ERROR DETECTION AND CORRECTION

While transmit the information from source to destination some of the information is corrupted due to external environment .so there is need of error correction and error detection codes are required. These codes can correct and detect errors. The general definitions of the terms are as follows:

- Error detection is the detection of errors caused by noise or other impairments during transmission from the transmitter to the receiver.[1]
- Error correction is the detection of errors and reconstruction of the original, error-free data.

one bit error detection codes

One bit Error detection code is parity check. A **parity bit** is a bit that is added to ensure that the number of set bits (i.e., bits with the value 1) in a group of bits is even or odd. A parity bit can only detect an odd number of errors (i.e., one, three, five, etc. bits that are incorrect).

There are two variants of parity bits:

1. even parity bit

2.odd parity bit.

When using even parity, the parity bit is set to 1 if the number of ones in a given set of bits (not including the parity bit) is odd, making the entire set of bits (including the parity bit) even. When using odd parity, the parity bit is set to 1 if the number of ones in a given set of bits (not including the parity bit) is even, making the entire set of bits (including the parity bit) odd. In other words, an even parity bit will be set if the number of set bits plus one is even, and an odd parity bit will be set if the number of set bits plus one is odd.

There is a limitation to parity schemes. A parity bit is only guaranteed to detect an odd number of bit errors. If an even number of bits (i.e., two, four, six, etc.) are flipped, the parity bit will appear to be correct even though the data is erroneous. Extensions and variations on the parity bit mechanism are horizontal redundancy checks, vertical redundancy checks, and "double," "dual," or "diagonal" parity (used in RAID-DP).

Error detection and correction codes

HAMMING CODE

Hamming code is one bit error detection and correcting code. In mathematical terms, Hamming codes are a class of binary linear codes. For each integer $m > 2$ there is a code with m parity bits and $2^m - m - 1$ data bits. The parity-check matrix of a Hamming code is constructed by listing all columns of length m that are pair wise independent

Parity adds a single bit that indicates whether the number of 1 bits in the preceding data was even or odd. If an odd number of bits is changed in transmission, the message will change parity and the error can be detected at this point. (Note that the bit that changed may have been the parity bit itself!) The most common convention is that a parity value of 1 indicates that there is an odd number of ones in the data, and a parity value of 0 indicates that there is an even number of ones in the data. In other words: The data and the parity bit together should contain an even number of 1s.

Parity checking is not very robust, since if the number of bits changed is even, the check bit will be valid and the error will not be detected. Moreover, parity does not indicate which bit contained the error, even when it can detect it. The data must be discarded entirely and re-transmitted from scratch. On a noisy transmission medium, a successful transmission could take a long time or may never occur. However, while the quality of parity checking is poor, since it uses only a single bit, this method results in the least overhead. Furthermore, parity checking does allow for the restoration of an erroneous bit when its position is known

If more error-correcting bits are included with a message, and if those bits can be arranged such that different incorrect bits produce different error results, then bad bits could be identified. In a 7-bit message, there are seven possible single bit errors, so three error control bits could potentially specify not only that an error occurred but also which bit caused the error.

Hamming studied the existing coding schemes, including two-of-five, and generalized their concepts. To start with, he developed a nomenclature to describe the system, including the number of data bits and error-correction bits in a block. For instance, parity includes a single bit for any data word, so assuming ASCII words with 7-bits, Hamming described this as an (8,7) code, with eight bits in total, of which 7 are data. The repetition example would be (3,1), following the same logic. The code rate is the second number divided by the first, for our repetition example, 1/3.

Hamming also noticed the problems with flipping two or more bits, and described this as the "distance" (it is now called the Hamming distance, after him). Parity has a distance of 2, as any two bit flips will be invisible. The (3,1) repetition has a distance of 3, as three bits need to be flipped in the same triple to obtain another code word with no visible errors. A (4, 1) repetition (each bit is repeated four times) has a distance of 4, so flipping two bits can be detected, but not corrected. When three bits flip in the same group there can be situations where the code corrects towards the wrong code word.

Hamming was interested in two problems at once; increasing the distance as much as possible, while at the same time increasing the code rate as much as possible. During the 1940s he developed several encoding schemes that were dramatic improvements on existing codes. The key to all of his systems was to have the parity bits overlap, such that they managed to check each other as well as the data.

GENERAL ALGORITHM

The following general algorithm generates a single-error correcting (SEC) code for any number of bits.

1. Number the bits starting from 1: bit 1, 2, 3, 4, 5, etc.
2. Write the bit numbers in binary. 1, 10, 11, 100, 101, etc.

3. All bit positions that are powers of two (have only one 1 bit in the binary form of their position) are parity bits.
4. All other bit positions, with two or more 1 bit in the binary form of their position, are data bits.
5. Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position.
    1. Parity bit 1 covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself), 3, 5, 7, 9, etc.
    2. Parity bit 2 covers all bit positions which have the second least significant bit set: bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.
    3. Parity bit 4 covers all bit positions which have the third least significant bit set: bits 4–7, 12–15, 20–23, etc.
    4. Parity bit 8 covers all bit positions which have the fourth least significant bit set: bits 8–15, 24–31, 40–47, etc.
    5. In general each parity bit covers all bits where the binary AND of the parity position and the bit position is non-zero.

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.

Eexample:

A byte of data: 10011010
Create the data word, leaving spaces for the parity bits: _ _ 1 _ 0 0 1 _ 1 0 1 0
Calculate the parity for each parity bit (a ? represents the bit position being set):

- Position 1 checks bits 1,3,5,7,9,11:
  ? _ 1 _ 0 0 **1 _ 1** 0 **1** 0. Even parity so set position 1 to a 0: **0 _ 1 _ 0** 0 **1 _ 1** 0 **1** 0
- Position 2 checks bits 2,3,6,7,10,11:
  0 **? 1** _ 0 **0 1** _ 1 **0 1** 0. Odd parity so set position 2 to a 1: 0 **1 1** _ 0 **0 1** _ 1 **0 1** 0
- Position 4 checks bits 4,5,6,7,12:
  0 1 1 **? 0 0 1** _ 1 0 1 **0**. Odd parity so set position 4 to a 1: 0 1 1 **1 0 0 1** _ 1 0 1 **0**
- Position 8 checks bits 8,9,10,11,12:
  0 1 1 1 0 0 1 **? 1 0 1 0**. Even parity so set position 8 to a 0: 0 1 1 1 0 0 1 **0 1 0 1 0**
- Code word: 011100101010.