E
Error Correcting Codes

The problem of adding reliability to data has already been mentioned in Section 2.12. This appendix discusses general methods for detecting and correcting errors. Reliability is, in a sense, the opposite of data compression, since it is achieved by *increasing data redundancy.* Nevertheless, many practical situations call for reliable data, so a good data compression program should be able to use codes for increased reliability, if necessary.

Every time information is transmitted, on any channel, it may get corrupted by noise. In fact, even when information is stored in a storage device, it may become bad, because no hardware is absolutely reliable. This also applies to non-computer information. Speech sent on the air gets garbled by noise, wind, high temperature, etc. Speech, in fact, is a good starting point for understanding the principles of error-detecting and -correcting codes. Imagine a noisy cocktail party where everybody talks simultaneously, on top of blaring music. We know that even in such a situation it is possible to carry on a conversation, except that more attention than usual is needed.

E.1 First Principles

What makes our language so robust, so immune to errors? There are two properties, *redundancy* and *context.*

- Our language is redundant because only a very small fraction of all possible words are valid. A huge number of words can be constructed with the 26 letters of English. Just the number of seven-letter words, e.g., is $26^7 \approx 8.031$ billion. Yet only about 50,000 words are commonly used, and even the Oxford Dictionary lists "only" about 500,000 words. When we hear a garbled word our brain searches through many similar words, for the "closest" valid word. Computers are very good at such searches, which is why redundancy is the basis for error-detecting and error-correcting codes.

■ Our brains work by associations. This is why we humans excel in using the context of a message to repair errors in the message. In receiving a sentence with a garbled word or a word that doesn't belong, such as "pass the thustard please," we first use our memory to find words that are associated with "thustard." Then we use our accumulated life experience to select, among many possible candidates, the word that best fits in the present context. If we are on the freeway, we pass the bastard in front of us; if we are at dinner, we pass the mustard (or custard). Another example is the (corrupted) written sentence

if u cn rd ths u cn bcm a c prgmr!

which we can easily understand. Computers don't have much life experience and are notoriously bad at such tasks, which is why context is not used in computer codes. In extreme cases, where much of the sentence is bad, even we may not be able to correct it, and we may ask for a retransmission "say it again, Sam."

The idea of using redundancy to add reliability to information is due to Claude Shannon, the founder of information theory. It is not a trivial idea, since we are conditioned against it. Most of the time, we try to *eliminate* redundancy in computer data, in order to save space. In fact, all the data-compression methods discussed here do just that.

Figure E.1 shows the stages that a piece of computer data may go through when it is created, stored, transmitted, received, and used at the receiving end.

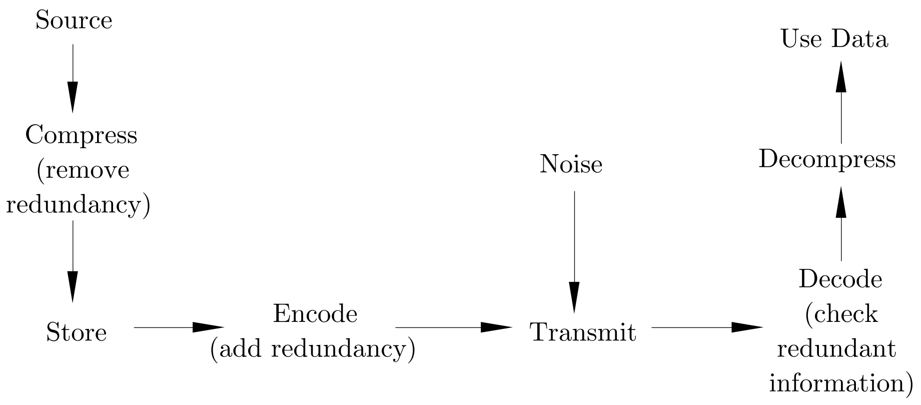

Figure E.1: Manipulating Information.

We discuss two approaches to reliable codes. The first approach is to *duplicate* the code, which leads to the idea of *voting codes*; the second one uses *check bits* and is based on the concept of *Hamming distance*.

E.2 Voting Codes

The first idea that usually occurs, when one thinks about redundancy, is to duplicate the message and send two copies. Thus if the code 110 has to be sent, one can send 110|110. A little thinking shows that this may, perhaps, be a good idea for error detection, but not for error correction. If the receiver receives two different copies, it cannot tell which one is good. What about triplicating the message? We can send 110|110|110 and tell the receiver to decide which of the three copies is good by comparing them. If all three are identical, the receiver assumes that they are correct. Moreover, if only two are identical and the third one different, the receiver assumes that the two identical copies are correct. This is the principle of *voting codes.* If all three copies are different, the receiver at least knows that an error has occurred, i.e., it can detect an error even though it cannot correct it.

To keep the analysis simple, let's limit ourselves to just 1-bit errors. When the three copies are received, the following cases are possible:

1. All three are identical. There are two subcases:
   1a: All three are good. This is the normal case.
   1b: All three have been corrupted in the same way. This is a rare case.
2. Two copies are identical, and the third one is different. Again, there are two subcases:
   2a: The two identical ones are good. This is the normal case.
   2b: They are bad. This is, we hope, a rare case.
3. All three copies are different.

Using the principle of voting, we assume that the three identical copies in case 1 are good. In case 1a our assumption is correct, and in case 1b, it isn't. Similarly, in case 2a the receiver makes the right decision, and in case 2b, the wrong one. In case 3, the receiver cannot correct the error, but at least it can detect one, so it does not make a wrong decision.

The only cases where the receiver makes the wrong decision (where the principle of voting does not work) are therefore 1b and 2b. A little thinking shows that the probability of case 1b is much smaller than that of 2b. We will, therefore, try to estimate the probability of case 2b and, if it is small enough, there would be no need to worry about case 1b.

It is hard to calculate the probability of two copies being garbled *in the same way.* We will, therefore, calculate the probability that one bit gets changed in two of the three copies (either in the same or in different bit positions). If this probability is small enough, there is no need to worry about case 2b, since its probability is even lower.

We denote by p the probability that one bit will get corrupted in our transmissions. The probability that one bit will go bad in two of the three copies is $\binom{3}{2}p^2 = 3p^2$. It is not simply p^2, since it is possible to select two objects out of three in $\binom{3}{2}$ = three ways.

[The notation $\binom{m}{n}$ is pronounced "m over n" and is defined as

$$\frac{m!}{n!(m-n)!}.$$

It is the number of ways n objects can be selected out of a set of m objects.]

Let's assume that $p = 10^{-6}$ (on average, one error in a million bits transmitted) and we want to send 10^8 bits. Without duplication we can expect $10^8 \times 10^{-6} = 100$ errors, an unacceptably high rate. With three copies sent, we have to send a total of 3×10^8 bits, and the probability that two out of the three copies will go wrong is 3×10^{-12}. The expected number of errors is thus $(3 \times 10^8) \times (3 \times 10^{-12}) = 9 \times 10^{-4} = 0.0009$ errors, a comfortably small number.

If higher reliability is needed, more copies can be sent. A code where each symbol is duplicated and sent nine times is extremely reliable. Voting codes are thus simple, reliable, and have only one disadvantage, they are too long. In practical situations, sending nine, or even three, copies of each message may be prohibitively expensive. This is why much research has been done in the field of coding in the last 40 years and, today, many sophisticated codes are known that are more reliable and shorter than voting codes.

E.3 Check Bits

In practice, error-detection and correction is usually done by means of *check bits*, which are added to the original *information bits* of each word of the message. In general, k check bits are appended to the original m information bits, to produce a *codeword* of $n = m + k$ bits. Such a code is referred to as an (n, m) code. The codeword is then transmitted to the receiver. Only certain combinations of the information bits and check bits are valid, in analogy to a natural language. The receiver knows what the valid codewords are. If a non-valid codeword is received, the receiver considers it an error. In Section E.7 we show that, by adding more check bits, the receiver can also correct certain errors, not just detect them. The principle of error correction is that, on receiving a bad codeword, the receiver selects the valid codeword that is the "closest" to it.

Example: A set of 128 symbols needs to be coded. This implies $m = 7$. If we select $k = 4$, we end up with 128 valid codewords, each 11 bits long. This is therefore an $(11, 7)$ code. The valid codewords are selected from a total of $2^{11} = 2,048$ possible codewords, so there remain $2,048 - 128 = 1,920$ non-valid codewords. The big difference between the number of valid and non-valid codewords means that if a codeword gets corrupted, chances are it will change to a non-valid one.

It may, of course, happen that a valid codeword gets changed, during transmission, to another valid codeword. Our codes are thus not completely reliable, but can be made more and more reliable by adding more check bits and by selecting the valid codewords carefully. One of the basic theorems of information theory says that codes can be made as reliable as desired by adding check bits, as long as n (the size of a codeword) does not exceed the channel's capacity.

It is important to understand the meaning of the word "error" in data transmission. When a codeword is received, the receiver always receives n bits, but some of them may be bad. A bad bit does not disappear, nor does it change into something other than a bit. A bad bit simply changes its value, either from 0 to 1, or from 1 to 0. This makes it relatively easy to correct the bit. The code should tell the receiver which bits are bad, and the receiver can then easily correct those bits by inverting them.

In practice, bits may be sent on a wire as voltages. A binary 0 may, e.g., be represented by any voltage in the range 3–25 volts. A binary 1 may similarly be represented by the voltage range of −25v to −3v. Such voltages tend to drop over long lines, and have to be amplified periodically. In the telephone network there is an amplifier (a *repeater*) every 20 miles or so. It looks at every bit received, decides whether it is a 0 or a 1 by measuring the voltage, and sends it to the next repeater as a clean, fresh pulse. If the voltage has deteriorated enough in passage, the repeater may make a wrong decision when sensing it, which introduces an error into the transmission. At present, typical transmission lines have error rates of about one in a billion, but under extreme conditions—such as in a lightning storm, or when the electric power suddenly fluctuates—the error rate may suddenly increase, creating a burst of errors.

E.4 Parity Bits

A parity bit can be added to a group of m information bits to complete the total number of 1 bits to an odd number. Thus the (odd) parity of the group 10110 is 0, since the original group plus the parity bit would have an odd number (3) of ones. Even parity can also be used, and the only difference between odd and even parity is that, in the case of even parity, a group of all zeros is valid, whereas, with odd parity, any group of bits with a parity bit added cannot be all zeros.

Parity bits can be used to design simple, but not very efficient, error-correcting codes. To correct 1-bit errors, the message can be organized as a *rectangle* of dimensions $(r-1)\times(s-1)$. A parity bit is added to each row of $s-1$ bits, and to each column of $r-1$ bits. The total size of the message (Table E.2a) becomes $s\times r$.

0	1	0	0	**1**
1	0	1	0	**0**
0	1	1	1	**1**
0	0	0	0	**0**
1	1	0	1	**1**
0	**1**	**0**	**0**	**1**

(a)

0	1	0	0	**1**
1	0	1	**0**	
0	1	**0**		
0	**0**			
1				

(b)

Table E.2: Parity Bits.

If only one bit gets bad, a check of all $s-1+r-1$ parity bits will discover it, since only one of the $s-1$ parities and only one of the $r-1$ ones will be bad.

The overhead of a code is defined as the number of parity bits divided by the number of information bits. The overhead of the rectangular code is, therefore,

$$\frac{(s-1+r-1)}{(s-1)(r-1)}\approx\frac{s+r}{s\times r-(s+r)}.$$

A similar, slightly more efficient, code is a triangular configuration, where the information bits are arranged in a triangle, with the parity bits at the diagonal

(Table E.2b). Each parity bit is the parity of all the bits in its row *and* column. If the top row contains r information bits, the entire triangle has $r(r+1)/2$ information bits and r parity bits. The overhead is thus

$$\frac{r}{r(r+1)/2} = \frac{2}{r+1}.$$

It is also possible to arrange the information bits in a number of two-dimensional planes, to obtain a three-dimensional cube, three of whose six outer surfaces are made up of parity bits.

It is not obvious how to generalize these methods to more than 1-bit error correction.

Symbol	code_1	code_2	code_3	code_4	code_5	code_6	code_7
A	0000	0000	001	001001	01011	110100	110
B	1111	1111	010	010010	10010	010011	0
C	0110	0110	100	100100	01100	001101	10
D	0111	1001	111	111111	10101	101010	111
k:	2	2	1	4	3	4	

Table E.3: Code Examples With $m = 2$.

E.5 Hamming Distance and Error Detecting

Richard Hamming developed the concept of distance, in the 1950s, as a general way to use check bits for error detection and correction.

To illustrate this concept, we start with a simple example involving just four symbols A, B, C, and D. Only 2 information bits are required, but the codes of Table E.3 add some check bits, for a total of 3–6 bits per symbol. code_1 is simple. Its four codewords were selected from the 16 possible 4-bit numbers, and are not the best possible ones. When the receiver receives one of them, say, 0111, it assumes that there is no error and the symbol received is D. When a non-valid codeword is received, the receiver signals an error. Since code_1 is not the best possible, not every error is detected. Even if we limit ourselves to single-bit errors, this code is not very good. There are 16 possible single-bit errors in our 4-bit codewords and, of those, the following 4 cannot be detected: A 0110 changed during transmission to 0111, a 0111 changed to 0110, a 1111 corrupted to 0111, and a 0111 garbled to 1111. The error detection rate is thus 12 out of 16, or 75%. In comparison, code_2 does a much better job; it can detect every single-bit error.

◇ **Exercise E.1:** Prove the above statement.

We therefore say that the four codewords of code_2 are sufficiently *distant* from each other. The concept of distance of codewords is, fortunately, easy to define.

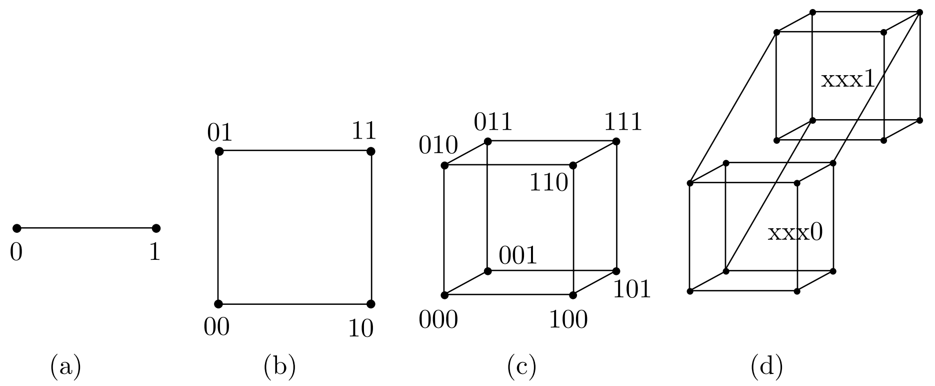

Figure E.4: Cubes of Various Dimensions and Corner Numbering.

Definitions: (1) Two codewords are a Hamming distance d apart if they differ in exactly d of their n bits and (2) a code has a Hamming distance of d if every pair of codewords in the code is, at least, a Hamming distance d apart.

These definitions have a simple geometric interpretation. Imagine a hypercube in n-dimensional space. Each of its 2^n corners can be numbered by an n-bit number (Figure E.4), such that each of the n bits corresponds to one of the n dimensions. In such a cube, points that are directly connected have a Hamming distance of 1, points with a common neighbor have a Hamming distance of 2, etc. If a code with a Hamming distance of 2 is needed, only points that are not directly connected should be selected as valid codewords.

The reason code$_2$ can detect all single-bit errors is that it has a Hamming distance of 2. The distance between valid codewords is 2, so a 1-bit error always changes a valid codeword into a non-valid one. When two bits go bad, a valid codeword is moved to another codeword at distance 2. If we want that other codeword to be non-valid, the code must have at least distance 3.

In general, a code with a Hamming distance of $d+1$ can detect all d-bit errors. code$_3$ has a Hamming distance of 2 and thus can detect all 1-bit errors even though it is short ($n=3$).

◇ **Exercise E.2:** Find the Hamming distance of code$_4$.

It is now obvious that we can increase the reliability of our transmissions to any desired level, but this feature does not come free. As always, there is a tradeoff, or a price to pay, in the form of the overhead. Our codes are much longer than m bits per symbol because of the added check bits. A measure of the price is $n/m = (m+k)/m = 1+k/m$, where the quantity k/m is called the *overhead* of the code. In the case of code$_1$ the overhead is 2 and, in the case of code$_3$, it is 3/2.

Example: A code with a single check bit, which is a parity bit (even or odd). Any single-bit error can easily be detected, since it creates a non-valid codeword. Such a code therefore has a Hamming distance of 2. code$_3$ above uses a single, odd, parity bit.

Example: A 2-bit error-detecting code for the same four symbols (see code$_4$).

It must have a Hamming distance of 4, and one way of generating it is to duplicate code_3.

E.6 Hamming Codes

The principle of error-correcting codes is to separate the codes even farther by adding more redundancy (more check bits). When an invalid codeword is received, the receiver corrects the error by selecting the valid codeword that is closest to the one received. code_5 has a Hamming distance of 3. When one of its four codewords has a single bit changed, it is 1-bit distant from the original one, but is still 2 bits distant from any of the other codewords. Thus, if there is only one error, the receiver can always correct it. The receiver does that by comparing every codeword received to the list of valid codewords. If no match is found, the receiver assumes that a 1-bit error has occurred, and it corrects the error by selecting the codeword that is closest to the one received.

In general, when d bits go wrong in a codeword C_1, it turns into an invalid codeword C_2 at a distance d from C_1. If the distance between C_2 and the other valid codewords is at least $d+1$, then C_2 is closer to C_1 than it is to any other valid codeword. This is why a code with a Hamming distance of $d+(d+1)=2d+1$ is needed to correct all d-bit errors.

How are the codewords selected? The problem is to select a good set of 2^m codewords out of the 2^n possible ones. The first approach uses brute force. It is easy to write a computer program that will examine all the possible sets of 2^m codewords, and select one that has the right distance. The problems with this approach are: (1) The time and storage required at the receiving end to verify and correct the codes received and (2) the amount of time it takes to examine all the possible sets of codewords.

1. The receiver must have a list of all the 2^n possible codewords. For each codeword it must have a flag indicating whether it is valid and, if not, which valid codeword is the closest to it. Every codeword received has to be searched and located in this list in order to verify it.

2. In the case of four symbols, only four codewords need be selected. For code_1 and code_2, they had to be selected from among 16 possible numbers, which can be done in $\binom{16}{4}=7,280$ ways. It is possible to write a simple program that will systematically examine sets of four codewords until it finds a set with the required distance. In the case of code_4, the four codewords had to selected from a set of 64 numbers, which can be done in $\binom{64}{4}=635,376$ ways. It is still possible to write a program that will systematically explore all the possible codeword selections for this case. In practical cases, however, where sets with hundreds of symbols are involved, the number of possibilities in selecting sets of codewords is too large even for the fastest computers to handle comfortably.

Clearly, a clever algorithm is needed, to select the best codewords, and to verify them on the fly, as they are being received. The transmitter should use the algorithm to generate the codewords when they have to be sent, and the receiver should use it to check them when they are received. The approach described here is due to Richard Hamming. In Hamming's codes [Hamming 86] the n bits of a codeword are indexed from 1 to n. The check bits are those with indexes that are powers of 2. Thus bits b_1, b_2, b_4, $b_8, \ldots$ are check bits, and b_3, b_5, b_6, b_7, $b_9, \ldots$ are information bits. The index of each information bit can be written as the sum of the indexes of certain check bits. Thus b_7 can be written as b_{1+2+4} and is, therefore, used in determining the values of check bits b_1, b_2, b_4. *The check bits are simply parity bits.* The value of b_2, e.g., is the parity (odd or even) of b_3, b_6, b_7, $b_{10}, \ldots$ etc., since $3 = 2 + 1$, $6 = 2 + 4$, $7 = 2 + 1 + 4$, $10 = 2 + 8, \ldots$.

Example: A 1-bit error-correcting code for the set of symbols A, B, C, and D. It must have a Hamming distance of $2d + 1 = 3$. Two information bits are needed to code the four symbols, so they must be: b_3 and b_5. The parity bits are therefore b_1, b_2, and b_4. Since $3 = 1 + 2$ and $5 = 1 + 4$, the 3 parity bits are defined as b_1 is the parity of bits b_3 and b_5, b_2 is the parity of b_3, and b_4 is the parity of b_5. This is how code$_5$ of Table E.3 was constructed.

Example: A 1-bit error-correcting code for a set of 256 symbols. It must have a Hamming distance of $2d + 1 = 3$. Eight information bits are required to code the 256 symbols, so they must be b_3, b_5, b_6, b_7, b_9, b_{10}, b_{11}, and b_{12}. The parity bits are, therefore, b_1, b_2, b_4, and b_8. The total size of the code is 12 bits. The following relations define the 4 parity bits:
$3 = 1 + 2$, $5 = 1 + 4$, $6 = 2 + 4$, $7 = 1 + 2 + 4$, $9 = 1 + 8$, $10 = 2 + 8$, $11 = 1 + 2 + 8$, and $12 = 4 + 8$.
They imply that b_1 is the parity of b_3, b_5, b_7, b_9, and b_{11}.

◇ **Exercise E.3:** What are the definitions of the other parity bits?

◇ **Exercise E.4:** Construct a 1-bit error-correcting Hamming code for 16-bit codes ($m = 16$).

A common question at this point is how the number of parity bits is determined. The answer is that it is determined implicitly. We know that m data bits are needed, and we also know that bits $b_1, b_2, b_4, b_8, \ldots$ should be the parity bits. We thus allocate the first m bits of the set $b_3, b_5, b_6, b_7, b_9, b_{10}, b_{11}, \ldots$ to the data, and this implicitly determines the number of parity bits needed.

What is the size of a general Hamming code? The case of a 1-bit error-correcting code is easy to analyze. Given a set of 2^m symbols, 2^m valid codewords are needed. We are looking for the smallest k required to construct codewords of size $m + k$ and Hamming distance 3. The 2^m valid codewords should be selected from a total of 2^n numbers (where $n = m + k$), such that each codeword consists of m information bits and k check bits.

Since we want any single-bit error in a codeword to be corrected, such an error should not take us too far from the original codeword. A single-bit error takes us to a codeword at distance 1 from the original one. As a result, all codewords at distance 1 from the original codeword should be non-valid. Each of the original 2^m codewords is n bits long and thus has n codewords at distance 1 from it. They should be declared non-valid. This means that the total number of codewords needed (valid plus non-valid) is $2^m + n2^m = (1 + n)2^m$. This number has to be selected from the 2^n available numbers, so we end up with the relation $(1 + n)2^m \le 2^n$. Since $2^n = 2^{m+k}$, we get $1 + n \le 2^k$ or $k \ge \log_2(1 + n)$. The following table illustrates the meaning of this relation for certain values of m.

n:	4	7	12	21	38	71
k:	2	3	4	5	6	7
$m = n - k$:	2	4	8	16	32	64
k/m:	1	.75	.5	.31	.19	.11

There is a geometric interpretation that provides another way of obtaining the same result. We imagine 2^m spheres of radius one tightly packed in our n-dimensional cube. Each sphere is centered on one of the corners and encompasses all its immediate corner neighbors. The *volume* of a sphere is defined as the number of corners it includes, which is $1+n$. The spheres are tightly packed, but they don't overlap, so their total volume is $(1+n)2^m$, and this should not exceed the total volume of the cube, which is 2^n.

The case of a 2-bit error-correcting code is similarly analysed. Each valid codeword should define a set that includes itself, the n codewords at distance 1 from it, and the set of $\binom{n}{2}$ codewords at distance 2 from it, a total of $\binom{n}{0}+\binom{n}{1}+\binom{n}{2} = 1+n+n(n-1)/2$. Those sets should be non-overlapping, which implies the relation

$$\bigl(1+n+n(n-1)/2\bigr)2^m \le 2^n \Rightarrow 1+n+n(n-1)/2 \le 2^k \Rightarrow k \ge \log_2\bigl(1+n+n(n-1)/2\bigr).$$

In the geometric interpretation we again imagine 2^m spheres of radius 2 each. Each sphere centered around a corner and containing the corner, its n immediate neighbors, and its $\binom{n}{2}$ second place neighbors (corners differing from the center corner by 2 bits).

E.7 The SEC-DED Code

However, even though we can estimate the length of a 2-bit error-correcting Hamming code, we don't know how to construct it! The best that can be done today with Hamming codes is single-error-correction combined with double-error-detection. An example of such a SEC-DED code is code$_6$. It was created by simply adding a parity bit to code$_5$.

◇ **Exercise E.5:** Table E.3 contains one more code, code$_7$. What is it?

The receiver checks the SEC-DED code in two steps. In step 1, the single parity bit is checked. If it is bad, the receiver assumes that a 1-bit error occurred, and it uses the other parity bits, in step 2, to correct the error. It may happen, of course, that three or even five bits are bad, but the simple SEC-DED code cannot detect such errors.

If the single parity is good, then there are either no errors, or two bits are bad. The receiver switches to step 2, where it uses the other parity bits to distinguish between these two cases. Again, there could be four or six bad bits, but this code cannot handle these cases.

The SEC-DED code has a Hamming distance of 4. In general, a code for c-bit error correction and d-bit error detection should have a distance of $c+d+1$.

E.8 Generating Polynomials

There are many approaches to the problem of developing codes for more than 1-bit error correction. They are, however, more complicated than Hamming's method, and require a background in group theory and Galois fields. In this section we briefly sketch one such approach, using the concept of a *generating polynomial*.

We use the case $m = 4$ for illustration. Sixteen codewords are needed, which can be used to code any set of 16 symbols. We know from the discussion above that, for 1-bit error correction, 3 parity bits are needed, bringing the total size of the code to $n = 7$. Here is an example of such a code:

0000000 0001011 0010110 0011101 0100111 0101100 0110001 0111010
1000101 1001110 1010011 1011000 1100010 1101001 1110100 1111111

Note that it has the following properties:

- The sum (modulo 2) of any two codewords equals another codeword. This implies that the sum of any number of codewords is a codeword. The 16 codewords above thus form a *group* under this operation.
(Addition and subtraction modulo-2 is done by $0+0 = 1+1 = 0$, $0+1 = 1+0 = 1$, $1-0 = 0-1 = 1$. The definition of a group should be reviewed in any text on algebra.)
- Any circular shift of a codeword is another codeword. This code is thus *cyclic*.
- It has a Hamming distance of 3, as required for 1-bit error correction.

Interesting properties! The 16 codewords were selected from the 128 possible ones by means of a generator polynomial. The idea is to look at each codeword as a polynomial, where the bits are the coefficients. Here are some 7-bit codewords associated with polynomials of degree 6.

$$\begin{array}{ccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 \\
x^6 & & & +x^3 & +x^2 & +x & +1 \\[2ex]
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
 & x^5 & +x^4 & & & +x & \\[2ex]
0 & 1 & 0 & 0 & 1 & 1 & 1 \\
 & x^5 & & & +x^2 & +x & +1
\end{array}$$

The 16 codewords above have been selected by finding the degree-6 polynomials that are evenly divisible (modulo 2) by the generating polynomial $x^3 + x + 1$. For example, the third codeword '0100111' in the table corresponds to the polynomial $x^5 + x^2 + x + 1$, which is divisible by $x^3 + x + 1$ because $x^5 + x^2 + x + 1 = (x^3 + x + 1)(x^2 + 1)$.

To understand how such polynomials can be calculated, let's consider similar operations on numbers. Suppose we want to know the largest multiple of 7 that is ≤ 30. We divide 30 by 7, obtaining a remainder of 2, and then subtract the 2 from the 30, getting 28. Similarly with polynomials. Let's start with the 4 information bits 0010, and calculate the remaining 3 parity bits. We write 0010*ppp*, which gives us the polynomial x^4. We divide x^4 by the generating polynomial, obtaining a remainder of $x^2 + x$. Subtracting that remainder from x^4 gives us something that will be evenly divisible by the generating polynomial. The result of the subtraction is $x^4 + x^2 + x$, so the complete codeword is 0010110.

Any generating polynomial can get us the first two properties. To get the third property (the necessary Hamming distance), the right generating polynomial should be used, and it can be selected by examining its roots. This topic is outside the scope of this book, but it is discussed in any text on error-correcting codes. A common example of a generating polynomial is CRC (Section 3.23).

Bibliography

Hamming, Richard (1950) "Error Detecting and Error Correcting Codes," *Bell Systems Technical Journal* **29**:147–160, April.

Hamming, Richard (1986) *Coding and Information Theory*, 2nd Ed., Englewood Cliffs, NJ, Prentice-Hall.

Lin, Shu (1970) *An Introduction to Error Correcting Codes*, Englewood Cliffs, NJ, Prentice-Hall.

Errors using inadequate data are much
less than those using no data at all.

Charles Babbage (1792–1871)

Give me fruitful error any time, full of seeds, bursting with its
own corrections. You can keep your sterile truth for yourself.

Wilfredo Pareto (1848–1923)