligned}
n &= 2^m - 1 \\
n-k &= N \\
k &= 2^m - 1 - N = n-N
\end{aligned}$$

Recall from Section 7.1 ( Fact 3 ) the upper bound on error correction capability of any code:

$$t \leq \text{Int}[(n-k)/2]$$

For the arbitrary-p Reed-Solomon codes we have

$$t_{\text{design}} = \text{Int}(N/2) = \text{Int}[(n-k)/2]$$

Thus, all Reed-Solomon codes saturate this bound. This is a very strong strong selling point for R-S codes in general. There can exist no other codes which can correct more errors for a given n-k. Reed Solomon codes are *maximum-distance-separable*.

In terms of the (n,k) notation, we have the Reed-Solomon codes specified as:

$$(n,k) = (n,n-N) \text{ where } n = p^m - 1$$

Here is a list of the R-S codes for p=2 and m=3 (code symbol = 3-bit nibble)

| <u>m</u> | <u>n</u> | <u>N</u> | <u>t(design)</u> | <u>k</u> | <u>(n,k)</u> |
|----------|----------|----------|------------------|----------|--------------|
| 3        | 7        | 1        | 0                | 6        | (7,6)        |
| <b>3</b> | <b>7</b> | <b>2</b> | <b>1</b>         | <b>5</b> | <b>(7,5)</b> |
| 3        | 7        | 3        | 1                | 4        | (7,4)        |
| <b>3</b> | <b>7</b> | <b>4</b> | <b>2</b>         | <b>3</b> | <b>(7,3)</b> |
| 3        | 7        | 5        | 2                | 2        | (7,2)        |
| <b>3</b> | <b>7</b> | <b>6</b> | <b>3</b>         | <b>1</b> | <b>(7,1)</b> |

There seems little reason to use a code with odd N, since the next lower even N code has the same t(design) and you get one more data symbol and need one less parity symbol. For this reason, one often sets  $N = 2t$  and summarizes the codes as:

$$(n,k) = (n,n-2t) \text{ where } n = p^m - 1$$

Obviously, there are a *lot* of Reed-Solomon Codes.

### Comparison to a Digital Filter

Another selling point of p=2 R-S codes is that one can work with polynomial dividers and related circuits that have data paths that are m-bits wide. Hardware is well geared for doing this. In fact, all these circuits are identical to circuits used as digital filters with one exception which is worth pointing out.

In an integer-math digital filter with data paths m-bits wide, the + and • tables are those of modulo-N arithmetic, where  $N = 2^m$ . The field in which the numbers reside is  $Z_N$ .

In a R-S polynomial divider or multiplier with data paths  $m$ -bits wide ( as one would encounter in an encoder or decoder), the  $+$  and  $\bullet$  tables are those of the field  $GF(N=2^m)$ .

In  $Z_N$ , there is "carry" between the bit positions during addition. In  $GF(N)$ , there is no carry, each bit is added independently in  $GF(2)$ .

Earlier it was stressed that the multiplication table for  $Z_4$  is different from that of  $GF(4)$ . Thus is true in general for any  $N = 2^m$  (except  $m=1$ ). In both  $Z_N$  and  $GF(N)$  there is an interaction between all the bit lines during a multiplication.

Thus, in building R-S circuits, one can use XOR gates for addition independently on the individual bit lines, but multiplication must be done with some kind of lookup or combinatoric circuit which computes correctly the products of  $GF(2^m)$ . As in a digital filter, multiplication cannot be done independently on the individual bit lines.

**Appendix 7.1:** Figures showing row and column index ranges for G and H (for Section 7.1, Fact 0)

