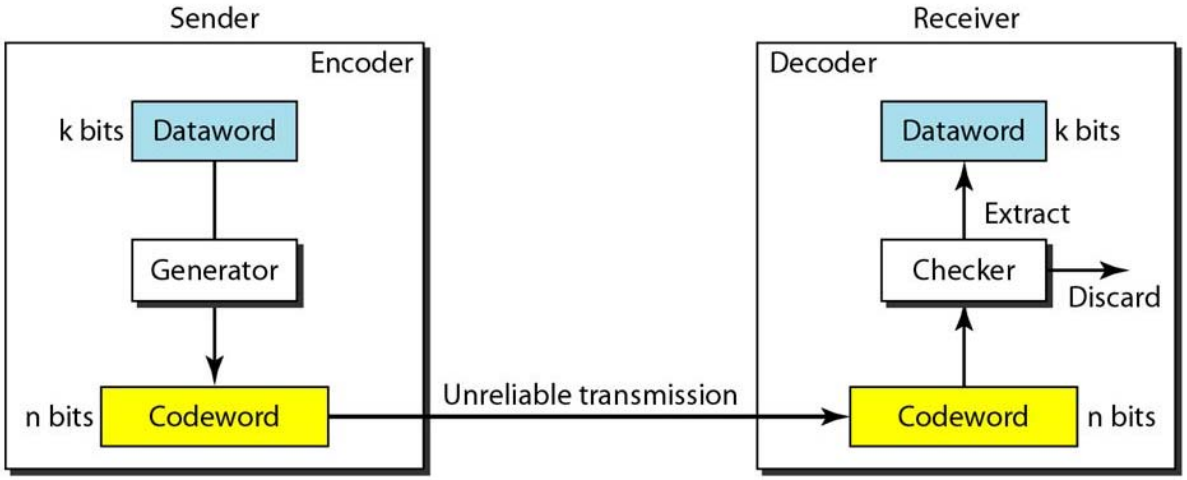
Data Communication Assignment 1

Instructor: Asst. Prof. Boonsit Yimwadsana, Asst. Prof. Thitinan Tantidham

Submission deadline: May 10, 2020
This assignment can be done in a group of one or two students.
Scoring: 100 points (each problem is 20 points)
Criteria: Correctness of output – 50 %, Correctness of methods – 50%

Problem:
In data communication, error can happen during the transmission of data from sender to receiver over unreliable medium.



In order for the receiver to check for error, redundancy bits are added to the dataword to help the receiver verifies the validity of the received data (codeword).

Use the knowledge you learnt in class, implement the following programs or functions and write a demo code showing how all these functions can be used for error control.

1. **Unreliable transmission**: out_frame = *unreliable_transmission*(in_frame)
   Implement unreliable transmission function which simulates an unreliable transmission link between a sender and the receiver. This function is described by its prototype as follows:

   Input:
   1. bit string of any size up to k (input frame).
   2. Probability of having a bit in error (p).

   Output:
   1. bit string of size k defined by the input bit string (output frame).

Behavior:

The function randomly generates errors on the bits of the input frame. A bit can be in error with probability *p*. The function outputs an output frame which could be a corrupted or uncorrupted frame.

Testing:

You must show in your report, for p = 0.05, 0.1, and 0.2, the average ratio of erroneous bits and total number of bits in the received codeword after 100 trials for each value of *p*.

2. **Parity bit**:

Generator: codeword = *parity_gen*(dataword, parity_type, size)

Implement a function that generates one or two-dimensional even parity codeword to an input dataword.

Input:

1. A bit string of any size up to k (input frame) where k ≥ 9 (input frame)
2. Type of parity (even, odd, two-dimensional-even, two-dimensional-odd)
3. Size of two-dimensional block for dataword (if two-dimensional parity bit is used).

Output:

1. A code word based on the type of parity

Behavior:

This function calculates the redundancy bit(s) for the dataword and output the codeword. If two-dimensional parity check is used, the size of the two-dimensional block for dataword must be specify. For example, 3x3, 4x4.

Testing:

Check if this function encodes the dataword correctly for at least 10 samples.

Checker: validity = *parity_check*(codeword, parity_type, size)

Input:

1. A bit string of any size up to k (codeword)
2. Type of parity (even, odd, two-dimensional-even, two-dimensional-odd)
3. Size of two-dimensional block for dataword (if two-dimensional parity bit is used).

Output:

1. Validity of codeword

Behavior:

This function verifies the codeword.

Testing:

Check if this function verifies the codeword correctly for at least 10 samples.

3. **CRC**

Generator: codeword = *CRC_gen*(dataword, CRC-type)

Implement a function that generates CRC codeword for an input dataword.

Input:

1. Dataword of size k
2. CRC-type (string e.g, CRC-8, CRC-16, CRC-32, …)

Output:

1. A codeword based on CRC

Behavior:

This function calculates the redundancy bit(s) for the dataword and output the codeword for CRC. Hint: you have to create a modulo-2 division function. Use the CRC divisor as shown in the table below.

Testing:

Check if this function encodes the dataword correctly for at least 10 samples.

CRC Method	Generator Polynomial	Number of Bits
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	32
CRC-24	$x^{24}+x^{23}+x^{14}+x^{12}+x^8+1$	24
CRC-16	$x^{16}+x^{15}+x^2+1$	16
Reversed CRC-16	$x^{16}+x^{14}+x+1$	16
CRC-8	$x^8+x^7+x^6+x^4+x^2+1$	8
CRC-4	$x^4+x^3+x^2+x+1$	4

Checker: validity = *CRC_check*(codeword, CRC-type)

Input:

1. codeword size k
2. CRC-type (string e.g, CRC-8, CRC-16, CRC-32, …)

Output:

1. Validity of codeword

Behavior

This function verifies the CRC codeword.

Testing:

Check if this function verifies the codeword correctly for at least 10 samples.

4. Checksum

Generator: codeword = *Checksum_gen*(dataword, word_size, num_blocks)

Implement a function that generates a checksum codeword for a set of input datawords.

Input:

1. Dataword of any length n (dataword)
2. Size of each separated word (word_size)
3. Number of word blocks used for one set of checksum operation (num_blocks)

Output:

1. A codeword based on Checksum

Behavior:

This function calculates the redundancy bit(s) for the dataword and output the codeword for Checksum.

Testing:

Check if this function encodes the dataword correctly for at least 10 samples.

Checker: validity = *Checksum_check*(codeword, word_size, num_blocks)

Input:

1. codeword
2. Size of each separated word (word_size)
3. Number of word blocks used for one set of checksum operation (num_blocks)

Output:

1. Validity of codeword

Behavior

This function verifies the Checksum codeword.

Testing:

Check if this function verifies the codeword correctly for at least 10 samples.

5. Hamming code

Generator: codeword = *Hamming_gen*(dataword)

Implement a function that generates a codeword for an input dataword.

Input:
1. A dataword

Output:
1. A codeword based on Hamming code

Behavior

This function calculates the redundancy bit(s) for the dataword and output the codeword for Hamming code.

Testing:

Check if this function encodes the dataword correctly for at least 10 samples.

Checker: error_pos = *Hamming_check*(codeword)

Input:
1. codeword

Output:
1. Position of error (in case of a single bit error, return -1 if there is no error found)

Behavior

This function verifies the Hamming codeword and report the location of error for the case of single bit error.

Testing:

Check if this function verifies the codeword correctly for at least 10 samples.

Deliverables:

1. Programming source codes
2. Documents containing the following content
   2.1. How to run your programs
   2.2. Screenshots of outputs from running your programs (testing results)
3. Extra credits:
   3.1. Screenshots of outputs from running checker programs on another group's generator. Provide information which group you collaborate.
   3.2. Clean and easy-to-understand source code with proper indentation and comments.

4. Penalty
    4.1. Copy codes from others and the Internet