**Chapter 10: Error Detection and Correction**

Table of Contents

[10.1 Introduction 4](#_Toc63188513)

[Types of Errors 4](#_Toc63188514)

[Redundancy 4](#_Toc63188515)

[Forward Error Correction 5](#_Toc63188516)

[Retransmission 5](#_Toc63188517)

[Coding 5](#_Toc63188518)

[10.2 Block Coding 6](#_Toc63188519)

[Parity Checking 6](#_Toc63188520)

[Parity Checking 7](#_Toc63188521)

[Hamming Code 9](#_Toc63188522)

[10.3 Linear Block Codes 10](#_Toc63188523)

[Two-Dimensional Parity Checker 10](#_Toc63188524)

[Hamming Code 12](#_Toc63188525)

[10.4 Cyclic Coding 15](#_Toc63188526)

[Cyclic Redundancy Check 15](#_Toc63188527)

[10.5 Checksum 19](#_Toc63188528)

[One’s Complement Method 19](#_Toc63188529)

[Performance 20](#_Toc63188530)

For this chapter, there are some changes in the 5th edition of the book. The changes are not massive, but it is still recommended to follow the 4th edition of the book for this chapter.

While sending data, some extra information is generated at both the sender and receiver sides. The sender sends this extra information along with the data and the receiver matches this with the information generated at their own end. This is essentially how errors are detected.

## 10.1 Introduction

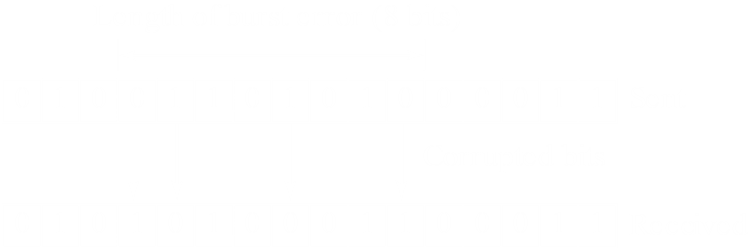
### Types of Errors

There are two types of errors:

* Single Errors
* Burst Errors

Single errors are simple enough. They are errors where a single bit in the data unit has changed.

Burst errors are when more than 1 bit has been changed. The length of the burst is measured from the first changed bit to the last one. In between, there may be bits that are unchanged.



### Redundancy

To detect or correct errors, we need to send some extra, redundant bits with the original data, also called the dataword. The entire result, with the dataword and the redundant bits, is called the codeword. This process is called redundancy. These extra bits are discarded as soon as the accuracy of the transmission has been determined.

### Forward Error Correction

Forward error correction is the process by which the receiver tries to guess the message and correct it using the redundant bits.

### Retransmission

Retransmission is a technique where the receiver detects some error and asks the sender to resend that data unit. This is the normal error correction mechanism.

### Coding

Redundancy can be achieved through various coding schemes. Two of these are block coding and convolution coding. We will only be discussing block coding for now.

In coding schemes, redundant bits are added to the data through a process in a way such that there is a relationship between the redundant bits and the actual data bits.

We have already seen block coding in previous chapters. In block coding, we replace a -bit block of code with an -bit block, thereby adding redundancy.

In block coding, we will need modulo-2 arithmetic. In module-2 arithmetic, both addition and subtraction of bits follows the XOR operation. If both bits are the same, we will get a and if they are different, we will get a .

## 10.2 Block Coding

In block coding, we divide the data into bit blocks, and then add bits to each of those blocks to produce bit blocks. The mapping is one-to-one. Since there are bits in a block, there are possible combinations. Each combination is called a dataword. For the bit blocks produced, there are possible combinations, each of which are called codewords.

For example, if we take -bit datawords, there can be possible combinations. If we add bits to each of the datawords, we will get -bit codewords. There are codewords possible. We only need of these to represent the actual data. The other codewords provide redundancy.

On the receiver side, the same process is repeated in the opposite direction. If the receiver finds an invalid codeword, it realizes that there is an error.

### Parity Checking

There are many ways to create codewords from datawords. One example is a parity checker. Consider the example below:

|  |  |
| --- | --- |
| Datawords | Codewords |
| 00 | 000 |
| 01 | 011 |
| 10 | 101 |
| 11 | 110 |

One extra bit, called the parity bit, is being added to the end of the datawords. This bit should make the codeword have even parity, i.e. a is added if there are an even number of s in the dataword and a is added if there are an odd number of s in the dataword.

We will deal with two types of codewords, linear codewords and cyclic codewords. The above example is for a linear codeword. One property of a set of linear codewords is if we take any two of them and perform an XOR operation, the result will be another codeword. Thus, this is a linear block coding scheme. We will look into more details about this later on.

The fundamental concept of block coding and redundancy is that the data being sent is divided into blocks of bits each, bits are added to each of the blocks and the resulting bit blocks are sent to the receiver. The receiver takes these bit codewords and matches them with a list of valid codewords. If it finds a mismatch, it realizes that there was an error during transmission and asks for that block to be retransmitted.

### Parity Checking

Say and . The possible datawords are

If we want to use even parity, the corresponding codewords will be

The last bit that we added is called the parity bit. On the sender’s side, the parity bit is generated using a generator.

The receiver knows these valid codewords. However, there are possible codewords, so if the receiver receives one of the invalid codewords, it will know there is an error. The received codeword is checked using a checker and based on this, a value called the ‘syndrome’ is generated. This value is used with some logic to identify whether or not there was an error.

The interesting part is that if a single bit is changed in the valid codewords, it is guaranteed to be one of the invalid codewords. Changing one bit from a codeword can never give us another valid codeword.

Consider that is being sent. The valid codeword for this is . If the second bit is changed somehow, it will become . This is not a valid codeword, so the error will be detected.

However, the problem here is that if two different bits are changed in a codeword, the receiver will be unable to detect it. This is because changing two bits from a valid codeword will produce another valid codeword.

For the valid codeword , if the second and third bits are both changed, it becomes . This is another valid codeword.

In reality, the parity checking scheme, whether we use even parity or odd parity, will be unable to detect an even number of errors.

### Hamming Code

A major limitation of parity checking is that it cannot tell us where exactly an error has occurred. To overcome this limitation, we can use hamming code.

The hamming distance between two words is the number of differences in the words. Thus, for the words and , the hamming distance is . The hamming distance can be found by XORing the two words and counting the number of s in the result.

The smallest hamming distance between all possible codewords in a set is the minimum hamming distance, . For the set , , and , .

The reason the minimum hamming distance is important is because any coding scheme can be defined using three parameters, , and . The coding scheme, , is written as and a separate expression is used for .

For example, the first example we saw in this lecture can be defined as with .

To be able to detect bits of error, the set of codewords must have a minimum hamming distance of . This will ensure that the received codeword with errors does not match any of the valid codewords. This explains why the first example could only detect bit errors.

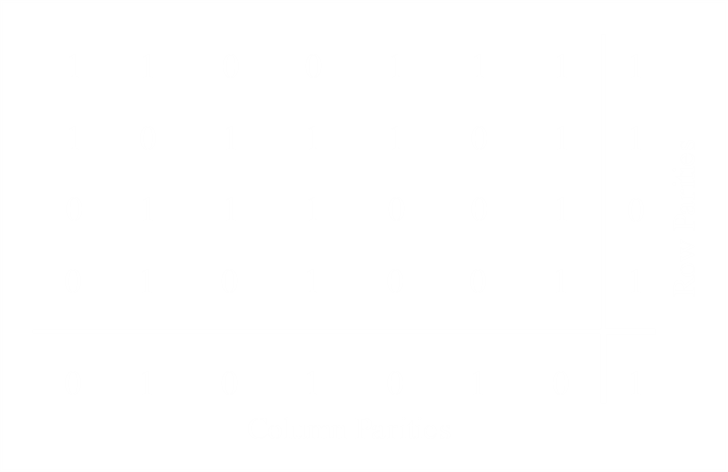
Think about why this works. For the first example, the value is , meaning any two valid codewords have at least bits different. If we get a codeword that has only bit of difference, obviously this codeword cannot be any of the valid codewords and must therefore be an error. We shall later see that this formula holds for error detection, but for error correction, .

## 10.3 Linear Block Codes

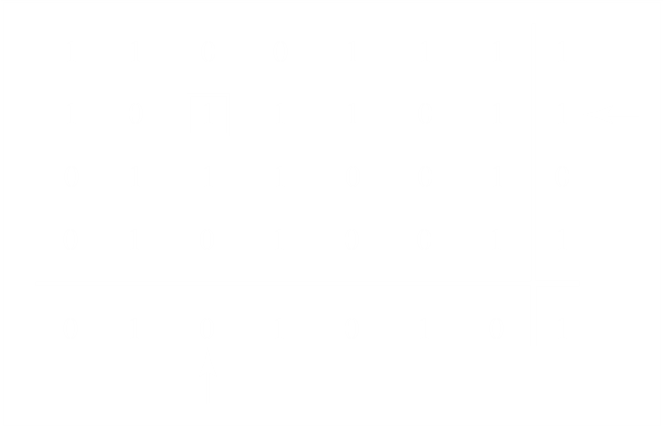
For linear block codes, if we XOR any two valid codewords, the result will be a valid codeword. If we look at the examples we have seen so far, we will find that they were all linear block codes.

### Two-Dimensional Parity Checker

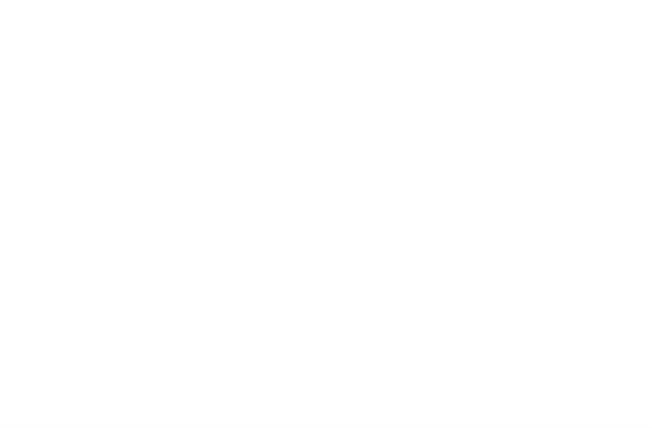
We have previously seen a simple parity checker and come to the conclusion that we are unable to detect an even number of errors. We can extend this to detecting two errors by using a two-dimensional parity checker.



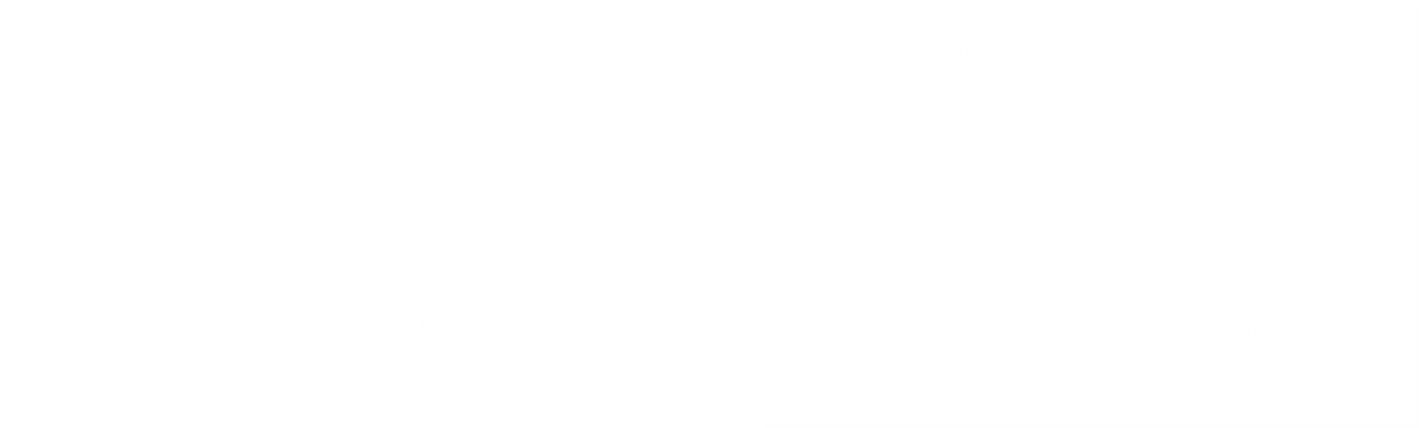
Here we have four rows of -bit blocks. Each row produces a parity bit. As we know, we can use those parity bits to check for single-bit errors in any of the rows. Additionally, each column also produces a parity bit, which gives us an extra row of column parities. Thus, a single-bit error will affect not only its own row parity bit but also its column parity bit, meaning we will get errors in two different parity bits.



On the other hand, if there are two errors in a single row, the row parity bit will be unable to detect it. However, we will get errors in two different column parity bits.

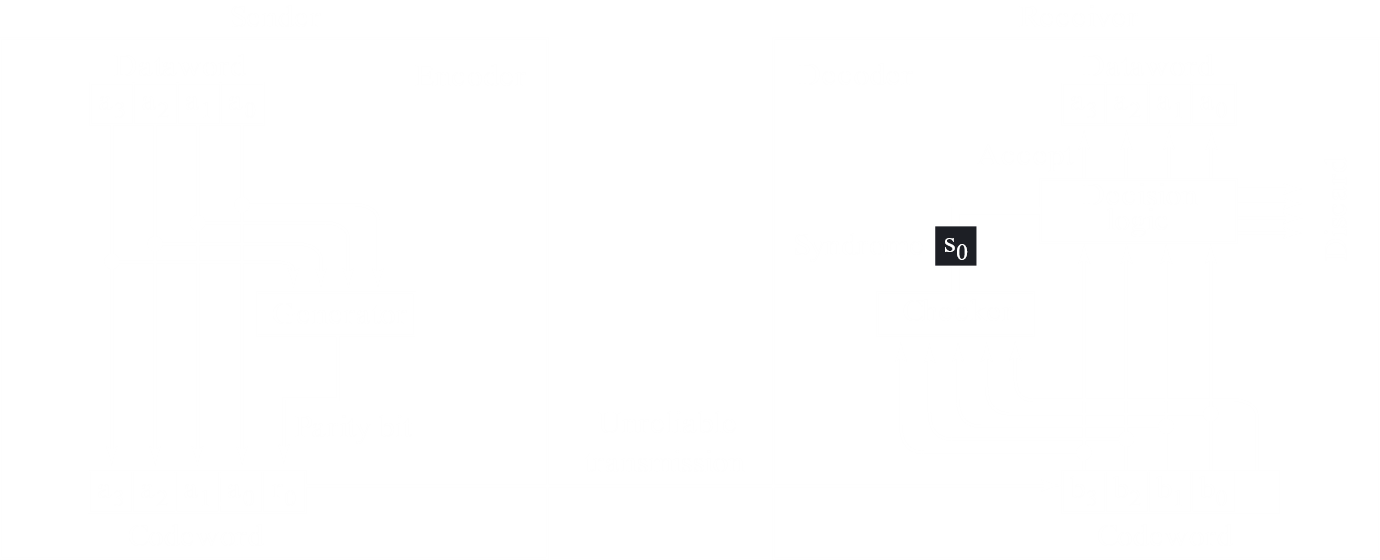


For three errors, we will still be able to detect it. We will even be able to detect four errors. The only time we cannot detect the errors is if we get into a situation like the one on the right below, with four errors in a square shape such that both row parities are unable to detect the errors and both column parities are also unable to detect the errors.



### Hamming Code

Hamming code is defined as , meaning the size of the dataword is bits and the size of the codeword is bits. It was originally designed with a minimum hamming distance of . Thus, it is capable of detecting up to bits of error and correcting bit of error.



We can see that on the receiver’s side, a -bit syndrome is generated. If this syndrome is , then it means that there is no error. However, if it is not , based on the value of the syndrome, we can tell which bit is corrupted. With bits, it can have values from to , and is the exact number of bits in the codeword.

#### Parity Bit Generation

The three parity bits are generated using the following formulae:

On the receiver’s side, the syndrome bits are generated using the following formulae:

Now take a look at the table for syndrome values.

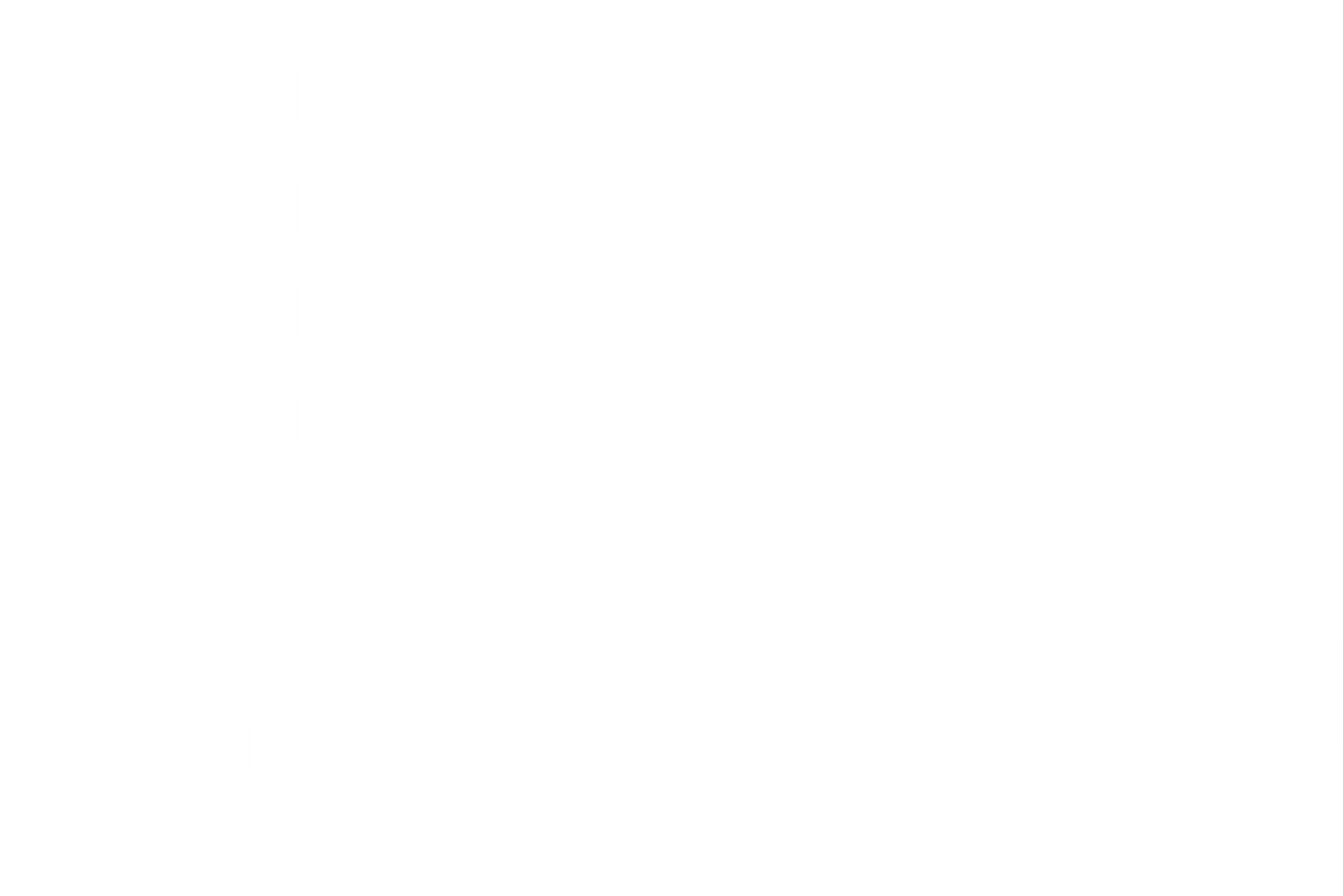
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Syndrome | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Error | None |  |  |  |  |  |  |  |

Consider the value of . was originally . If you look carefully, was the only bit that was present in all three formulae. Thus, has the value since if is changed, all three parity bits are affected. Similarly, has the value since only the first and third parity bits are affected if is changed.

However, there are no dataword bits that are present in only one of the formulae for generating parity bits. As such, if only one of the bits is changed, it means that the corresponding parity bit is the one that has an error.

#### Correcting Burst Errors

Hamming code can be used intelligently to allow us to detect burst errors as well.



From the sender’s side, the codewords are broken up and sent as one long stream. The bits are grouped based on their position in the original codewords, i.e. the bits in the first position are all together, the bits in the second position are all together and so on.

Now if a burst error occurs, it will affect multiple different codewords instead of affecting just one codeword. Whereas hamming code would fail to detect multiple errors in a single codeword, since the errors are spread out, hamming code can be used on each of the received codewords to detect and correct the errors.

Of course, there are problems like the burst being so large that multiple bits in a single codeword are still affected. In those cases, even this will not work.

## 10.4 Cyclic Coding

With linear codes, the property was that if any two codewords are XORed, the result is another codeword. Cyclic codes are a special type of linear code that have an additional property. If any codeword is cyclically shifted, the result is another codeword.

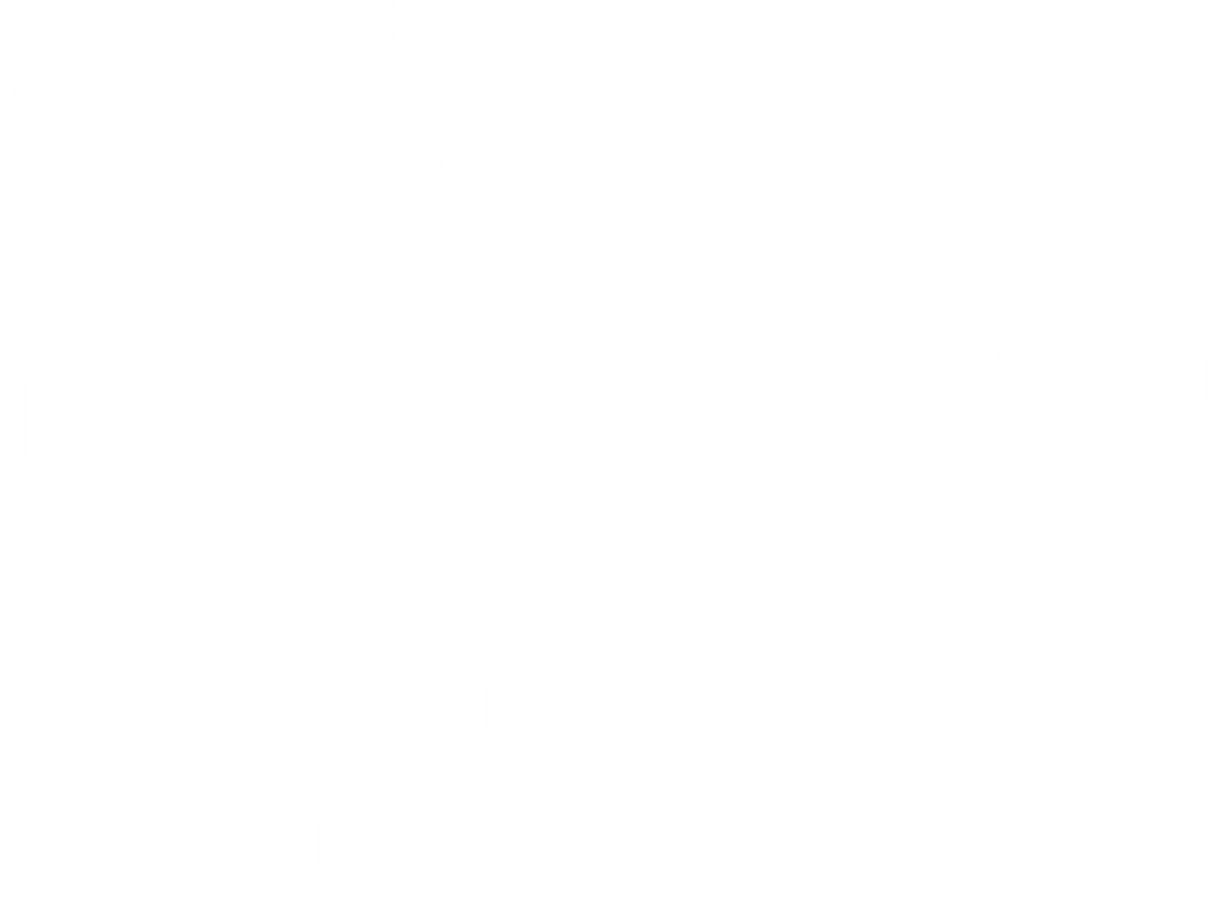
For example, if the codeword is and we perform a cyclic left-shift, the result, is also a codeword.

### Cyclic Redundancy Check

One of the most famous cyclic codes is cyclic redundancy check (CRC). In CRC, both the sender and receiver agree upon a divisor, normally . The sender divides the data by this divisor, which results in some remainder. This remainder is added to the data as the CRC and the entire thing is sent to the receiver. On the receiver side, the entire data and CRC is divided by the divisor. If the result is , then the data is accepted as valid. Otherwise, it is rejected as an error.

CRCs must have two properties:

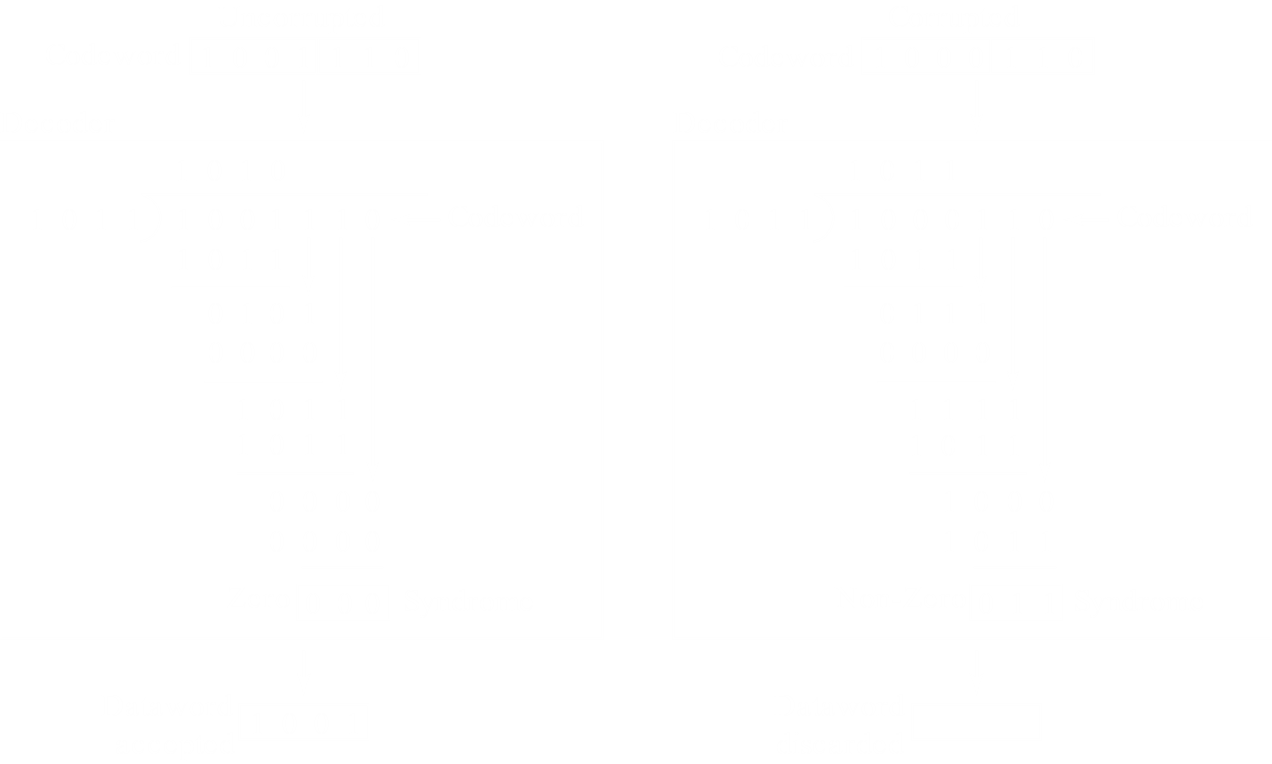
1. It must have exactly bit less than the divisor.
2. The resulting bit sequence (data + CRC) must be exactly divisible by the divisor.



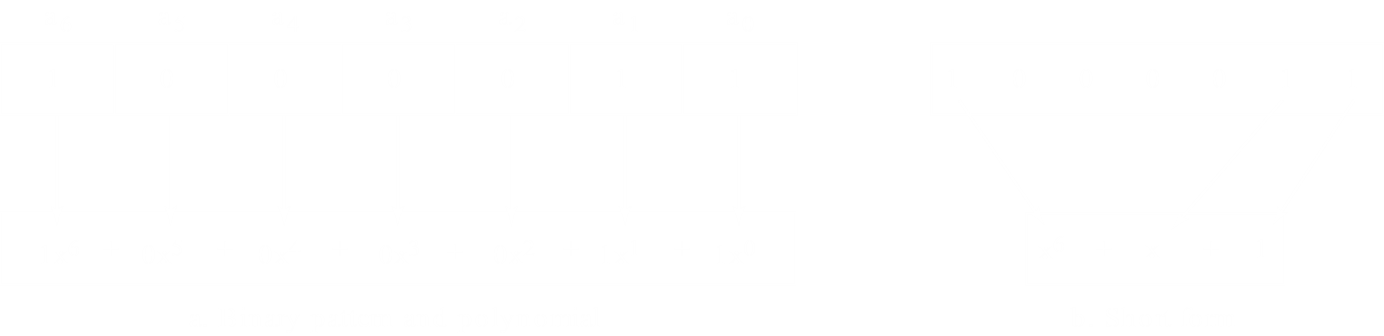
In this example, the dataword is . Three bits are added to the right of the data word to create a -bit dividend. These three bits will be replaced by the remainder. Thus, like the parity bits in hamming code, CRC also has bits.

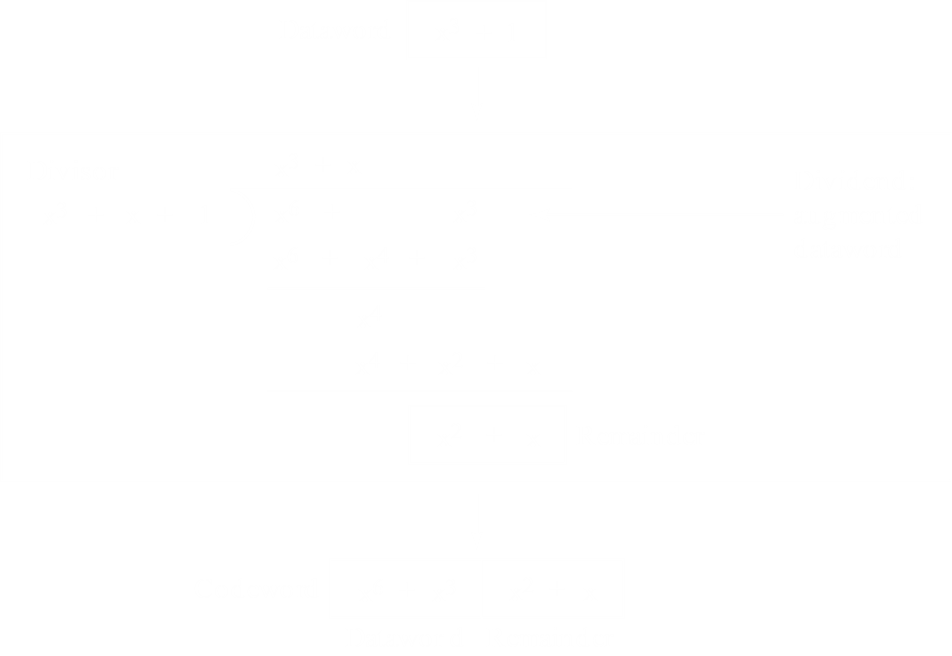
If the MSB at any step is , the quotient has a added to it. Otherwise, the quotient has a added to it.

On the receiver’s end, we could have two cases. Either there was no error, or there was an error. Examples of both cases are shown below.



We can perform the same operation using polynomial expressions. This might be a little easier to work out. Consider this other example:





Because of the polynomial version of this, the divisor of a cyclic code is normally called the generator polynomial, or simply, the generator.

There are two advantages to using polynomial expressions. Firstly, we can use the short form. This means we need fewer bits to represent the data, since the s are ignored. During the actual transmission though, we will still have to use binary, so this advantage is just for us during calculations. Secondly, in the original form, we technically had two divisors. For the example seen, the divisors were and . For the polynomial example, we did not need to take into consideration. A single divisor was enough.

CRCs are normally used in networks line LANs and WANs because they are very fast when implemented using hardware.

## 10.5 Checksum

Say there are 5 integers, , , , and . The idea of checksum is that, along with this data, the sum of the data, which is in this case, will also be sent. On the receiver’s end, the data will again be added up and if the sum matches, then the conclusion will be reached that there were no errors. The process can be made a little easier if we send the negated sum, i.e. . In that case, the receiver can just add up all the values, including the checksum, and if the result is , then there are no errors. This negated sum is called the checksum.

If we had a message instead of integers, the message could still be converted to some values using encoding schemes like ASCII.

### One’s Complement Method

There is one major drawback to the previous example. The four integers in the actual data can all be represented using bits, but the checksum cannot. If we had a limitation that we can only use bits, we would be in trouble here. The solution is to use One’s Complement.

In one’s complement, if we need to represent a number in bits, the extra bits on the left are wrapped around and added. For example, is represented as . The two left-most bits are wrapped around and added. Thus, . This value is in decimal.

For the value , the checksum we get is the negation of . In one’s complement, negating a number means flipping all of its bits. Thus, the value we get is , which is in decimal.

We now send the data along with the checksum, , , , , and .

On the receiver’s end, this data is received and summed up, resulting in , or . Using the same wrapping, this gives us . Negating this, we get , or in decimal. Thus, there were no errors.

### Performance

Traditionally, -bit checksums are used for messages of any size. Checksums are not as strong as CRC, since if there are two errors where one value is incremented and another is decremented, the errors cannot be detected using checksums. There are some workarounds to this, but the tendency in the Internet is to use CRCs instead.