EE521 Analog and Digital Communications

Instructor: James K Beard, PhD **Office:** Ft. Washington 115

Email:. jkbeard@temple.edu, jkbeard@comcast.net

Office Hours: Wednesdays 5:00 PM to 6:00PM

Location: Ft. Washington 107 **Time:** Wednesdays 6:00 PM – 8:30 PM

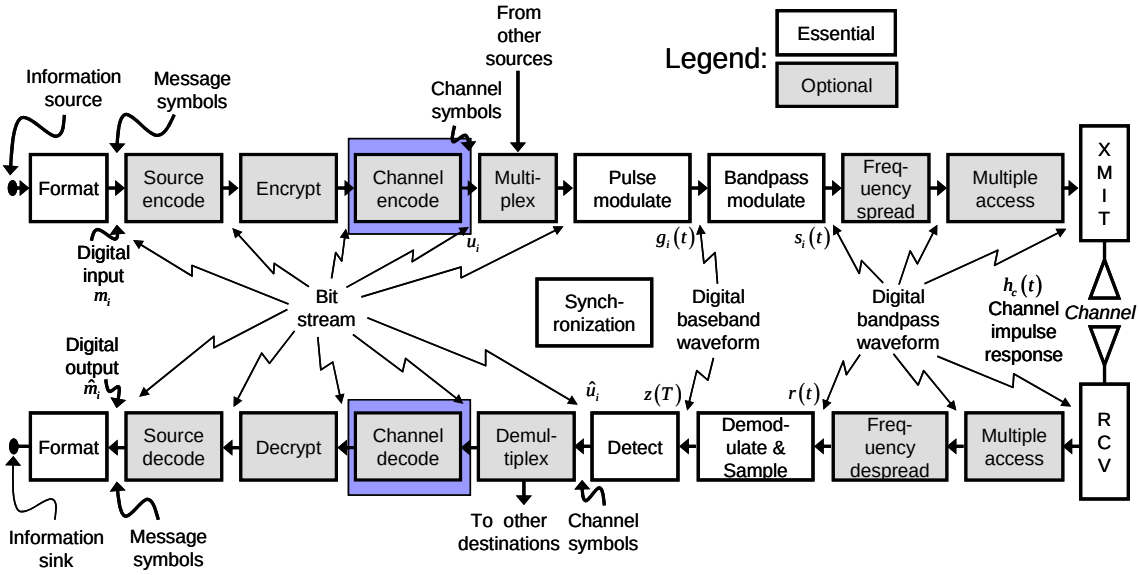
Web Page: http://temple.jkbeard.com

Texts:

- Bernard Sklar, Digital Communications, Second Edition, Prentice Hall P T R, 2001 (2004 printing), ISBN 0-13-084788-7
- Digital Communication Systems Using SystemVue, by Dr. Silage, ISBN 1-58-450850-7

Today's Topics

Channel Coding: Part 1



1 Example: Probability of Error for Coded and Uncoded Messages

We have a (8,4) code with t=2, and a BER of 10^{-2}. What is the probability of an uncorrected error, with and without encoding?

The probability that N bits will have less than a errors in a channel with a BER of p is

$$P_M = \sum_{j=a+1}^{N} \binom{N}{j} \cdot p^j \cdot (1-p)^{N-j} \tag{1.1}$$

where

$$\binom{N}{j} = \frac{N!}{j!\cdot(N-j)!} \tag{1.2}$$

is the binomial coefficient. Note that Excel implements the binomial coefficient as the combinatorial function, COMBIN(N,j).

See the Excel spreadsheet for the computation. The results are a word error rate of $5.39\cdot10^{-5}$ for the coded word and 0.039 for the uncoded word, an improvement by a factor of about 730.

2 Constructing a Linear Block Code for Maximum Error-Correcting Capability

An error-correcting code can be constructed as follows. We begin with the specification of the data word size *k* and the error correction bound *t*. We find the code word length *n* from the Plotkin bound (use for low-rate codes) to find *n* from *k*,

$$d_{\min} \le \frac{n\cdot 2^{k-1}}{2^k - 1} \approx \frac{n}{2} + n\cdot 2^{-k-1} \text{ for large } k \tag{2.1}$$

or the Hamming bound (use for high-rate codes) to find *(n-k)* from *t*,

$$2^{n-k} \ge \sum_{j=0}^{t}\binom{n}{j} = 2^n \cdot I_{.5}(n-1, t+1) \tag{2.2}$$

where $I_x(a,b)$ is the incomplete beta function. Any code must meet both bounds; the Plotkin bound is more stringent for low-rate codes and the Hamming bound is more stringent for high-rate codes. The probability density function utility on the EE521 web site provides a way for you to compute the incomplete beta function. Given the code word length *n* we have the number of columns in the parity array *P*, *(n-k)*, we need to select a set of basis vectors for a codeword set that has a minimum Hamming weight of d_{min}. Note that we are selecting the rows of the parity array *P*. The generator matrix consists of rows that are the basis set of codeword vectors, so we can make a good start by ensuring that the number of ones in each row equals or exceeds the required d_{min}.

3 Definition of Low Density Parity Check (LDPC) Codes

LDPC codes are known to enable achieving error-free data rates that are very close to the Shannon limit, while retaining the simplicity of implementation of the linear block codes.

The LDPC codes are defined in terms of the *k* by *n* parity check matrix *H*. If

- The number of ones in each row is constant, and is a small fraction of *n*.
- The number of ones in each column is constant, and is a small fraction of *k*.
- The number of ones in common between two columns does not exceed one.

Then, the codewords orthogonal to *H* are a LDPC. These are "regular" LDPC codes; irregular LDPC codes are sometimes used that have weaker conditions. Note that "a small fraction" is not precise, but the parity check matrix must be defined to be sparse.

Note that the parity check matrix is not forced to contain I_{n-k} in this definition. In fact, requiring that the number of ones in each row be constant makes this impossible for d_{min} greater than one. However, the code is defined by its codeword set, which means that the generator and parity check matrices for a given code is not unique. Given a parity check matrix from a given design, a new parity check matrix can be constructed by replacing rows with sums of three rows of the original parity check matrix. In this way, a LDPC code can be constructed using a generator matrix that allows a systematic code while using a parity check matrix that qualifies it as a LDPC code.

4 Example of a (8,2) Linear Block Code

This example is impractical because its code rate is ¼, which is far smaller than codes such as the (6,3) code we used as an example and other codes we will examine later. The example was selected by Sklar for an example for its simplicity and tutorial value.

4.1 Specifying the Code and Selecting the Codeword Length

We will begin by the simplest possible non-trivial code, a two-bit data word linear block code. We require that we correct both bits, *t=2*, so that we require a d_{min} of 5. We use the Plotkin bound to find the codeword length because this is a low-rate code. The Plotkin bound,

$$d_{\min} \le \frac{n \cdot 2^{k-1}}{2^k - 1} \tag{4.1}$$

or,

$$n \ge d_{\min} \cdot 2 \cdot \left(1 - 2^{-k}\right) = 5 \cdot 2 \cdot \left(1 - \frac{1}{4}\right) = \frac{5 \cdot 2 \cdot 3}{4} = 7.5 \tag{4.2}$$

so that the codeword length *n* must be at least 8. Thus we have a specification for an (8,2) code with a d_{min} of 5.

4.2 Designing the (8,2) Code

Since we have an *(8,2)* code, we know that

- The number of codewords is 2^k=4.
- The zero vector must be one of the codewords
- The codewords will be 8 bits, 5 of which are nonzero, except the zero vector.
- We must have closure, which means that the binary sum of two nonzero vectors must be equal to the third nonzero vector.
- We will end one of them with 01 and another with 10 so that we have a systematic code.

Because we have only three nonzero codewords in this very simple code, we can use Excel to design the code by manipulating two codewords and observing the properties of the third codeword. We construct a table of codeword bits, the first being the zero vector, the third being the binary sum of the second and third, which will become the generator matrix. The last two bits of the second and third are set by virtue of our requirement for a systematic code. We set the bits of the first and second so that the Hamming weight of each of the three nonzero vectors is at least 5. One such code is given below. Note that this is not the code used as an example in Sklar.

Table 1 Codewords for the (8,2) Code

Message	Codeword								d
00	0	0	0	0	0	0	0	0	0
01	0	1	0	1	1	1	0	1	5
10	1	0	1	0	1	1	1	0	5
11	1	1	1	1	0	0	1	1	6

We now have the generator matrix G,

$$G = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{4.3}$$

and the parity check matrix *H*, which we give here as its transpose:

$$H^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{4.4}$$

Note that this is *not* a LDPC because the number of ones in the rows and columns of the parity check matrix are not constant.

4.3 Error Detection and Correction Trades

We have designed this code to detect two errors, but in implementation of the decoder we have the option of designing it to correct α bits and detect β bits, where

$$\alpha + \beta \le d_{\min} - 1 \tag{4.5}$$

(Sklar's Equation (6.49) page 346) and the correction capability is bounded by

$$\beta \le t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \tag{4.6}$$

(Sklar's Equation (6.44) page 345). Of course, since we have just two message bits, detection of three or more errors is helpful only in estimating the achieved BER when it is too high for this code to be used for communication, but this can be a legitimate function during initial handshake, as part of the process of characterizing a channel for the sequence of decisions that are made in the handshake process.

For correction of single bit errors, we can use a syndrome table of only eight six-bit entries. For correction of two bit errors, we need a syndrome table of

$$\frac{n \cdot (n+1)}{2} = \frac{8 \cdot 9}{2} = 36 \tag{4.7}$$

six-bit entries (why?). There are 64 possible bit patterns for a six-bit syndrome, and the others, exclusive of zero, represent bit errors that are detected but not corrected.

4.4 Working With the (8,2) Code

Refer to the Excel spreadsheet. The code can be implemented by checking the syndrome. If it is non-zero, right-multiply the matrix of syndromes in the spreadsheet with the syndrome and search for a zero. If one is found, add the corresponding syndrome to the received codeword. If a zero is not found in the result, then an uncorrectable error is flagged.

4.5 The Standard Array

The *standard array* is defined from an *(n,k)* code an 2^{n-k} by 2^k array of codewords, valid and invalid. The top row is all of the valid codewords, beginning with the zero vector. The first column is the zero vector followed by all of the correctable error patterns; the first column is called the *coset leader*. Each remaining entry of the standard array is the binary sum of the corresponding valid codeword in the first row and the correctable error pattern in the first column. Each row is called a *coset*. Thus the standard array consists of all the valid codewords and correctable error patterns as the first row and column, and all codewords that allow correction as the other entries. All 2^n possible combinations of the n-tuple codeword are present in the standard array with no duplications. The standard array is discussed in Sklar section 6.4.8, pages 336-340.

4.6 Perfect Codes

A *perfect code* is an *(n,k)* code that correct up to *t* errors in a codeword, and the coset leader, the first column in the standard array, has all the error combinations of *t* or fewer errors, and no more, below the top row.

5 Finite Fields and m-Sequences

This topic is treated in Sklar Chapter 8. I provide it here because it provides a simple basis for moving forward with the cyclic block codes and convolutional codes and demonstrates many principles that we will use in codes of several types.

5.1 Definition of an m-Sequence

A sequence of bits can be produced by a shift register containing memory for *k* bits and feedback to perform binary addition to various places in the shift register. There are many ways to do this, but it is apparent that the shift register can be in any one of 2^k staetes, one of which is all zeros. A feedback scheme that produces all 2^k-1 states produces a maximal length sequence, or m-sequence.

5.2 Finite Fields of Order 2^k-1 Produce m-Sequences

There is a simple theory that produces simple results that helps us understand m-sequences and their properties and use. This is Galois finite field theory, and the specific part of this theory that we are interested in for error correcting codes is the theory of finite fields of 2^k elements. These fields are denoted as $GF(2^k)$, or $GF(q)$ where $q=2^k$.

Finite fields of order 2^k have elements that are vectors of *k* bits (mathematicians and others have other representations that they find convenient, but we will use this here). These are the familiar vectors that we use to organize signals for linear block codes, basis vectors, and codewords for linear block codes. Here, to help us visualize finite field theory, we will denote them as polynomials, with the bits being the coefficients of the polynomials. Thus the *k* bits are represented as polynomials of order *k-1* with coefficients that are 1 and 0. This representation of binary signals as polynomials is one that you will see in many areas of advanced signal processing theory.

Arithmetic using these vectors is done using the rules of polynomial arithmetic, except that we use binary arithmetic like we have been doing with linear block codes instead of ordinary arithmetic.

5.3 Construction of GF(q)

The field is constructed by defining arithmetic modulo a *generating polynomial*, sometimes called a *primitive polynomial* or an *irreducible polynomial*. This is necessary to define multiplication so that the result is a polynomial is of order *k-1* and thus is part of the set. We start with an element, *x*, and all the elements of the set are produced by successive powers of *x*. Multiplication by *x* is analogous to a left-shift by one bit in the shift register. The generating polynomial is of order *k* so that any term of order *k* or above in a polynomial product is mapped to lower order polynomials. The generating polynomial must have a nonzero coefficient for x^0 because if it did not, that bit would become zero after the first multiplication by *x* and would never be set again, so that we could not obtain all 2^k states. At least one other bit must be set in the generator polynomial because if only the x^0 bit were set, we would have only *k* states, not 2^k. Other than that, there are no rules that assure us that we can simply write down a generator polynomial; a trial must be made to ensure that powers of *x* do indeed repeat only after

2^k-1 iterations. Sklar has a table of some generator polynomials for *k* from 3 through 24 as Table 8.1 page 448. Interestingly he has none for *k*=2; we may add $1+x+x^2$ as a primitive polynomial for that case. For some values of *k* there is more than one generator polynomial; Sklar lists only one. This is sufficient because Galois finite field theory tells us that any one of them will do because there is a one-to-one correspondence between the elements of any one Galois field of a given order and any other of the same order, in which all the arithmetic operations track.

Sklar shows polynomial arithmetic modulo a generator polynomial by using polynomial division. A simpler method is to note that the generator polynomial provides us with the identity

$$x^k = gr(x) = p(x) - x^k = \langle \text{polynomial of order } k-1 \text{ or less} \rangle \tag{5.1}$$

and, for binary arithmetic, adding one is the same as subtracting one. Also, we note that the fundamental theorem of division,

$$\frac{p(x)}{g(x)} = q(x) + \frac{r(x)}{g(x)} \tag{5.2}$$

sometimes stated as

$$p(x) = g(x) \cdot q(x) + r(x) \tag{5.3}$$

or

$$p(x) - g(x) \cdot q(x) = r(x) \tag{5.4}$$

where *q(x)* is the quotient and *r(x)* is the remainder polynomial of degree at least one less than *g(x)*. We can apply the division theorem to the product of the generator polynomial *g(x)* and any other polynomial *h(x)*,

$$p(x) - g(x) \cdot h(x) \cdot q_{gh}(x) = r_{gh}(x) \tag{5.5}$$

We note that the new remainder $r_{gh}(x)$, is still the original polynomial minus a multiple of *g(x)*. We can find the remainder of this polynomial when divided by *g(x)*,

$$r_{gh}(x) - g(x) \cdot q_{ghg}(x) = s(x) \tag{5.6}$$

and show that the remainder *s(x)* is the desired result *r(x)* by substituting (5.5) into (5.6),

$$\begin{aligned} & p(x) - g(x) \cdot h(x) \cdot q_{gh}(x) - g(x) \cdot q_{ghg}(x) \\ &= p(x) - g(x) \cdot \left(h(x) \cdot q_{gh}(x) + q_{ghg}(x) \right) \\ &= p(x) - g(x) \cdot q(x) \\ &= r(x) \end{aligned} \tag{5.7}$$

where we know that

$$h(x) \cdot q_{gh}(x) + q_{ghg}(x) = q(x) \tag{5.8}$$

because the remainder of *(p(x)-r(x))* where *r(x)* is of order *k*-1 or less, divided by *g(x)*, must be zero. We can define *h(x)* as a power of *x* and use

$$x^{k+m} = x^m \cdot gr\left(x\right) \tag{5.9}$$

to reduce powers of *x* higher than *k* that occur with multiplication of polynomials. This makes the polynomial remainder computation very simple and quick. The quotient is extracted by setting the coefficient for x^m equal to one when (5.9) is used to reduce the product. In summary, division in binary arithmetic is rcursive,

$$\begin{aligned} &\left.\begin{aligned} p_0\left(x\right) &= p\left(x\right) \\ q_0\left(x\right) &= 0 \end{aligned}\right\} \text{Initialization} \\ &\left.\begin{aligned} p_{i+1}\left(x\right) &= p_i\left(x\right) + x^p \cdot g\left(x\right) \\ q_{i+1}\left(x\right) &= q_i\left(x\right) + x^p \end{aligned}\right\} \text{Recursion} \end{aligned} \tag{5.10}$$

where the power of x *p* is selected to annihilate the highest power of $p_i(x)$ and thus reduce its order, and the process stops when the order of $p_{i+1}(x)$ is less than that of *g(x)*. The final $p_{i+1}(x)$ is the remainder. See the Excel example for a way to do this for general numerators and denominators. The Excel example is amenable to implementation in Matlab, other HLLs, or FPGAs. Note that this method is *not* presented in Sklar.

In mathematics, a field is defined as a set of values that have these properties:

- Addition and multiplication are defined.
- Addition and multiplication are commutative and associative.
- There is a zero; multiplication by zero yields zero, and addition of a value to zero produces a sum equal to the value, unchanged – an additive identity.
- There is a one; multiplication of a value by one produces the value, unchanged – a multiplicative identity.
- There is a negative value for every value; addition of a value to its negative produces zero.
- There is a reciprocal for every value except zero; multiplication of a value by its reciprocal produces one.

The last two mean that subtraction and division are also defined, and that they are commutative and associative.

5.4 Division in GF(q)

The collection of polynomials of order *k-1* and less that we have constructed has most of the properties of a field because of the way that we have defined it; to have a field, we need only note that the polynomial with all zero coefficients meets the requirements that there be a zero, the polynomial that has only a one meets the requirements that there be a unity element, and find a reciprocal for the nonzero elements. We can find a reciprocal by noting that, for any element α of *GF(q),*

$$\alpha^q = \alpha \tag{5.11}$$

because sequences of powers of α must either step through all possible *q-1* nonzero values, or through a cycle that is a number that divides *q-1*, then the cycle must repeat. The number of values of the set that are produced by a sequence of powers is called the *order* of the element. The order of any element divides *q-1*. When the order is the full set of *q-1* elements, we call that element a *primitive element* or *primitive root* of *GF(q)*. We have defined our construction so that the polynomial *x* is a primitive element. Values of *x* to a power that does not divide *q-1* are also primitive elements (why?).

From (5.11) we see that, for any element except the zero element,

$$\alpha^{q-1} = 1 \tag{5.12}$$

for any element of *GF(q)*. Thus, we can define the reciprocal of any element except zero as

$$\alpha^{-1} = \alpha^{q-2} \tag{5.13}$$

We will find that using feedback registers in generating convolutional codes produces overlapping code vectors that are members of *GF(q)* where $q=2^k$. We will use division to recover the message bits.

6 Cyclic Codes

Cyclic codes are linear block codes formulated so that the codeword set is closed on end-around shifting. The theory of cyclic codes is based on representation of the message, codeword, and generator as polynomials. We begin with the basic formulation and then address systematic cyclic codes.

6.1 Basic Cyclic Code Theory

We build an *(n,k)* cyclic code by defining the message polynomial,

$$m(x) = m_0 + m_1 \cdot x + \ldots + m_{k-1} \cdot x^{k-1} \tag{6.1}$$

which is of order *k*-1 or less, and a generator polynomial of order *(n-k)*

$$g(x) = g_0 + g_x \cdot x + \ldots + g_{n-k} \cdot x^{n-k} \tag{6.2}$$

where g_0 and g_{n-k} must be one. The cyclic property is defined by taking codeword polynomials modulo x^n-1:

$$x^i \cdot u(x) = q(x) \cdot (x^n + 1) + u^{(i)}(x) \tag{6.3}$$

The modulo operation moves the coefficient of x^n end-around to the x^0 coefficient for each multiplication by *x*, so that

$$u^{(i)}(x) = u_{n-i} + u_{n-i+1} \cdot x + \ldots + u_{n-1} \cdot x^{i-1} + u_0 \cdot x^i + \ldots + u_{n-i-1} \cdot x^{n-1} \tag{6.4}$$

The generator polynomial is chosen as an irreducible factor of x^n+1; finding the generator polynomial is the most difficult part of defining a code, but irreducible polynomials are available in tables for most orders of interest.

The codeword polynomial is generated by a polynomial multiplication,

$$u(x) = m(x) \cdot g(x) \tag{6.5}$$

and is of order *n-1* or less, e.g. the sum of the orders of *m(x)* and *g(x)*. Thus the codeword is formed without a polynomial modulo operation.

A matrix form of this operation is

$$\underline{u} = \underline{m} \cdot G \tag{6.6}$$

where the *k* by *n* generator matrix *G* is

$$G = \begin{bmatrix} g_0 & g_1 & g_2 & \cdots & g_{n-k} & 0 & 0 & \cdots & & & \cdots & 0 \\ 0 & g_0 & g_1 & \cdots & g_{n-k-1} & g_{n-k} & 0 & \cdots & & & & \vdots \\ 0 & 0 & g_0 & \cdots & g_{n-k-2} & g_{n-k-1} & g_{n-k} & \cdots & & & & \\ \vdots & \vdots & \vdots & \ddots & & & & & & & & \\ & & & & \ddots & \ddots & & & \ddots & & & \\ & & & & \cdots & g_0 & g_1 & g_2 & \cdots & g_{n-k} & 0 & 0 \\ \vdots & & & & \cdots & 0 & g_0 & g_1 & \cdots & g_{n-k-1} & g_{n-k} & 0 \\ 0 & \cdots & & & \cdots & 0 & 0 & g_0 & \cdots & g_{n-k-2} & g_{n-k-1} & g_{n-k} \end{bmatrix} \tag{6.7}$$

Decoding is done by dividing the received codeword *z(x)* by *g(x)*,

$$z(x) = m'(x) \cdot g(x) + s(x) \tag{6.8}$$

The division process is recursive as described in (5.10) above,

$$\begin{aligned} &z_0(x) = z(x);\ m'_0(x) = 0 \\ &z_{i+1}(x) = z_i(x) + x^p \cdot g(x);\ m'_{i+1}(x) = m'_{i+1}(x) + x^p \end{aligned} \tag{6.9}$$

where *p* is the order of $z_i(x)$, and is less amenable to representation as a matrix operation.

The remainder is the syndrome *s(x)*, a polynomial of order *n-k-1* that has *n-k* bits.

We note that the message is the quotient and syndrome *s(x)* is

$$e(x) = s(x) + g(x) \cdot h(x) \tag{6.10}$$

so that the syndrome can be used directly to correct the lower *n-k* bits of the message vector if it is assumed that *h(x)* is zero – i.e. that the error is not in bits *n-k+1* through *n* of the codeword. Thus error detection is good but error correction capability is very limited.

6.2 Systematic Cyclic Codes

A systematic cyclic code can be build by using polynomial operations to build parity bits and appending them to the message bits in an augmented polynomial. The formulation is, of course, fundamentally different from that of the simple cyclic codes. We form *n-k*

parity bits as the coefficient of a parity polynomial formed as the remainder of the quotient of x^{n-k} times the message polynomial and *g(x)*:

$$\frac{x^{n-k}\cdot m(x)}{g(x)} = q(x) + \frac{p(x)}{g(x)} \tag{6.11}$$

or,

$$x^{n-k}\cdot m(x) = g(x)\cdot q(x) + p(x) \tag{6.12}$$

The lower *n-k* bits of the codeword are the coefficients of the parity matrix and the upper *k* bits are the message bits:

$$u_S(x) = p(x) + x^{n-k}\cdot m(x) = g(x)\cdot q(x) \tag{6.13}$$

Decoding is done by using the remainder of the received codeword $z_S(x)$

$$z_S(x) = u_S(x) + e_S(x) \tag{6.14}$$

divided by *g(x)* as before,

$$\frac{z_S(x)}{g(x)} = q(x) + \frac{e_S(x)}{g(x)} \tag{6.15}$$

The division can be accomplished by the simple recursion shown in (5.10) above. Again, the lower *n-k* bits of the codeword can be corrected by adding $e_S(x)$ to it. Since this is only the parity bits, message correction cannot be accomplished.

7 Well-Known Block Codes

7.1 Hamming Codes

Hamming codes are *(n,k)* block codes where

$$\begin{aligned} n &= 2^m - 1 \\ k &= 2^m - 1 - m \end{aligned} \tag{6.16}$$

for integer values of *m* of 2 and greater. They all have a d_{min} of 3 and are perfect codes (see 4.6 above). They can be formulated so that the syndrome is a binary pointer to the error pattern.

Although the Hamming codes offer simplicity of implementation and error correction, they are limited to correction of one bit error and have a code rate below ½ which limits their application because of the availability of better-performing codes.

7.2 Extended Golay Code

There exists a *(23,12)* block code with a d_{min} of 7 called the *Golay code*. Adding an overall parity bit to make it a *(24,12)* code increases d_{min} from 7 to 8 and produces a code rate of exactly ½, which simplifies implementation in modern communications systems. Extended Golay codes are usually implemented to correct three bit errors.

7.3 The BCH Codes

Contributions by three people named Bose, Chadhuri and Hocquenghem in extending the Hamming codes are referred to as BCH codes, after their initials. BCH codes are cyclic codes defined for a wide variety of values of *n*, *k* and *t*. Table 6.4 page 372 of Slkar lists polynomial coefficients of *g(x)* in octal format for BCH codes up to an *n* of 255.

BCH codes outperform all other block codes of the same block length and code rate at block lengths in the low hundreds of bits.

8 Assignment

8.1 Reading

Sklar, Chapter 6, Sections 6.1-6.6, pages 356-374.

8.2 Homework

Take-home quiz for 75% of your grade for Quiz 2. The SystemView problem from last time is 25% of your grade.

8.3 SystemView

Class discussion. We will discuss the take-home SystemView problem for Quiz 2.