Chapter 10

Information Protection

In many systems, there is often a desire and/or need to protect information for a variety of reasons and in a variety of ways. There are at least four distinct subareas that address different needs:

- **Error detection and correction codes** This area is also commonly called coding theory. The goal of these techniques is to add redundancy (extra bits) to protect against (usually) random errors. The data is expanded to try to get reliable communication through an unreliable (noisy) channel.

- **Cryptography** This term is often used interchangeably with the term cryptology, although sometimes these terms are used with slightly different meanings. The goal of these techniques is to encrypt data to make it unintelligible except for intended users. Using suitable keys, the intended users can decrypt the string of symbols to recover the original data. These techniques are typically used for security or authentication.

- **Spread spectrum communications** This area was originally developed for military applications to avoid jamming or interception of radio communications. If the enemy knows the frequency band which is being used for communication, then they could "tune in" and eavesdrop or transmit noise at the same frequency to jam/corrupt the transmission. Roughly, spread spectrum techniques use a broad range of frequencies in various ways to reduce eavesdropping and jamming. These techniques are now playing a major role in mobile/wireless communications.

- **Watermarking** This area has seen much recent activity as various forms of media are widely available digital form and over the internet. Watermarking is closely related to topics known as steganography and data

†Lecture Notes for ELE201 Introduction to Electrical Signals and Systems.
‡Thanks to Sean McCormick for producing the figures.

hiding. The goal is to keep data intelligible, but utilize redundancy in the data to embed additional information. For example, for an image, one might like to add either a visible or invisible watermark for ownership protection or other purposes.

In this chapter, we focus only on error detection and correction.

10.1 Expansion for Protection

As mentioned above, the goal of error detection and/or correction is to deal with unintended noise in data, which is usually assumed to be random. The key idea is to add redundancy to the data so that there is some internal consistency or structure that can be used to combat the noise.

For example, a very simple (but not very useful) idea to detect errors is to repeat the data twice. If we wish to send the message 00101, we will instead transmit 00101 00101. If the recipient gets something other than the same copy twice, they will know an error has occurred. However, the receiver will not be able to correct the error, since they will not know which copy contains the error. But, the receiver may then ask for yet another transmission!

To be able to correct a single error, we could repeat the message three times and send 00101 00101 00101. In this case, if the transmission incurs a single error anywhere in the message (e.g., becomes 00101 01101 00101), the receiver will still have two copies of the original (uncorrupted) message that match, and so will be able to correct the error. Unfortunately, these simple methods expand the data so much that they are not very practical. However, much more clever and sophisticated techniques exist to do similar things much more efficiently.

The main point is that we build resilience to noise by expanding the data. This is exactly the opposite of what we were trying to do in compression. There we were trying to squeeze out all the redundancy we could to reduce the data as much as possible. Here we are adding back some redundancy. Although at first it might seem silly, doing both can be very productive. The reason is that redundancy in the original data is either not highly structured, or the structure is not known well enough to be fully exploited. Therefore, we can reap substantial gains by removing the original redundancy and then adding some redundancy back but in a careful and highly structured manner. In this way, we can reduce the overall data size *and* increase the resilience to noise.

The techniques are so useful they are used in a huge range of applications involving data transmission and/or storage. For example, error detection and/or protection is used in modems, CD players, transmission in data networks, cell phones, and many other applications.

10.2 Noise and Level of Protection

As one might expect, the suitability of a technique depends crucially on assumptions about the noise. Three common types of noise in binary data are

Substitution:

Original message	1	5	9	B
Original binary signal	0001	0101	1001	1011
Binary signal with error	0001	0*0*01	1001	1011
Message with error	1	1	9	B

Erasure:

Original message	1	5	9	B
Original binary signal	0001	0101	1001	1011
Binary signal with erasure	0001	0_01	1001	1011
New Binary signal	0001	0011	0011	0110
Message with error	1	3	3	6

Addition:

Original message	1	5	9	B
Original binary signal	0001	0101	1001	1011
Binary signal with addition	0001	0*1*101	1001	1011
New Binary signal	0001	0110	1100	1101
Message with error	1	6	C	D

Figure 10.1: Types of Noise in Binary Data

substitution errors (in which a 0 turns into a 1 or vice versa), erasures (also called deletions, in which a bit is lost), and additions (also called insertions, in which an extra bit is inserted). Examples of these are shown in Figure 10.1. The most common model for noise is the substitution error, which is what we will primarily consider.

In addition to the type of noise, the suitability of an error protection technique depends on both the tolerable error rate in the system as well as on the distribution of the noise. In some applications we can tolerate a large amount of error. For example, in sending a fax, a small amount of noise will not cause a problem and may not even be noticed. Even if one in every ten thousand pixels gets corrupted, the result will still be useful. On the other hand, if we are transmitting an executable program file of a couple megabytes, even one error will likely render the transmission useless. Financial data or other records may also require an extremely high level of reliability.

Typically one designs a method to protect up to some number of errors, for example, single error detection, triple error detection plus double error correction, etc. When we say a code provides a certain level of protection, we mean that it can detect and/or correct *any* error of that particular type. For example, a code that provides single error correction against substitution errors can

correct any single substitution error, including errors in the check bits added to provide the protection. Some single error correcting codes may happen to be able to correct certain double errors, but unless the code can correct all double errors it will not be considered a double error correcting code.

Here we will primarily focus on single error detection and single error correction, but we briefly discuss the conceptual approach for protecting against multiple errors. We will also focus almost exclusively on the case of binary data – i.e., just 0's and 1's (except for ISBN numbers that we will describe briefly). Binary data is the most commonly encountered setting in applications, and moreover many of the ideas in the binary case can be extended to non-binary data.

Typically, the data will be divided into units, e.g., 8 bits for each gray level or symbol, or 128 bits for a set of symbols or other packet of data. Redundancy is added to each data unit to provide the desired level of protection. The level of protection chosen may depend on the probability of error for each bit, the size of the data units, and the tolerable level of corrupted units.

For example, suppose we are working with 8 bit codewords and can tolerate an average of one erroneous codeword out of every million codewords. Let p be the probability that a given bit gets corrupted. If p is small enough (say less than 10^{-7} so that only one bit in every ten million gets corrupted), then the number of corrupted codewords may be within tolerable bounds. In this case, we may not need any error protection since the probability of error itself is sufficiently small. On the other hand, if a single error per codeword occurs more often than 1 in a million, but two or more errors are very unlikely, then single error correction may be enough. Even single error detection may be enough if re-transmission of corrupted data units is possible. The main point is that the level of protection we need to design for depends greatly on the probability of an error, size of data units, and tolerable losses in the data.

Actually, in some cases knowing just the probability of error is not enough. The distribution of noise may also be important. For example, if errors are expected to be "bursty," then if an error occurs several consecutive bits may get corrupted. Refinements to the basic techniques may be necessary to deal with such cases, although we will not deal with such issues here.

10.3 Single Error Detection

As mentioned above, repeating the message (or each bit) twice provides a simple, if impractical scheme for detecting single errors. The downside is that this doubles the amount of data. Some partial consolation for this outrageous excess data is that repetition is robust to essentially any type of single error (since the two copies won't match and hence error will be detected). It can even detect many types of double (or more) errors (except with presumably small probability when both copies happen to get corrupted in the same way). Nevertheless, the price of doubling the data is too great, and if one is concerned about double errors or other types of errors other techniques may be more suitable.

Parity Checking				
Assume we have a data stream that consists of groups of three bits	001	101	111	001
We add a check bit to each group to make even parity	001 1	101 0	111 1	001 1
Errors will make recieved parity odd	+__1 _ 000 1	___ _ 101 0	_1__ _ 101 1	___ _ 001 1
Unless two errors occur in the same block, making even parity	+_11 _ 010 1	___ _ 101 0	___ _ 101 1	1_1 _ 100 1

Figure 10.2:

Parity Check Bits

A very elegant (and also simple) alternative to to detect single substitution errors is to add an extra bit, so that the number of 1's in the data unit (including the check bit) is even. In this case, the resulting message including the check bit is said to have even parity. To determine whether or not there is an error in a received data unit, the receiver simply checks the parity of the received data. If it is odd, then clearly there was an error. If it is even, no error is assumed to have occurred. Of course, in this case, if two errors occur then these will not be detected. Instead, the receiver will assume the data received is error-free. An example is shown in Figure 10.2.

Of course, we could have chosen to use odd parity, instead of even parity. That is, we could have appended a check bit to make sure the number of 1's in the data unit is odd. This will work just as well. The main requirement is that the encoder and decoder must agree on whether to use even or odd parity.

A parity check bit provides single error detection with a single additional bit. If the probability of error for each bit is quite small, then the probability of a double error will be extremely small. In this case, a parity check bit can be quite adequate, especially in applications where the data can be re-transmitted or re-read. Thus, for single error detection parity check bits are the most widely used scheme.

ISBN numbers

The performance of a protection scheme depends on type of noise encountered, and therefore the scheme should be designed taking knowledge of noise properties into account. We end this section with an example designed for commonly occurring human errors.

All books are assigned a 10-digit International Standard Book Numbers (or

ISBN). You can usually find the ISBN on the back of the book near the bottom. ISBN's give a standardized way to identify a given book. E.g., the book *Information Theory: 50 Years of Discovery* published by the IEEE Press, has the ISBN 0-7803-5363-3.

To discuss ISBN's in more detail, we will denote the 10 digits in an ISBN by $d_{10}d_9 \ldots d_2d_1$. So, for the book mentioned above, $d_{10} = 0$, $d_9 = 7$, and so on.

The digit d_{10} indicates whether the book is published by a U.S. or international publisher. The next two digits d_9d_8 identify the publisher. The digits d_7 through d_2 are a book number assigned by the publisher.

The last digit, d_1 is a check digit. It is chosen so that the quantity

$$10d_{10} + 9d_9 + \cdots + 2d_2 + d_1$$

is divisible by 11. For example, for the book mentioned above, we get

$$10 \times 0 + 9 \times 7 + \cdots + 2 \times 3 + 3 =$$

A natural question is why is the weighted sum used (instead of a check using an unweighted sum)? Using a weighted sum allows detecting any single substitution error, which could also be detected by an unweighted scheme. However, a common human error is to switch order of adjacent digits. The weighted sum scheme can detect this (for any adjacent digits), while an unweighted sum clearly cannot (since switching digits would leave the sum unchanged).

What happens if d_1 needs to be 10 in order to make to make the weighted sum divisible by 11 (for example, if $10d_{10} + \cdots + 2d_2 = 210$)? The convention is to use the symbol "X" as the last "digit" in this case.

10.4 Single Error Correction: Preliminaries

As with single error detection, simplest scheme for single error correction repetition, namely just repeat the message 3 times. In this case, to correct an error the receiver can simply take a majority vote, bit by bit. Although this triplicate code is simple and quite robust to different types of errors, it's not practical because it triples the amount of data. Much more efficient schemes exist that we will discuss in the next three sections, focusing only on substitution errors. However, first we discuss some preliminaries.

Some clarification is needed on our use of terms describing a certain level of protection. First, it's clear that by repeating the message three times, certain (in fact many) double errors can be detected (as well as corrected). But, does the repetition code allow detecting *all* double errors?

Well, in a sense one might say yes, since if there are any two errors then certainly not all three copies of the message will be the same. Therefore, the receiver will know that something is wrong. However, the problem with this is that if we insist on correcting single errors, then *certain* double errors (namely, an error in the same bit in two of the copies) will be incorrectly assumed to be *single* error, and therefore will be "fixed" incorrectly.

For example, suppose the original message is 00101 so that 00101 00101 00101 gets transmitted. If two errors occur and they happen to be the first bit in the first two copies, then the data received will be 10101 10101 00101. In this case, a receiver that is implementing single error correction will erroneously think that there is a single error in the first bit of the third copy, and will "correct" this error concluding that the original message was 10101.

Now, if we are not interested in correcting single errors, then any double errors can be detected properly. Although this might be useful in some cases, the convention is that when a code is said to detect and/or correct a certain number of errors, it is assumed that all smaller number of error will be detected *and* corrected. Of course, if a code can correct a certain number of error, then it can surely detect that same number of error. Thus, the hierarchy in terms of level of protection is (1) single error detection, (2) single error correction, (3) single error correction plus double error detection, (4) double error correction, (5) double error correction plus triple error detection, etc.

Now we introduce some notation and terminology that will let us better compare the performance of different codes. An (n, m) code will refer to a code with data units that have:

- n total bits
- m message/information bits
- $k = n - m$ check bits

The *redundancy* and *excess redundancy* of the code are defined to be

$$\text{redundancy} = \frac{n}{m} = 1 + \frac{k}{m} \tag{10.1}$$

$$\text{excess redundancy} = \frac{k}{m} \tag{10.2}$$

For a given level of protection, we would like the redundancy to be as small as possible.

As an example, consider the case of the triplicate code. With m message bits, we will have $n = 3m$ total bits and $k = 2m$ check bits. Therefore, the redundancy is 3 and the excess redundancy is 2. It so happens that for the triplicate code, the redundancy and excess redundancy do not depend on the size of the data unit (i.e., the number of message bits m or the number of total bits n). However, in general, for a given lass of codes the redundancy decreases as the size of the data unit increases.

For example, for single error detection with a parity check bit, the redundancy is $1 + 1/m$ and the excess redundancy is $1/m$. The larger the block to which we append the check bit, the smaller the redundancy. The tradeoff is that as the size of the block increases, the probability that one or more of the bits is corrupted increases. If it's too large, then an unacceptable number of blocks may have single or multiple (possibly undetected) errors.

10.5 Rectangular and Triangular Codes

Rectangular Codes

A clever yet simple method for single error correction is to arrange the block of message/information bits in a rectangular array, and then add a parity check bit to each row and each column. Figure 10.5 shows the arrangement of message and check bits with a 3x4 array of message bits.

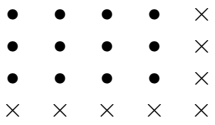

Figure 10.3:

The check bit at the end of a row is chosen to make all the one in that row (including the check bit) have the appropriate parity. Similarly for the columns. Any single error can be corrected by finding the row and column for which the parity check fails.

The check bit in the the corner is actually not necessary. If it's used, the check bit in the corner will check both the row and column of check bits. That is, the corner bit is chosen to make all the check bits (including the corner) have the proper parity. In this case, if there's an error in a check bit, then both the corresponding row or column check will fail and the corner check will fail. If we don't use the corner bit, then everything will work as before if the error is in one of the message bits. If the error is in a check bit, then only the check for that row or column will fail. In this case, the arrangement of message bits and check bits is as shown in Figure 10.4.

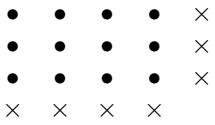

Figure 10.4:

Example 10.1 (Example of a 3x4 Rectangular Code)

Suppose the original message is 101101011000. Arranging this into a rectangular array we get

$$\begin{matrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{matrix}$$

Using even parity, the message with the check bits is

$$\begin{array}{ccccc} 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & \end{array}$$

Now suppose the receiver observes a possible corrupted bit stream. The receiver would arrange the bits in the same way, and perform the parity checks on each row and column as shown below.

$$\begin{array}{ccccccc} 1 & 0 & 1 & 1 & 1 & & \\ 0 & 1 & 1 & 1 & 0 & \text{F} & \\ 1 & 0 & 0 & 0 & 1 & & \\ 0 & 1 & 1 & 0 & & & \\ & & \text{F} & & & & \end{array}$$

Since the parity checks in row 2 and column 3 fail, this indicates that the position of the error is the bit in the 2nd row and 3rd column. The receiver can therefore correct the error and recover the original message.

■

There are several things to note about this error correction scheme:

- As before, it doesn't matter whether we use even parity or odd parity, as long as both the coder and decoder agree.
- In an algorithmic implementation, we don't actually have to arrange the bits in a rectangle. We just need to add the (redundant) check bits that perform the necessary parity checks. The data are arranged in a stream, and the coder and decoder only need to know (agree) which subsets are supposed to satisfy the parity checks.

The redundancy of a general rectangular code with an array of a_1 by a_2 message bits can computed, but for simplicity, we consider the special case of a square code. For $m = a^2$ message bits using a square code, we have $k = 2_a$ and $n = a^2 + 2_a$ (assuming we do not use the use the extra corner check bit). In this case, the redundancy is $1 + 2/a$ and the excess redundancy is $2/a$. Even for $a = 2$, this code is more efficient than the triplicate code, and the benefits clearly get much greater for larger a.

An alternate way to compare the codes is to fix the number of check bits k and see how many message bits can be accommodated. For the triplicate code, $m = k/2$, but for the square code, $m = (k/2)^2$.

Triangular Codes

A more efficient code can be obtained by arranging the message bits in triangular array, and adding another layer (diagonal) of check bits, as shown in Figure 10.5.

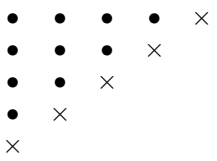

Figure 10.5:

Each check bit checks *both* the row and column to which it belongs. The check bit makes sure that the total number of ones in the row and column together, including the check bit itself, is of the right parity.

If there is a single error in a message bit, then two check bits (in both the corresponding row and column) will fail, and the receiver will know which message bit is in error. If a check bit is in error, then only that check bit will fail the parity check. In either case, a single error can be corrected. An example is shown in Figure 10.6.

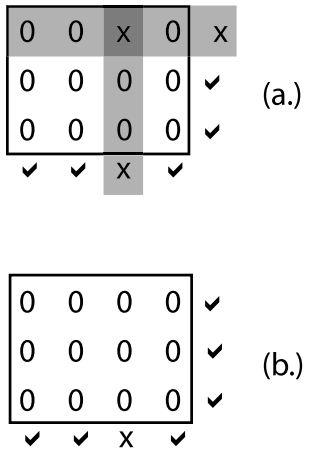

Figure 10.6: (a.) shows a data bit in error, and two check bits indicating error. (b.) shows only a check bit with an error, indicating the error is only in the check bit.

As with the rectangular codes, although the picture helps conceptually, in an implementation one does not really need to arrange the bits in a triangular array. Instead, check bits are added that simply check the appropriate subsets of bits.

To get an idea of the efficiency of the triangular code, consider a fixed number of check bits k. The number of message bits that can be accommodated is

$$m = 1 + 2 + \cdots + (k-1) = \frac{k(k-1)}{2}$$

This is larger than the number of message bits for the square code for any $k \geq 2$, which is always the case for these codes.

10.6 Higher Dimensional Codes

An idea that leads to codes even more efficient than the triangular codes is to arrange the data in a 3-D cube. A check bit would be added for each plane. That is the check bits would be added along the three edges of the coordinate axes, as depicted in Figure 10.7. Each check bit would be chosen so that the number of 1's in all bits that lie in the corresponding plane have even parity. Note that, as before, we don't need to add a check bit in the corner.

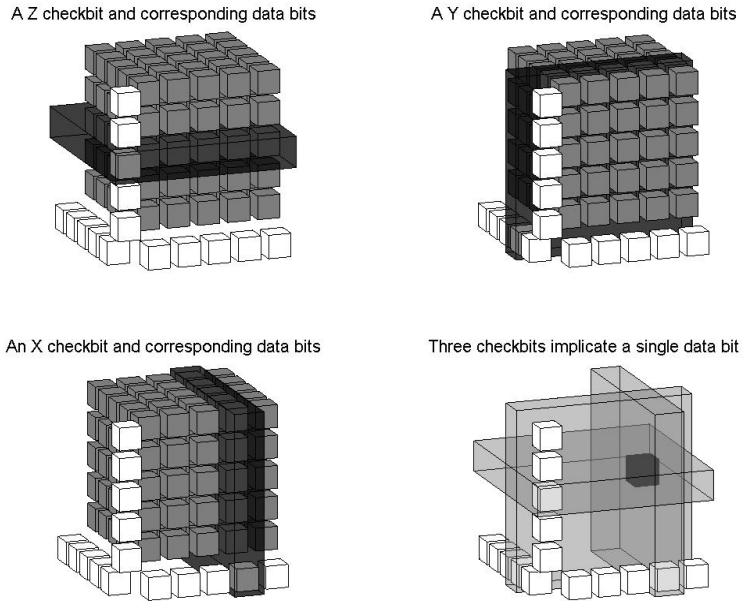

Figure 10.7: 3-D parity checking code

If the data (including the check bits) is arranged in a cube of size $s \times s \times s$, then there are $n = s^3 - 1$ total bits (the -1 is because of the missing corner). These include $k = 3(s-1) = 3s - 3$ check bits resulting in $m = s^3 - 3s + 2$ message bits. Thus, the excess redundancy is $(3s-3)/(s^3 - 3s + 2)$. For large s, this is approximately $3/s^2$, which is better than the results in 2-D.

These results suggest that yet more efficiency can be obtained if we go to even higher dimensions. Of course, as before, in an implementation we won't actually arrange the data into a hypercube. This is even difficult to depict in a figure. Nevertheless, we can add check bits that perform the necessary checks. If this is done, then in 4-D we will get an excess redundancy of approximately $4/s^3$ for large s where the bits (message and check bits) are arranged in a hypercube

with s bits along each dimension.

What's the highest dimension possible? Well, we need at least 2 bits along that dimension (otherwise we haven't actually used the dimension). So, with d dimensions, we will have $n = 2^d - 1$ total bits (again, the -1 is for the missing corner). We have only 1 check bit per dimension. This is because each edge has 2 positions, one of which is the corner. Hence, we will have $k = d$ check bits, resulting in $m = 2^d - d - 1$ message bits.

Can we do any better by using some other tricks? The answer is no, and we can actually see this with a relatively simple argument. Suppose we have k check bits and wish to correct single errors. When we receive a string, we will consider each check bit to see if the corresponding parity check passes. There are only two possibilities for each check bit — either the test passes or not. Thus, with k check bits, we can represent 2^k situations. On the other hand, the situations we must be able to represent include the case of no error, and n situations for the case that the error happens to be in each of the n bits. This gives $n + 1$ situations that we must be able to represent. Therefore, we must have $n+1 \leq 2^k$ so $n \leq 2^k - 1$. The d-dimensional case described above achieves this constraint with equality.

The discussion above was rather abstract. Although it suggests what is the best we can do for single error correction, it would be nice to have a concrete algorithm that is easy to understand. This is the subject of the next section.

10.7 Hamming Codes

We'll describe the (7,4) Hamming code, which means there are 7 total bits, 4 message bits and 3 check bits. This satisfies the optimality condition $n = 2^k - 1$ as one can easily check.

Let the check bit positions be at powers of 2, so that out of the 7 bits, the 3 check bits will be at positions 1, 2, and 4. The message bits will be placed in the remaining 4 bits (positions 3,5,6,7). This is shown in Figure XX.

$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$
$\uparrow$	$\uparrow$		$\uparrow$			
c	c		c			

Every number between 1 and 7 can be expressed by adding some combination of the numbers 1, 2, and 4. This is just the binary expansion. Each check bit will check those positions whose binary representation uses the corresponding value. That is,

1	checks positions	$1, 3, 5, 7$
2	checks positions	$2, 3, 6, 7$
4	checks positions	$4, 5, 6, 7$

We will use even parity, so that, for example, check bit 1 is chosen to make the number of 1's in positions 1, 3, 5, 7 even.

Example 10.2 (Example of Coding)

Consider the message 1011. To find the codeword, we place the message in positions 3,5,6,7, as shown:

__	__	1	__	0	1	1
1	2	3	4	5	6	7

Let b_i denote the value of bit i. The check bits are as follows:

$$\begin{aligned}
&\text{check bit 1}: \quad b_1 = 0 \text{ to make } b_1 + b_3 + b_5 + b_7 \text{ even} \\
&\text{check bit 2}: \quad b_2 = 1 \text{ to make } b_2 + b_3 + b_6 + b_7 \text{ even} \\
&\text{check bit 4}: \quad b_4 = 0 \text{ to make } b_4 + b_5 + b_6 + b_7 \text{ even}
\end{aligned}$$

Therefore, the codeword is 0110011.

■

Error correction is also quite simple. To correct a single error, we find which check bits fail the parity check (i.e., have odd parity), and then we add the corresponding values. For example, if check bits 1 and 4 fail, but check bit 2 passes, we add $1 + 4 = 5$. The result tells us which bit has the error. If all checks pass then we have a valid codeword (no errors), and the original message is in positions 3, 5, 6, 7 as they were received.

Example 10.3 (Example of Error Correction)

Suppose we receive 0110111. How can we tell which bit (if any) has an error? We consider each check bit:

$$\begin{aligned}
&\text{check bit 1}: \quad b_1 + b_3 + b_5 + b_7 \text{ is odd } \longrightarrow \text{ fails} \\
&\text{check bit 2}: \quad b_2 + b_3 + b_6 + b_7 \text{ is even } \longrightarrow \text{ passes} \\
&\text{check bit 4}: \quad b_4 + b_5 + b_6 + b_7 \text{ is odd } \longrightarrow \text{ fails}
\end{aligned}$$

Since check bits 1 and 4 failed, the error is in position $1 + 4 = 5$. Therefore, the corrected codeword is 0110011 and the original 4 bit message in positions 3, 5, 6, 7 is 1011.

■

Interestingly, the way the code is constructed, the position of the error can also seen by considering the binary number formed by the check bit results.

Specifically, c_1, c_2, c_4 represent the parity check results where 0 represents passing and 1 represents failure of a parity check. This is the same as adding the bits to be checked modulo 2. Then form the string $c_4c_2c_1$. The result gives the binary representation of the location of the error. For the codeword above, checks 1 and 4 failed while check 2 passed, giving the 101 which is the binary representation of 5. This means the error is in position 5 as we saw above.

10.8 Valid Codewords and Invalid Strings

We've seen that there are a large variety of codes for single error detection and single error correction. Via a representation argument we were able to deduce the best we can do for single error correction using k check bits. The Hamming code achieves this optimality when the total number of bits in the data unit is of the form $n = 2^k - 1$. But how can we design good codes for other values of n? And, what about higher levels of protection?

In this section, we describe a unifying conceptual view of error detection and correction. It will give a useful way to think about the requirements for good codes of a given size and level of protection, although unfortunately it won't tell us exactly what the code should be. Finding good codes and studying their properties is still a very active area.

Using codewords of a given length, a key property that allows any kind of error protection is that only *some* strings are "valid" codewords that correspond to messages. The others are "invalid" strings that arise due to errors.

We assume that the transmitter always sends us a valid codeword corresponding to the message they wish to send. If *every* codeword were "valid" then there's no hope for any kind of error detection or correction. In this case, no matter what bit string we receive, our best guess is that this is what the transmitter intended to send. On the other hand, consider the case where only some subset of strings are valid. We know that these are the only codewords the transmitter would send us. If we receive one of these valid codewords, then we assume there have been no errors, so that the bits received are exactly the ones that were sent. If we receive an invalid string, then we know that an error occurred.

For example, consider the case of 2 message bits and one even parity check bit, resulting in codewords with 3 bits. The codeword 101 is valid since it has even parity, while the codeword 001 is invalid since it has odd parity. Out of the 8 strings with 3 bits, only 4 of them are valid codewords. We should expect this since we started with only 2 message bits, which means there are only $2^2 = 4$ distinct messages that can be represented. Each valid codeword corresponds to one of these messages. This is depicted in Figure 10.8.

This view also frees us from thinking of the bits as either message bits or check bits. The point is there are 4 valid messages, which might correspond to 00, 01, 10, and 11, or more generally just four symbols s_1, s_2, s_3, s_4. Either way, we transmit 2 bits. Separating the bits into message bits versus check bits isn't necessary. In fact, it's much more general to not think in this way. A "code" is

Valid Codewords

Symbol	2 Bit Messages	Check bit	Codewords
s1	0 0	0	0 0 0
s2	0 1	1	0 1 1
s3	1 0	1	1 0 1
s4	1 1	0	1 1 0
		Other (Invalid) Codewords	
			0 0 1
			0 1 0
			1 0 0
			1 1 1

Figure 10.8:

Picking Codewords

s1	000
s2	001
s3	010
s4	100

s1	000
+error	__1
	001

Result looks like code for s2

Figure 10.9:

then just a subset of strings that we call valid codewords and that correspond to messages. The other strings are invalid. How we arrived at the valid codewords (whether by parity check bits or some other way) doesn't matter.

Notice that just picking any 4 valid codewords is not good enough to guarantee single error detection. We need to choose them cleverly. For example, suppose of the 8 strings, we chose 000, 001, 010 and 100 as the valid codewords and the others as invalid. These are shown in Figure 10.9. If 000 were transmitted but an error occurred and we received 001, instead of detecting the error we would erroneously assume that 001 was the transmitted message (received without error). In some sense, to detect or correct errors, the valid codewords need to be "far apart". How to formalize this is discussed in the next section.

10.9 Hamming Distance

The reason the example above fails to be a good code (i.e., doesn't allow even single error detection) is that the valid codewords are not "spread out" enough. Because they are too "close" to each other, a single error can take us from one valid codeword to another valid codeword, thereby preventing any type of error detection.

Formalizing the notion of "distance" between codewords will let us see exactly what condition the valid codewords need to satisfy to allow single error detection. It will also let us address higher levels of protection, both detection and correction.

Definition (Hamming Distance). Given two strings of equal length, the *Hamming distance* between them is defined to be the number of bits on which they disagree.

The reason this is the proper notion for us is that each error changes one bit, so it moves a string by Hamming distance 1. Two errors result in a string that's Hamming distance 2 away from the original, and so on.

If two valid codewords are Hamming distance 1 apart, then a single error in the appropriate bit will take us from one valid codeword to another valid codeword, making detection of the error impossible. We see that to allow single error detection, there must be a distance of at least 2 between *any* valid codewords. Note the emphasis on "any". Even if there are just two valid codewords with distance 1 between them with all the others far apart, we still cannot claim the code will detect single errors. Remember, to make this claim we need to detect *every* single error, no matter which codeword was originally sent. This leads to the following definition.

Definition (Distance of a Code). Given a code (i.e., a set of valid codewords of equal length), the *distance of the code* is defined to be the minimum Hamming distance between any two valid codewords.

To allow single error detection, we see that the distance of the code must be 2, i.e., all valid codewords must be be at least distance 2 apart.

Now, let's think about what is needed in order to correct errors. Remember that we assume a valid codeword is sent from the outset. If we receive an invalid string, we know that an error must have occurred. But, how should we go about trying to correct the error?

What "correcting" the error means is that given the observed invalid string, we find the original valid codeword that was sent. The obvious, and only reasonable, guess is to choose that valid codeword that is closest to the observation, assuming there is a unique closest valid codeword. We'll be able to recover the original codeword this way as long as after the errors, the result is still closer to the original than to any other valid codeword.

For example, if the distance of the code is 3 then all valid codewords are at least distance 3 apart. In this case, even after a single error, the result will still be closer to the original codeword than to any other valid codeword. Hence, we will be able to correct the error. However, we will not be able to detect double errors with a distance of three. If two valid codewords are a distance 3 apart, then certain double errors will cause the received bits to end up closer to a different valid codeword than the original. We'll think there was a single error and improperly "correct" this. This is depicted in Figure 10.10.

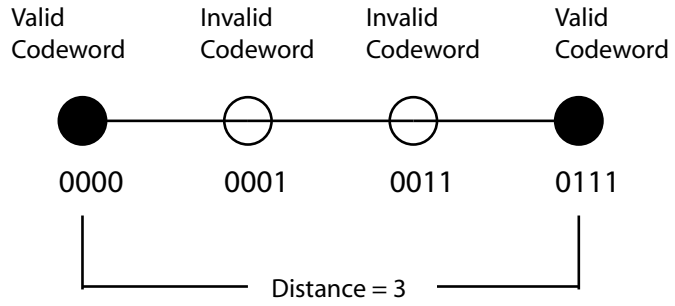

Figure 10.10: Distance 3 code.

To detect double errors and correct single errors, we will need a distance of 4. In this case, single errors will get corrected as before. A double error will leave us equally distant (distance 2) between two valid codewords, and so we will know there was a double error, although we won't know which was the message transmitted. This is depicted in Figure 10.11.

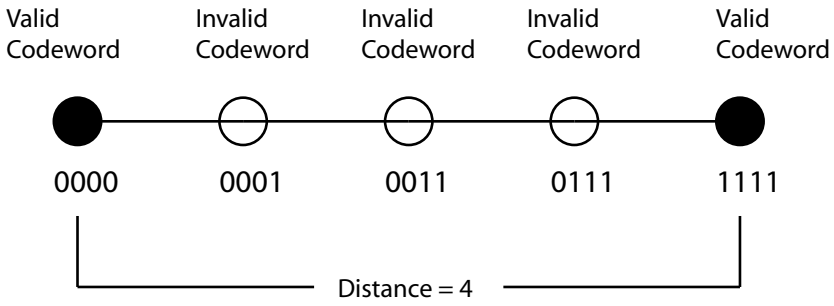

Figure 10.11: Distance 4 code.

A distance of 5 allows double error correction. A distance of 6 allows double error correction and triple error detection. A little thought shows that to correct k errors we need a distance of $2k+1$. To correct k errors and detect $k+1$ errors we need a distance of $2k+2$. Equivalently, if d is the distance of the code, then we can correct $\lfloor \frac{d-1}{2} \rfloor$ errors and can detect $\lfloor \frac{d}{2} \rfloor$ errors, where $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.