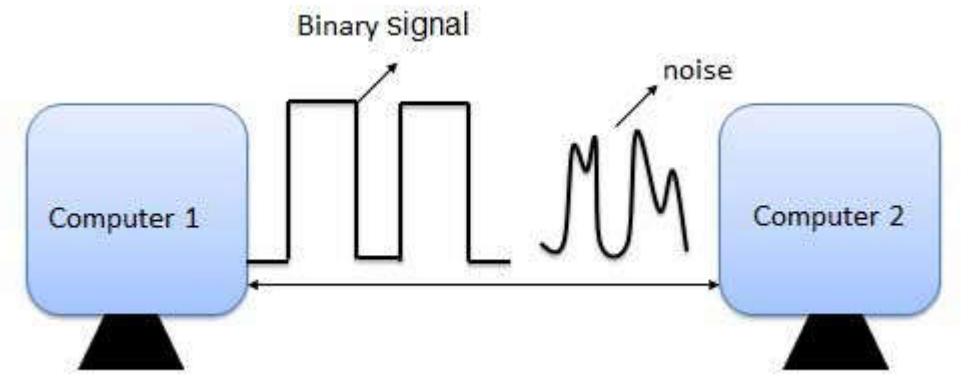
Error Detection and Correction Codes[1]

What is Error?

Error is a condition when the output information does not match with the input information. During transmission, digital signals suffer from noise that can introduce errors in the binary bits travelling from one system to other. That means a 0 bit may change to 1 or a 1 bit may change to 0.



Error-Detecting codes

Whenever a message is transmitted, it may get scrambled by noise or data may get corrupted. To avoid this, we use error-detecting codes which are additional data added to a given digital message to help us detect if an error occurred during transmission of the message. A simple example of error-detecting code is **parity check**.

Error-Correcting codes

Along with error-detecting code, we can also pass some data to figure out the original message from the corrupt message that we received. This type of code is called an **error-correcting code**. Error-correcting codes also deploy the same strategy as error-detecting codes but additionally, such codes also detect the exact location of the corrupt bit.

In error-correcting codes, parity check has a simple way to detect errors along with a sophisticated mechanism to determine the corrupt bit location. Once the corrupt bit is located, its value is reverted (from 0 to 1 or 1 to 0) to get the original message.

To detect and correct the errors, additional bits are added to the data bits at the time of transmission.

- The additional bits are called **parity bits**. They allow detection or correction of the errors.
- The data bits along with the parity bits form a **code word**.

[1] https://www.tutorialspoint.com/computer_logical_organization/error_codes.htm

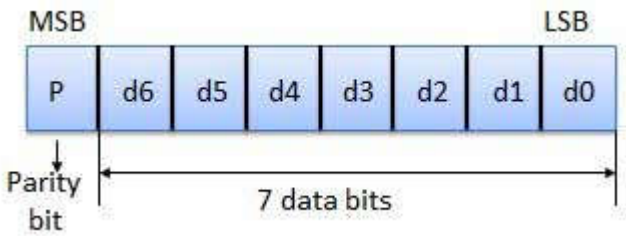
Parity Checking of Error Detection

In this text

- **MSB** – the most significant bit – the first bit,
- **LSB** – the least significant bit – the last bit.

MSB															**LSB**
1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	**0**

The parity check is the simplest technique for detecting and correcting errors. The MSB of an 8-bits word is used as the parity bit and the remaining 7 bits are used as data or message bits. The parity of 8-bits transmitted word can be either even parity or odd parity.



Even parity -- Even parity means the number of 1's in the given word including the parity bit should be even (2,4,6,....).

Odd parity -- Odd parity means the number of 1's in the given word including the parity bit should be odd (1,3,5,....).

Use of Parity Bit

The parity bit can be set to 0 and 1 depending on the type of the parity required.

- For **even parity**, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word is even. Shown in fig. (a).

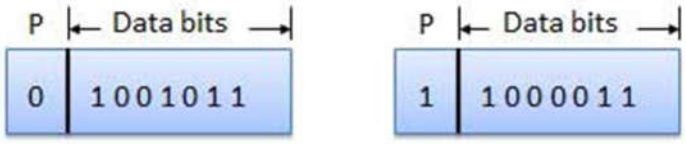


Fig. (a)

- For **odd parity**, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word is odd. Shown in fig. (b).

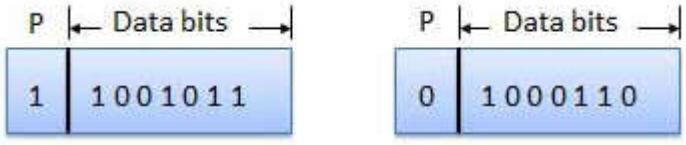
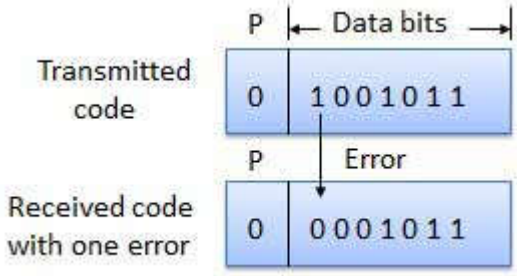


Fig. (b)

How Does Error Detection Take Place?

Parity checking at the receiver can detect the presence of an error if the parity of the receiver signal is different from the expected parity. That means, if it is known that the parity of the transmitted signal is always going to be "even" and if the received signal has an odd parity, then the receiver can

conclude that the received signal is not correct. If an error is detected, then the receiver will ignore the received byte and request for retransmission of the same byte to the transmitter.



Repetition Code

- Quite primitive and inefficient
- A predefined group of bits is repeated several times – if all repetitions are equal, the transmission was successful; otherwise the data is incorrect and it has to be re-sent.
- Example – send 3 bits 3-times:
  - 110 → sent code: 110 110 110 → received code 110 110 110 → OK
  - 101 → sent code: 101 101 101→ received code 101 100 101 → incorrect

CRC – Cyclic Redundancy Check

The principle of the CRC is a cyclic calculation of a value, which is typical of a group of bits. The outcome is then attached to the data.

DATA	Attached CRC

When the data is received, it undergoes the same calculation, whose result is then compared with the attached result. If there is any difference, an error has occurred.

DATA	Attached CRC

Recalculated CRC from DATA

Compare CRCs:

- *Equal → OK*
- *Different → problem*

Mathematically, it is a series of divisions between polynomials. Fortunately, in the binary data it is equivalent to XOR operation.

XOR – exclusive disjunction (either – or)

Truth table

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Example of CRC

- 4-bit CRC divisor: 1101
- Data to be sent: **1010 1101** (1 Byte)

1. Add 3 zeroes (the number of zeroes is the length of divisor – 1) → 1010 1101 000
2. Align the data and divisor, then perform XOR:

```
1010 1101 000
1101
-------------
0111 1101 000
```

3. Take the result as the new data, move the divisor towards right so it matches the first 1 in the data, and return to the step 2., until the data Byte is zeroed.

```
0111 1101 000
 110 1
-------------
0001 0101 000


0001 0101 000    (here it moved by 2 positions towards right)
   1 101
-------------
0000 1111 000


0000 1111 000
     1101
-------------
0000 0010 000


0000 0010 000
       11 01
-------------
0000 0001 010
```

```
0000 0001 010
        1 101
-------------
0000 0000 111   The bits of the original byte are zeroed.
```

The last 3 bits (in this case 111) are the resulting CRC code, which is added to the data.

In real cases the divisor is much longer – e.g. CRC32 uses 32-bit divisor (0x04C11DB7).

Hamming Code

Calculating the Hamming Code

The key to the Hamming Code is the use of extra parity bits to allow the identification of a single error. Create the code word as follows:

1. **Mark all bit positions that are powers of two as parity bits.** (positions 1, 2, 4, 8, 16, 32, 64, etc.)
2. **All other bit positions are for the data to be encoded.** (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)
3. **Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.**
   Position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc. (1,3,5,7,9,11,13,15,...)
   Position 2: check 2 bits, skip 2 bits, check 2 bits, skip 2 bits, etc. (2,3,6,7,10,11,14,15,...)
   Position 4: check 4 bits, skip 4 bits, check 4 bits, skip 4 bits, etc. (4,5,6,7,12,13,14,15,20,21,22,23,...)
   Position 8: check 8 bits, skip 8 bits, check 8 bits, skip 8 bits, etc. (8-15,24-31,40-47,...)
   Position 16: check 16 bits, skip 16 bits, check 16 bits, skip 16 bits, etc. (16-31,48-63,80-95,...)
   Position 32: check 32 bits, skip 32 bits, check 32 bits, skip 32 bits, etc. (32-63,96-127,160-191,...)
   etc.
4. **Set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.**

Example

A byte of data: **10011010**
Create the data word, leaving spaces for the parity bits: _ _ 1 _ 0 0 1 _ 1 0 1 0, or in the tabular form:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Bit	_	_	1	_	0	0	1	_	1	0	1	0

Calculate the parity for each parity bit (? represents the bit position being set):

Position 1 checks bits 1,3,5,7,9,11:

Position	**1**	2	**3**	4	**5**	6	**7**	8	**9**	10	**11**	12
Bit	?	_	1	_	0	0	1	_	1	0	1	0

The bits at these positions are **?10111** – the known bits have an even parity so set position 1 to a 0:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Bit	0	_	1	_	0	0	1	_	1	0	1	0

Position 2 checks bits 2,3,6,7,10,11:

Position	1	**2**	**3**	4	5	**6**	**7**	8	9	**10**	**11**	12
Bit	0	?	1	_	0	0	1	_	1	0	1	0

The bits at these positions are **?10101** – the known bits have an odd parity so set position 2 to a 1 to get the even parity:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Bit	0	1	1	_	0	0	1	_	1	0	1	0

Position 4 checks bits 4,5,6,7,12:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Bit	0	1	1	?	0	0	1	_	1	0	1	0

The bits at these positions are ?0010 – the known bits have an odd parity so set position 4 to a 1:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Bit	0	1	1	1	0	0	1	_	1	0	1	0

Now the last missing bit - position 8 checks bits 8,9,10,11,12 (and if there were more bits, then also positions 13,14,15, then 24,25,26,27,28,29,30,31):

Position	1	2	3	4	5	6	7	8	9	10	11	12
Bit	0	1	1	1	0	0	1	?	1	0	1	0

Even parity so set position 8 to a 0:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Bit	0	1	1	1	0	0	1	0	1	0	1	0

The final code word: **011100101010**.

Finding and fixing a bad bit

The above example created a code word of **011100101010**. Suppose the word that was received was **011100101110** instead. Then the receiver could calculate which bit was wrong and correct it. The method is to verify each check bit.

Write down all the incorrect parity bits. Doing so, you will discover that parity bits 2 and 8 are incorrect. It is not an accident that 2 + 8 = 10, and that bit position 10 is the location of the bad bit. In general, check each parity bit, and add the positions that are wrong, this will give you the location of the bad bit.

Try one yourself

Test if these code words are correct, assuming they were created using an even parity Hamming Code. If one is incorrect, indicate what the correct code word should have been. Also, indicate what the original data was.

- 010101100011
- 111110001100
- 000010001010

Unfortunately, there is no way of how to distinguish the situations with 1 and 2 errors.