Computer Programming for Engineers
Assignment 2 (Hamming Code in Network Transmission)

a) Step 1: Problem Identification and Statement

The Objective is to write the program code for a Hamming code technique, which can be used to detect and correct a single bit error that could occur during network message delivery. The program code consists of 6 functions, namely:

1. Int main()
2. int printmenu()
3. void encodeMessageStream(string)
4. void generateHammingCode(bool arr[], int SIZE)
5. void decodeMessageStream(string)
6. int checkMessageParity(bool arr1[], int SIZE)

It also makes use of two bitstream text files, one for the encodeMessageStream(string) function, and the other for decodeMessageStream(string) function. The encodeMessageStream(string) includes a function call for the generateHammingCode(bool arr[], int SIZE) function that accepts the message array and the size as parameters and updates the message array after calculating the parity bits.The decodeMessageStream(string) includes a function call for the checkMessageParity(bool arr1[], int SIZE) function that accepts the encodedmessage array and the size as parameters and updates the message array after calculating the parity bits again on the encoded message and after verifying the encoded messages in the text file. Write a program code that takes input from the user for the file name to be either encoded or decoded and outputs the encoded messages to a new text file "encodedmessages.txt" or outputs the decoded messages to a new text file "decodedmessages.txt".

b) Step 2: Gathering Information and I/O Description

In order to calculate the parity bits for encoding we use the information provided along with the input from the user and gather all information to cover up for any missing information in one place.

- User Inputs and Constants
  - The selection from the menu: userselection
  - Name of the File to be Encoded: FileToEncode
  - Name of the File to be Decoded: FileToDecode
  - Size of the Message which equals the size of the array: size = 7 (as there are 7 bits in the message but it is important to note that the array range would be from 0-6 for C++)

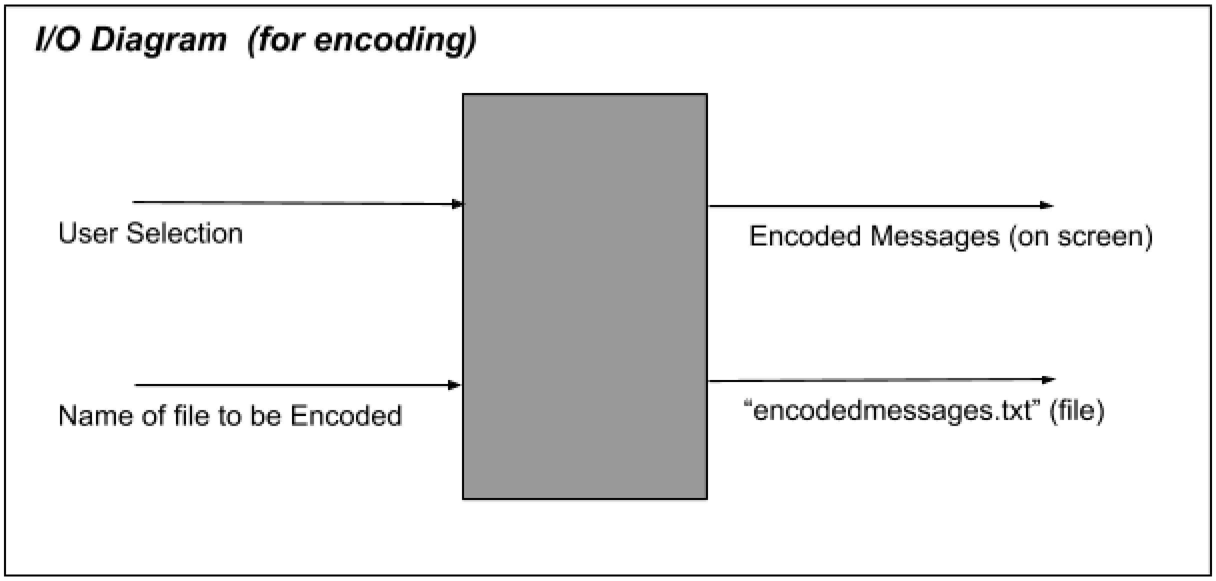
- Outputs
  - A file containing the encoded messages: "encodedmessages.txt"
  - A file containing the decoded messages: "decodedmessages.txt"
  - Output on the screen to show Encoded Messages
  - Output on the screen to show Decoded Messages in the format:
    - <encoded_message>|<corrected_message>|<original_message>|<message_status>

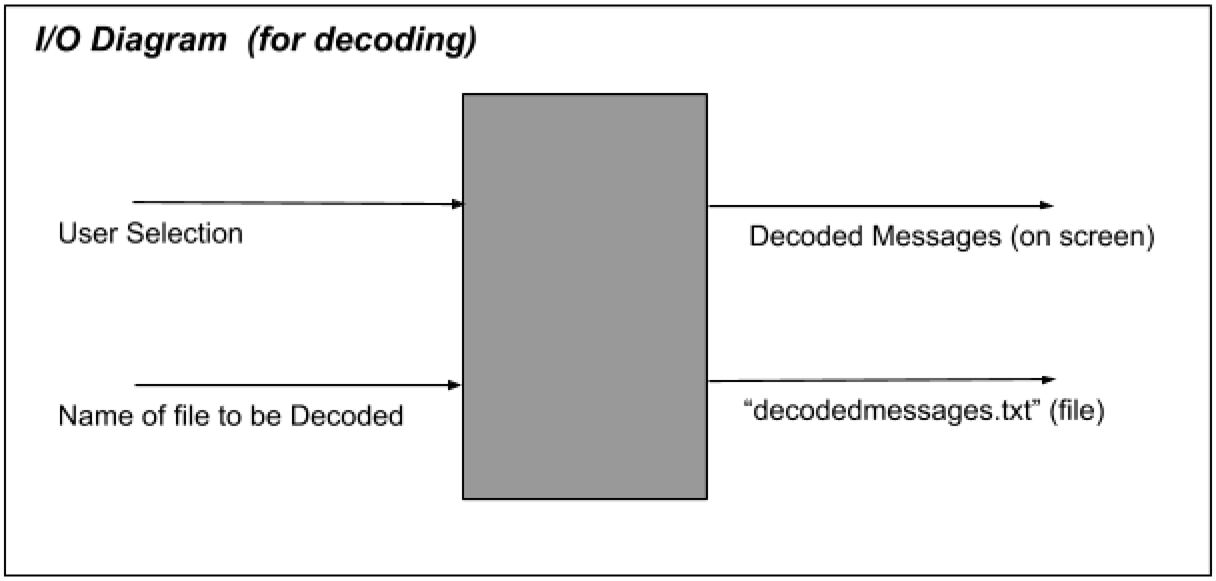
Next Page --->

- Parity Bit Calculation: (For this assignment we will be using the even parity Hamming code generation)
    - For the given set of 7 bits, if the number of 1's is even, the parity bit is set to '0', and set to '1' if the number of 1's is odd.
    - The Original Message consists of 4 bits, therefore to make it up to 7 bits, we will be adding 3 parity bits (p1,p2,p4)
    - p1, p2 and p4 are the parity bits placed at bit locations corresponding to the
    - power of 2 (1, 2 and 4) which would actually correspond to the message[6], message[5], message[3] positions in the array where the message array could be any array that is to be encoded.
    - To calculate the parity bits, we can use the XOR operation as <u>p1 = m1 *XOR* m2 *XOR* m3</u> where m1, m2, m3 are the original parity bits concerned with the specific parity bit and would be different for each parity bit to be calculated. In C++, we use ^ instead of XOR.
    - Parity bit p1 covers all bit positions which have first least significant bit set
    - Parity bit p2 covers all bit positions which have the second least significant bit set
    - Parity bit p4 covers all bit positions which have the third least significant bit set
    - Therefore the formulas to calculate the Parity Bits on the original message would be:
        - p1 = arr[0] XOR arr[2] XOR arr[3]
        - p2 = arr[0] XOR arr[1] XOR arr[3]
        - p3 = arr[0] XOR arr[1] XOR arr[2]
    - And the formulas to calculate the Parity Bits on the encoded message for decoding would be:
        - p1 = arr1[0] XOR arr1[2] XOR arr1[4] XOR arr1[6]
        - p2 = arr1[0] XOR arr1[1] XOR arr1[4] XOR arr1[5]
        - p3 = arr1[0] XOR arr1[1] XOR arr1[2] XOR arr1[3]

Next Page ---->

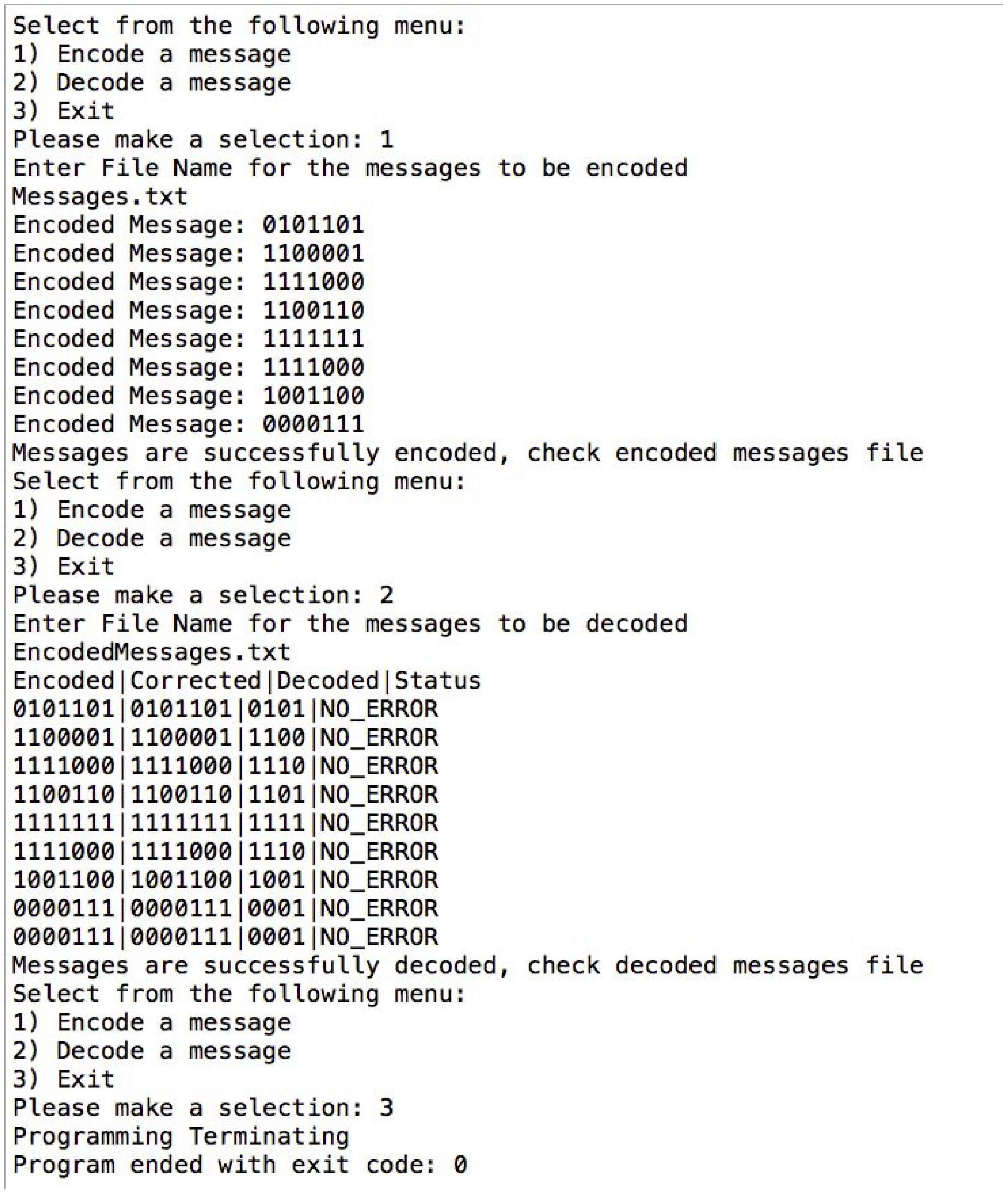
The I/O diagrams for this problem are illustrated below:


I/O Diagram (for encoding)
User Selection
Name of file to be Encoded
Encoded Messages (on screen)
"encodedmessages.txt" (file)


I/O Diagram (for decoding)
User Selection
Name of file to be Decoded
Decoded Messages (on screen)
"decodedmessages.txt" (file)

Next Page --->

The Output message with the relevant Input and Output information will be shown in a similar pattern to this:



```
Select from the following menu:
1) Encode a message
2) Decode a message
3) Exit
Please make a selection: 1
Enter File Name for the messages to be encoded
Messages.txt
Encoded Message: 0101101
Encoded Message: 1100001
Encoded Message: 1111000
Encoded Message: 1100110
Encoded Message: 1111111
Encoded Message: 1111000
Encoded Message: 1001100
Encoded Message: 0000111
Messages are successfully encoded, check encoded messages file
Select from the following menu:
1) Encode a message
2) Decode a message
3) Exit
Please make a selection: 2
Enter File Name for the messages to be decoded
EncodedMessages.txt
Encoded|Corrected|Decoded|Status
0101101|0101101|0101|NO_ERROR
1100001|1100001|1100|NO_ERROR
1111000|1111000|1110|NO_ERROR
1100110|1100110|1101|NO_ERROR
1111111|1111111|1111|NO_ERROR
1111000|1111000|1110|NO_ERROR
1001100|1001100|1001|NO_ERROR
0000111|0000111|0001|NO_ERROR
0000111|0000111|0001|NO_ERROR
Messages are successfully decoded, check decoded messages file
Select from the following menu:
1) Encode a message
2) Decode a message
3) Exit
Please make a selection: 3
Programming Terminating
Program ended with exit code: 0
```

Next Page ---->

c) Step 3: Test Cases and algorithm

<u>- Test Cases;</u>

Test cases use sample input data that is used in manual calculations by the user to produce expected output data in order to verify the functionality of the program.For Test Cases, different ranges of data are used as follows;

- **1) Sample Messages for Encoding: (Make a text file with the following data)**
  - 0101
  - 1100
  - 1110
  - 1101
  - 1111
  - 1110
  - 1001
  - 0001

The text file would contain this data and after entering the name of this text file which would be stored as “SampleMessages.txt”, the program should store the following data in the file “encodedmessages.txt” as well as the output screen:

The working is as follows:

- 0101:

Would be stored in the array and after allowing for parity bits as blanks, the array should be: 010_1_ _.

- p1 = 1, since number of 1’s at locations 3, 5 and 7 are odd.
- p2 = 0, since number of 1’s at locations 3, 6 and 7 are even.
- p4 = 1, since number of 1’s at locations 5, 6, and 7 are odd.

Hence the encoded message would be 0101101

- 1100:

Would be stored in the array and after allowing for parity bits as blanks, the array should be: 110_0_ _.

- p1 = 1, since number of 1’s at locations 3, 5 and 7 are odd.
- p2 = 0, since number of 1’s at locations 3, 6 and 7 are even.
- p4 = 0, since number of 1’s at locations 5, 6, and 7 are even.

Hence the encoded message would be 1100001

- 1110:

Would be stored in the array and after allowing for parity bits as blanks, the array should be: 111_0_ _.

- p1 = 0, since number of 1’s at locations 3, 5 and 7 are even.
- p2 = 0, since number of 1’s at locations 3, 6 and 7 are even.
- p4 = 1, since number of 1’s at locations 5, 6, and 7 are odd.

Hence the encoded message would be 1111000

Next Page ---->

- 1101:
  Would be stored in the array and after allowing for parity bits as blanks, the array should be:
110_1_ _.
    - p1 = 0, since number of 1's at locations 3, 5 and 7 are even.
    - p2 = 1, since number of 1's at locations 3, 6 and 7 are odd.
    - p4 = 0, since number of 1's at locations 5, 6, and 7 are even.

  Hence the encoded message would be 1100110

- 1111:
  Would be stored in the array and after allowing for parity bits as blanks, the array should be:
111_1_ _.
    - p1 = 1, since number of 1's at locations 3, 5 and 7 are odd.
    - p2 = 1, since number of 1's at locations 3, 6 and 7 are odd.
    - p4 = 1, since number of 1's at locations 5, 6, and 7 are odd.

  Hence the encoded message would be 1111111

- 1110:
  Would be stored in the array and after allowing for parity bits as blanks, the array should be:
111_0_ _.
    - p1 = 0, since number of 1's at locations 3, 5 and 7 are even.
    - p2 = 0, since number of 1's at locations 3, 6 and 7 are even.
    - p4 = 1, since number of 1's at locations 5, 6, and 7 are odd.

  Hence the encoded message would be 1111000

- 1001:
  Would be stored in the array and after allowing for parity bits as blanks, the array should be:
100_1_ _.
    - p1 = 0, since number of 1's at locations 3, 5 and 7 are even.
    - p2 = 0, since number of 1's at locations 3, 6 and 7 are even.
    - p4 = 1, since number of 1's at locations 5, 6, and 7 are odd.

  Hence the encoded message would be 1001100

- 0001:
  Would be stored in the array and after allowing for parity bits as blanks, the array should be:
000_1_ _.
    - p1 = 1, since number of 1's at locations 3, 5 and 7 are odd.
    - p2 = 1, since number of 1's at locations 3, 6 and 7 are odd.
    - p4 = 0, since number of 1's at locations 5, 6, and 7 are even.

  Hence the encoded message would be 0000111

There the Output stored in the "encodedmessages.txt" should be:

```
0101101110000111111000110011011111111111000100110000000111
```

Next Page ---->

- **2) File with No Errors for Decoding: (Make a text file with the following data)**
    - 01011011100001111100011001101111111111000100110000000111

  This data should be read and stored in the message arrays in the following manner:
    - 0101101
    - 1100001
    - 1111000
    - 1100110
    - 1111111
    - 1111000
    - 1001100
    - 0000111

  The text file would contain this data and after entering the name of this text file which would be stored as “NoErrors.txt”, the program should store the following data in the file “decodedmessages.txt” as well as the output screen:

  The working is as follows:

- 0101101:
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:
  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.
  Therefore, following the format, the output should be:
  ```
  0101101|0101101|0101|NO_ERROR
  ```

- 1100001
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:
  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.
  Therefore, following the format, the output should be:
  ```
  1100001|1100001|1100|NO_ERROR
  ```

- 1111000
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:
  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.
  Therefore, following the format, the output should be:
  ```
  1111000|1111000|1110|NO_ERROR
  ```

Next Page ---->

- 1100110
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  1100110|1100110|1101|NO_ERROR

- 1111111
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  1111111|1111111|1111|NO_ERROR

- 1111000
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  1111000|1111000|1110|NO_ERROR

- 1001100
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  1001100|1001100|1001|NO_ERROR

Next Page ---->

- 0000111
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  ```
  0000111|0000111|0001|NO_ERROR
  ```

  The final output would be:

  ```
  0101101|0101101|0101|NO_ERROR
  1100001|1100001|1100|NO_ERROR
  1111000|1111000|1110|NO_ERROR
  1100110|1100110|1101|NO_ERROR
  1111111|1111111|1111|NO_ERROR
  1111000|1111000|1110|NO_ERROR
  1001100|1001100|1001|NO_ERROR
  0000111|0000111|0001|NO_ERROR
  ```

- **3) File with One Error for Decoding: (Make a text file with the following data)**
    - 01011011100001111100011001101110111111000100110000000111

  This data should be read and stored in the message arrays in the following manner:
    - 0101101
    - 1100001
    - 1111000
    - 1100110
    - 1110111
    - 1111000
    - 1001100
    - 0000111

The text file would contain this data and after entering the name of this text file which would be stored as “OneError.txt”, the program should store the following data in the file “decodedmessages.txt” as well as the output screen:

The working is as follows:

- 0101101:
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  ```
  0101101|0101101|0101|NO_ERROR
  ```

Next Page ---->

- 1100001
  This would be stored in the array
  - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
  - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
  - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  1100001|1100001|1100|NO_ERROR

- 1111000
  This would be stored in the array
  - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
  - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
  - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  1111000|1111000|1110|NO_ERROR

- 1100110
  This would be stored in the array
  - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
  - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
  - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:

  1100110|1100110|1101|NO_ERROR

- 1110111
  This would be stored in the array
  - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
  - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
  - p4 = 1, since number of 1’s at locations 4, 5, 6, and 7 are odd.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 4 = 4
  The bit at message[3] would be flipped to correct it.

  So the final actual message can be extracted as 111_1__

  Therefore, following the format, the output should be:

  1110111|1111111|1111|ERROR_4

Next Page ---->

- 1111000
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:
```
1111000|1111000|1110|NO_ERROR
```

- 1001100
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:
```
1001100|1001100|1001|NO_ERROR
```

- 0000111
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 0 + 0 = 0
  Errorposition should be assigned (stored) -1.

  Therefore, following the format, the output should be:
```
0000111|0000111|0001|NO_ERROR
```

The final output would be:
```
0101101|0101101|0101|NO_ERROR
1100001|1100001|1100|NO_ERROR
1111000|1111000|1110|NO_ERROR
1100110|1100110|1101|NO_ERROR
1110111|1111111|1111|ERROR_4
1111000|1111000|1110|NO_ERROR
1001100|1001100|1001|NO_ERROR
0000111|0000111|0001|NO_ERROR
```

Next Page ---->

- **4) File with All Errors for Decoding: (Make a text file with the following data)**
    - 11011011000001101100011101101011111111001001110000110

  This data should be read and stored in the message arrays in the following manner:
    - 1101101
    - 1000001
    - 1011000
    - 1110110
    - 1011111
    - 1111100
    - 1001110
    - 0000110

  The text file would contain this data and after entering the name of this text file which would be stored as “AllErrors.txt”, the program should store the following data in the file “decodedmessages.txt” as well as the output screen:

  The working is as follows:
- 1101101:

  This would be stored in the array
    - p1 = 1, since number of 1’s at locations 1, 3, 5 and 7 are odd.
    - p2 = 1, since number of 1’s at locations 2, 3, 6 and 7 are odd.
    - p4 = 1, since number of 1’s at locations 4, 5, 6, and 7 are odd.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 1 + 2 + 4 = 7

  The bit at message [0] would be flipped to correct it.

  So the final actual message can be extracted as 010_1__

  Therefore, following the format, the output should be:

  1101101|0101101|0101|ERROR_7

- 1000001

  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 1, since number of 1’s at locations 2, 3, 6 and 7 are odd.
    - p4 = 1, since number of 1’s at locations 4, 5, 6, and 7 are odd.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 2 + 4 = 6

  The bit at message[1] would be flipped to correct it.

  So the final actual message can be extracted as 110_0__

  Therefore, following the format, the output should be:

  1000001|1100001|1100|ERROR_6

Next Page ---->

- 1011000
  This would be stored in the array
  - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
  - p2 = 1, since number of 1’s at locations 2, 3, 6 and 7 are odd.
  - p4 = 1, since number of 1’s at locations 4, 5, 6, and 7 are odd.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 2 + 4 = 6

  The bit at message[1] would be flipped to correct it.

  So the final actual message can be extracted as 111_0__

  Therefore, following the format, the output should be:

  1011000|1111000|1110|ERROR_6

- 1110110
  This would be stored in the array
  - p1 = 1, since number of 1’s at locations 1, 3, 5 and 7 are odd.
  - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
  - p4 = 1, since number of 1’s at locations 4, 5, 6, and 7 are odd.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 1 + 0 + 4 = 5

  The bit at message[2] would be flipped to correct it.

  So the final actual message can be extracted as 110_1__

  Therefore, following the format, the output should be:

  1110110|1100110|1101|ERROR_5

- 1011111
  This would be stored in the array
  - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
  - p2 = 1, since number of 1’s at locations 2, 3, 6 and 7 are odd.
  - p4 = 1, since number of 1’s at locations 4, 5, 6, and 7 are odd.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 2 + 4 = 6

  The bit at message[1] would be flipped to correct it.

  So the final actual message can be extracted as 111_1__

  Therefore, following the format, the output should be:

  1011111|1111111|1111|ERROR_6

- 1111100
  This would be stored in the array
  - p1 = 1, since number of 1’s at locations 1, 3, 5 and 7 are odd.
  - p2 = 1, since number of 1’s at locations 2, 3, 6 and 7 are odd.
  - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 1 + 2 + 0 = 3

  The bit at message[4] would be flipped to correct it.

  So the final actual message can be extracted as 111_0__

  Therefore, following the format, the output should be:

  1111100|1111000|1110|ERROR_3

Next Page ---->

- 1001110
  This would be stored in the array
    - p1 = 0, since number of 1’s at locations 1, 3, 5 and 7 are even.
    - p2 = 1, since number of 1’s at locations 2, 3, 6 and 7 are odd.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 0 + 2 + 0 = 2
  The bit at message[5] would be flipped to correct it.

  So the final actual message can be extracted as 100_1__
  Therefore, following the format, the output should be:

  1001110|1001100|1001|ERROR_2

- 0000110
  This would be stored in the array
    - p1 = 1, since number of 1’s at locations 1, 3, 5 and 7 are odd.
    - p2 = 0, since number of 1’s at locations 2, 3, 6 and 7 are even.
    - p4 = 0, since number of 1’s at locations 4, 5, 6, and 7 are even.

  Hence the encoded error position can be calculated as:

  Errorposition = (1 * p1) + (2 * p2) + (4 * p4) = 1 + 0 + 0 = 1
  The bit at message[4] would be flipped to correct it.

  So the final actual message can be extracted as 000_1__
  Therefore, following the format, the output should be:

  0000110|0000111|0001|ERROR_1

The final output would be:

```
1101101|0101101|0101|ERROR_7
1000001|1100001|1100|ERROR_6
1011000|1111000|1110|ERROR_6
1110110|1100110|1101|ERROR_5
1011111|1111111|1111|ERROR_6
1111100|1111000|1110|ERROR_3
1001110|1001100|1001|ERROR_2
0000110|0000111|0001|ERROR_1
```

Next Page ---->

- Algorithm;

```
Main() Function
    Assign printmenu() to userselection
    If userselection is equal to 1
        Print  "Enter File Name for the messages to be encoded", newline
        Read value into FileToEncode
        encodeMessageStream (FileToEncode)
    Otherwise If userselection is equal to 2
        Print  "Enter File Name for the messages to be decoded", newline
        Read value into FileToDecode
        encodeMessageStream (FileToDecode)
    Otherwise
        Exit program
    Return 0

printmenu() Function
        Print "Select from the following menu: " , newline;
        Print "1) Encode a message " , newline;
        Print "2) Decode a message " , newline;
        Print "3) Exit " , newline;
        Repeat
            Print "Please make a selection", newline
            Read value into USERSELECTION
            If USERSELECTION is less than 1 OR greater than 3
                Print "Invalid Selection." , newline
        While USERSELECTION is less than 1 OR greater than 3
        Return USERSELECTION

encodeMessageStream(EncodeFile) Function
    Assign 7 to size

    Open file EncodeFile for reading as file1
    If cannot open EncodeFile
        Print Error “File failed to open”
        Exit program

    Open file "encodedMessages.txt" for writing as file2
    If cannot open "encodedMessages.txt"
        Print Error “Error creating the file”
        Exit program
```

Next Page ---->

```
        Repeat while it is not end of file
                Assign 0 to i
                Repeat while i is less than 4
                        Get character from file1
                        Assign character to ch
                        Assign (ch - 48) to message[i]
                        Increment i

                generateHammingCode(message, size)

                Assign 0 to i
                Repeat while i is less than size
                        Write message[i] into file1
                        Increment i

                Print “Encoded Message: ”, message[0], message[1], message[2], message[3],
                message[4], message[5], message[6], newline

        Close file1
        Close file2
        Print "Messages are successfully encoded, check encoded messages file ", newline

generateHammingCode(arr[], SIZE) function
        Assign arr[0] XOR arr[2] XOR arr[3] to p1
        Assign arr[0] XOR arr[1] XOR arr[3] to p2
        Assign arr[0] XOR arr[1] XOR arr[2] to p3

        Assign arr[3] to arr[4]
        Assign p4 to arr[3]
        Assign p2 to arr[5]
        Assign p1 to arr[6]


decodeMessageStream(DecodeFile) Function
        Assign 7 to size

        Open file DecodeFile for reading as file1
        If cannot open DecodeFile
                Print Error “File failed to open”
                Exit program

        Open file "decodedMessages.txt" for writing as file2
        If cannot open "decodedMessages.txt"
                Print Error “Error creating the file”
                Exit program
```

Next Page ---->

```
Repeat while it is not end of file
        Assign 0 to i
        Repeat while i is less than size
                Get character from file1
                Assign character to ch
                Assign (ch - 48) to x[i]
                Assign (ch - 48) to encodedmessage[i]
                Increment i
        If ch is not equal to lastChar
                Assign checkMessageParity(encodedmessage, size) to errorposition
                If errorposition is equal to -1
                        Write x[0] , x[1] , x[2] , x[3] , x[4] , x[5] , x[6] , "|" ,encodedmessage[0] ,
                        encodedmessage[1] , encodedmessage[2] , encodedmessage[3],
                        encodedmessage[4] , encodedmessage[5] , encodedmessage[6] , "|"
                        ,encodedmessage[0] , encodedmessage[1] , encodedmessage[2] ,
                        encodedmessage[4], "|" ,"NO_ERROR" into file2

                        Print , x[0] , x[1] , x[2] , x[3] , x[4] , x[5] , x[6] , "|" ,encodedmessage[0] ,
                        encodedmessage[1] , encodedmessage[2] , encodedmessage[3],
                        encodedmessage[4] , encodedmessage[5] , encodedmessage[6] , "|"
                        ,encodedmessage[0] , encodedmessage[1] , encodedmessage[2] ,
                        encodedmessage[4], "|" ,"NO_ERROR" , newline
                Otherwise
                        Write x[0] , x[1] , x[2] , x[3] , x[4] , x[5] , x[6] , "|" ,encodedmessage[0] ,
                        encodedmessage[1] , encodedmessage[2] , encodedmessage[3],
                        encodedmessage[4] , encodedmessage[5] , encodedmessage[6] , "|"
                        ,encodedmessage[0] , encodedmessage[1] , encodedmessage[2] ,
                        encodedmessage[4], "|" , "ERROR_" , errorposition into file2

                        Print x[0] , x[1] , x[2] , x[3] , x[4] , x[5] , x[6] , "|" ,encodedmessage[0] ,
                        encodedmessage[1] , encodedmessage[2] , encodedmessage[3],
                        encodedmessage[4] , encodedmessage[5] , encodedmessage[6] , "|"
                        ,encodedmessage[0] , encodedmessage[1] , encodedmessage[2] ,
                        encodedmessage[4] , "|" , "ERROR_" , errorposition , newline

                Get character from file1
                Assign character to ch
                If ch is equal to ‘/n’
                        Assign ch to lastChar
Print "Messages are successfully encoded, check encoded messages file ", newline

Close file1
Close file2
```

Next Page --->

```
checkMessageParity (arr1[], SIZE) function

    Assign arr1[0] XOR arr1[2] XOR arr1[4] XOR arr1[6] to p1
    Assign arr1[0] XOR arr1[1] XOR arr1[4] XOR arr1[5] to p2
    Assign arr1[0] XOR arr1[1] XOR arr1[2] XOR arr1[3] to p4

    Assign ((1 * p1) + (2 * p2) + (4 * p4)) to errorbit
    If errorbit is not equal to 0
        If arr1[7 - errorbit] is equal to 0
            Assign 1 to arr1[7 - errorbit]
        Otherwise
            Assign 0 to arr1[7 - errorbit]

    If errorbit is equal to 0
        Assign -1 to errorbit

    Return errorbit
```

Next Page ---->

d) Step 4: Code or implementation

```
#include <iostream> // Required for cout, cin, endl
#include <fstream>
#include <string>
#include <cmath>
using namespace std;// The compiler will use all the library filenames in std
 // Function Prototypes
int printmenu();
void encodeMessageStream(string);
void generateHammingCode(bool arr[], int SIZE);
void decodeMessageStream(string);
int checkMessageParity(bool arr1[], int SIZE);
// The main function - the starting point of our program
int main()
{
	 // Declare and initialize objects
	int userselection = 0;
	string FileToEncode;
	string FileToDecode;
	userselection = printmenu();
	if (userselection == 1)
	{	// Print values
		cout << "Enter File Name for the messages to be encoded" << endl;
		cin >> FileToEncode; //User input
		encodeMessageStream(FileToEncode); // Function Call
	}
	else if (userselection == 2)
	{
		cout << "Enter File Name for the messages to be decoded" << endl;
		cin >> FileToDecode; // User input
		decodeMessageStream(FileToDecode); // Function Call
	}
	else
		exit(0);  // Exit Program
	system("pause");
	  // Exit program
	return (0);
}
 // Function for printing the menu
int printmenu()
{
	 // Declare and initialize objects
	int USERSELECTION = 0;
	// Print values
	cout << "Select from the following menu: " << endl;
	cout << "1) Encode a message " << endl;
	cout << "2) Decode a message " << endl;
	cout << "3) Exit " << endl;
```

Next Page --->

```
		// Validating the input from the user
		do
		{
			cout << "Please make a selection: ";
			cin >> USERSELECTION;
			if (USERSELECTION < 1 || USERSELECTION > 3)
				cout << "Invalid Selection " << endl;
		} while ((USERSELECTION < 1 || USERSELECTION > 3));
		return USERSELECTION;
}
 // Function to encode a message
void encodeMessageStream(string EncodeFile)
{
		 // Declare and initialize objects
# define size 7
		bool message[size];
		char ch;
		// Opening the file requested by the user
		ifstream file1;
		file1.open(EncodeFile, ios::in);
		if (file1.fail())
		{
			cerr << "File failed to open" << endl;
			exit(1);
		}
		// Creating the second file required for the user
		ofstream file2;
		file2.open("encodedMessages.txt", ios::out);
		if (file2.fail())
		{
			cerr << "Error creating the file " << endl;
			exit(0);
		}
		// Running the loop until end of file
		while (!file1.eof())
		{
			for (int i = 0; i < 4; i++)// Loop to fill message array
			{
				// Getting one character at a time from the file
				file1.get(ch);
				message[i] = ch - 48;
			}
			generateHammingCode(message, size); // Function call
			for (int i = 0; i < size; i++)
				file2 << message[i]; // Writing values to the file
			cout << "Encoded Message: " << message[0] << message[1] << message[2] <<
				message[3] << message[4] << message[5] << message[6] << endl;
			file1.get(ch);
		}
		file2.close(); // Closing the file
		file1.close(); // Closing the file
		cout << "Messages are successfully encoded, check encoded messages file " << endl;
}
```

Next Page --->

```
void generateHammingCode(bool arr[], int SIZE)
{
        bool p1, p2, p4;
        // Calculating Parity Bits
        p1 = arr[0] ^ arr[2] ^ arr[3];
        p2 = arr[0] ^ arr[1] ^ arr[3];
        p4 = arr[0] ^ arr[1] ^ arr[2];
        // Updating the array
        arr[4] = arr[3];
        arr[3] = p4;
        arr[5] = p2;
        arr[6] = p1;
}

 // Function to decode a message
void decodeMessageStream(string DecodeFile)
{
         // Declare and initialize objects
#define size 7
        bool encodedmessage[size], x[size];
        char ch;
        char lastChar('z');
        int errorposition = 0;
        // Opening the file requested by the user
        ifstream file1;
        file1.open(DecodeFile, ios::in);
        if (file1.fail())
        {
                cerr << "File failed to open" << endl;
                exit(1);
        }
        // Creating the second file required for the user
        ofstream file2;
        file2.open("decodedMessages.txt", ios::out);
        if (file2.fail())
        {
                cerr << "Error creating the file " << endl;
                exit(0);
        }
        // Running the loop until end of file
        while (!file1.eof())
        {
                for (int i = 0; i < size; i++)// Loop to fill message array
                {       // Getting one character at a time from the file
                        file1.get(ch);
                        x[i] = ch - 48;
                        encodedmessage[i] = ch - 48;
                }
                // If statement to account for any spaces in the next line
                if (ch != lastChar) {
                        errorposition = checkMessageParity(encodedmessage, size); // Function call
                        if (errorposition == -1)
```

Next Page ---->

```
                        {       // Writing values to the file
                                file2 << x[0] << x[1] << x[2] << x[3] << x[4] << x[5] << x[6] << "|"
<<encodedmessage[0] << encodedmessage[1] << encodedmessage[2] << encodedmessage[3]
<< encodedmessage[4] << encodedmessage[5] << encodedmessage[6] << "|" <<encodedmessage[0] <<
encodedmessage[1] << encodedmessage[2] << encodedmessage[4]<< "|" <<"NO_ERROR" <<endl;
                                // Printing Values to the screen
                                cout << x[0] << x[1] << x[2] << x[3] << x[4] << x[5] << x[6] << "|"
<<encodedmessage[0] << encodedmessage[1] << encodedmessage[2] << encodedmessage[3]
<< encodedmessage[4] << encodedmessage[5] << encodedmessage[6] << "|" <<encodedmessage[0] <<
encodedmessage[1] << encodedmessage[2] << encodedmessage[4]<< "|" <<"NO_ERROR" <<endl;
                        }
                        else
                        {       // Writing values to the file
                                file2 << x[0] << x[1] << x[2] << x[3] << x[4] << x[5] << x[6] << "|"
<<encodedmessage[0] << encodedmessage[1] << encodedmessage[2] << encodedmessage[3]
<< encodedmessage[4] << encodedmessage[5] << encodedmessage[6] << "|" <<encodedmessage[0] <<
encodedmessage[1] << encodedmessage[2] << encodedmessage[4]<< "|" << "ERROR_" << errorposition <<
endl;
                                // Printing Values to the screen
                                cout << x[0] << x[1] << x[2] << x[3] << x[4] << x[5] << x[6] << "|"
<<encodedmessage[0] << encodedmessage[1] << encodedmessage[2] << encodedmessage[3]
<< encodedmessage[4] << encodedmessage[5] << encodedmessage[6] << "|" <<encodedmessage[0] <<
encodedmessage[1] << encodedmessage[2] << encodedmessage[4] << "|" << "ERROR_" << errorposition <<
endl;
                        }

                        file1.get(ch);
                        if (ch == '\n') {
                                lastChar = ch; // Checking for the line to end
                        }
                }
        }
        cout << "Messages are successfully decoded, check decoded messages file " << endl;
        file2.close(); // Closing the file
        file1.close();// Closing the file
}

int checkMessageParity(bool arr1[], int SIZE)
{
        int errorbit = 0, x = 0;
        bool p1, p2, p4;
        // Calculating Parity Bits
        p1 = arr1[0] ^ arr1[2] ^ arr1[4] ^ arr1[6];
        p2 = arr1[0] ^ arr1[1] ^ arr1[4] ^ arr1[5];
        p4 = arr1[0] ^ arr1[1] ^ arr1[2] ^ arr1[3];
        errorbit = (1 * p1) + (2 * p2) + (4 * p4);
        // Updating the array
        if (errorbit != 0) {
                if (arr1[7 - errorbit] == 0)
                        arr1[7 - errorbit] = 1;
                else
                        arr1[7 - errorbit] = 0;

        }
```

Next Page --->

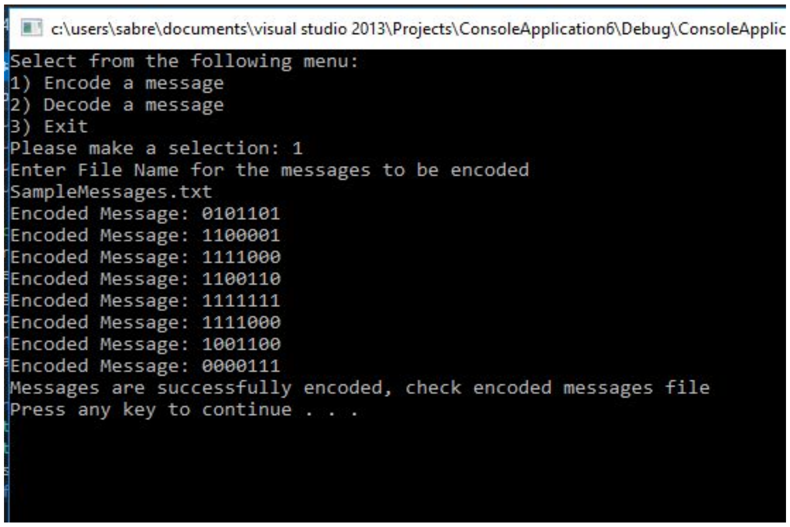
```
		if (errorbit == 0)
			errorbit = -1;
		return errorbit;
	}
```

Next Page ---->

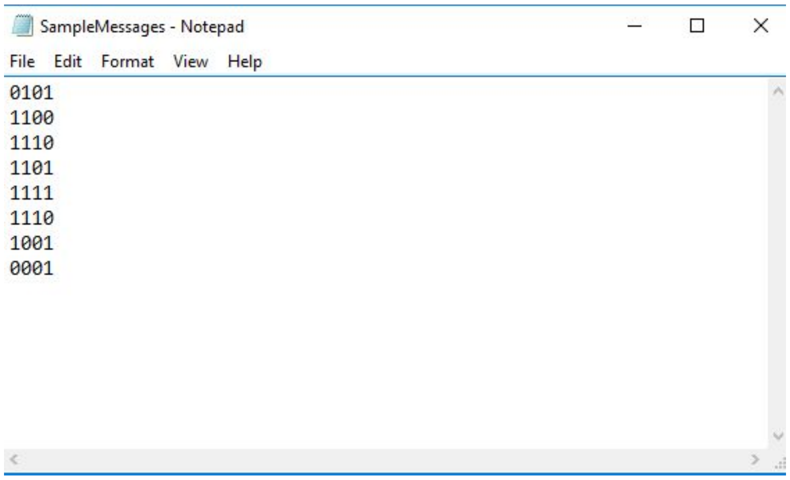
e) Step 5: Test and Verification

This step ensures if the final source code contains any errors or not. All the test cases in Step 3 are tried by running the program and the expected outcomes are compared with the actual outcomes.

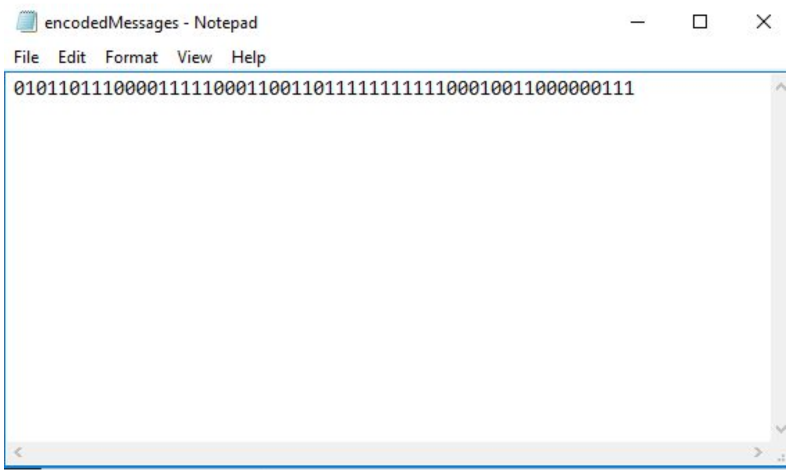
Test Case 1:



```
c:\users\sabre\documents\visual studio 2013\Projects\ConsoleApplication6\Debug\ConsoleApplic
Select from the following menu:
1) Encode a message
2) Decode a message
3) Exit
Please make a selection: 1
Enter File Name for the messages to be encoded
SampleMessages.txt
Encoded Message: 0101101
Encoded Message: 1100001
Encoded Message: 1111000
Encoded Message: 1100110
Encoded Message: 1111111
Encoded Message: 1111000
Encoded Message: 1001100
Encoded Message: 0000111
Messages are successfully encoded, check encoded messages file
Press any key to continue . . .
```



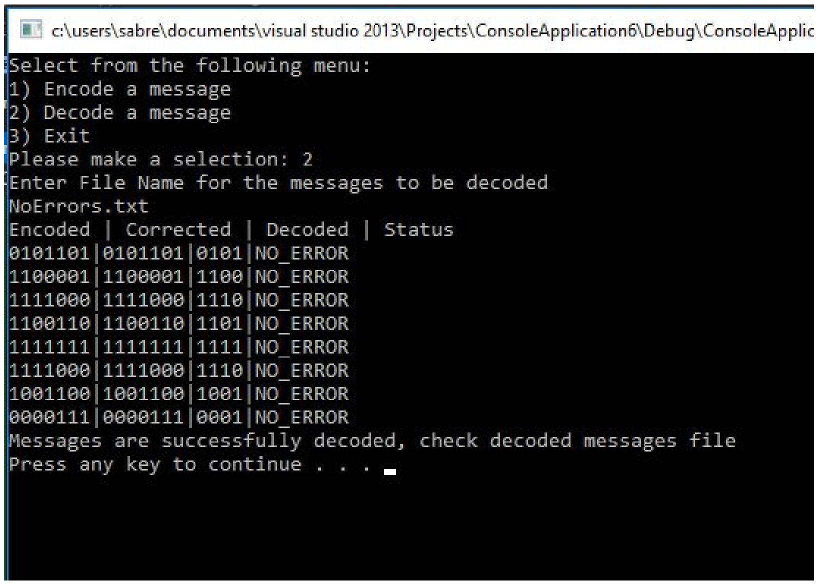
```
SampleMessages - Notepad
0101
1100
1110
1101
1111
1110
1001
0001
```



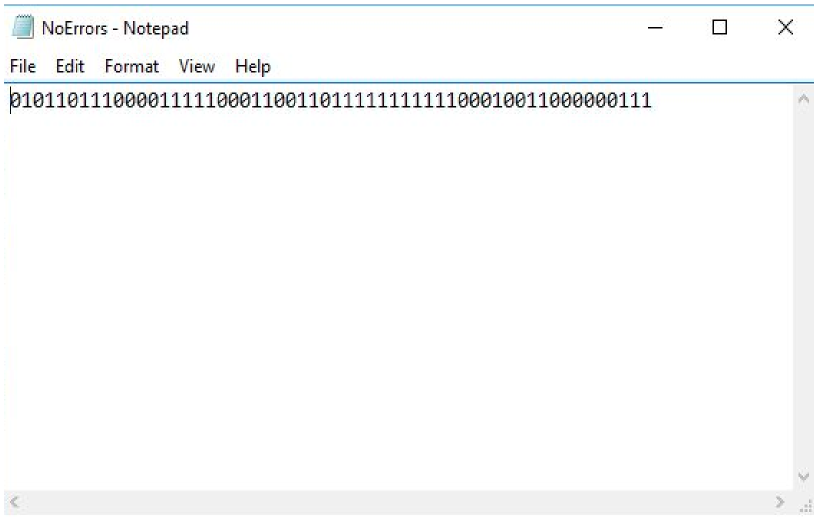
```
encodedMessages - Notepad
01011011100001111100011001101111111111110001001100000111
```

Next Page ---->

Test Case 2:

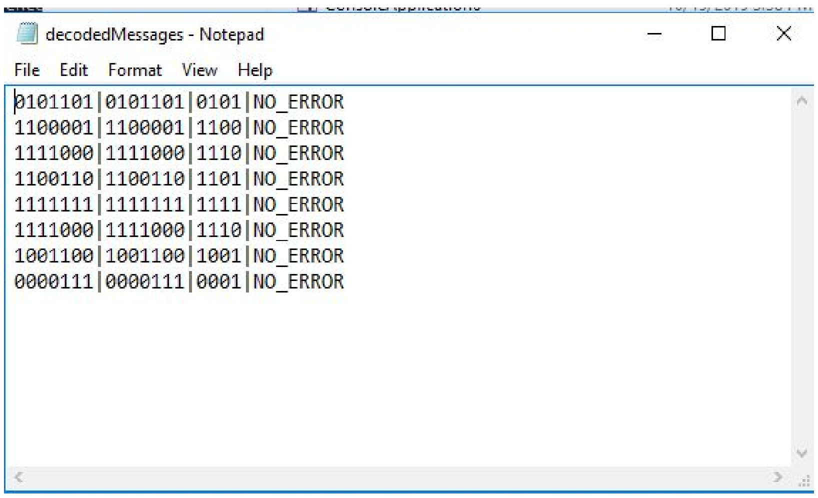


```
c:\users\sabre\documents\visual studio 2013\Projects\ConsoleApplication6\Debug\ConsoleApplic
Select from the following menu:
1) Encode a message
2) Decode a message
3) Exit
Please make a selection: 2
Enter File Name for the messages to be decoded
NoErrors.txt
Encoded | Corrected | Decoded | Status
0101101|0101101|0101|NO_ERROR
1100001|1100001|1100|NO_ERROR
1111000|1111000|1110|NO_ERROR
1100110|1100110|1101|NO_ERROR
1111111|1111111|1111|NO_ERROR
1111000|1111000|1110|NO_ERROR
1001100|1001100|1001|NO_ERROR
0000111|0000111|0001|NO_ERROR
Messages are successfully decoded, check decoded messages file
Press any key to continue . . .
```



NoErrors - Notepad

```
01011011100001111100011001101111111111110001001100000011 1
```

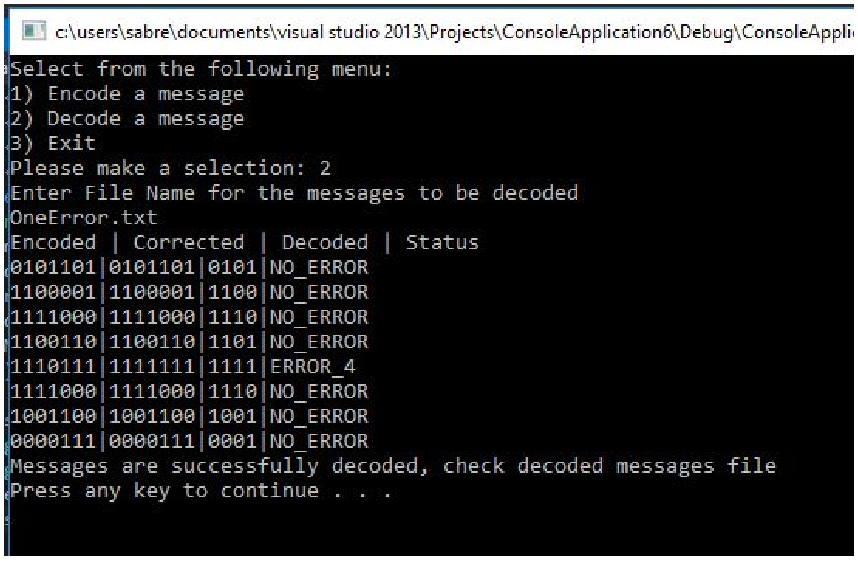


decodedMessages - Notepad

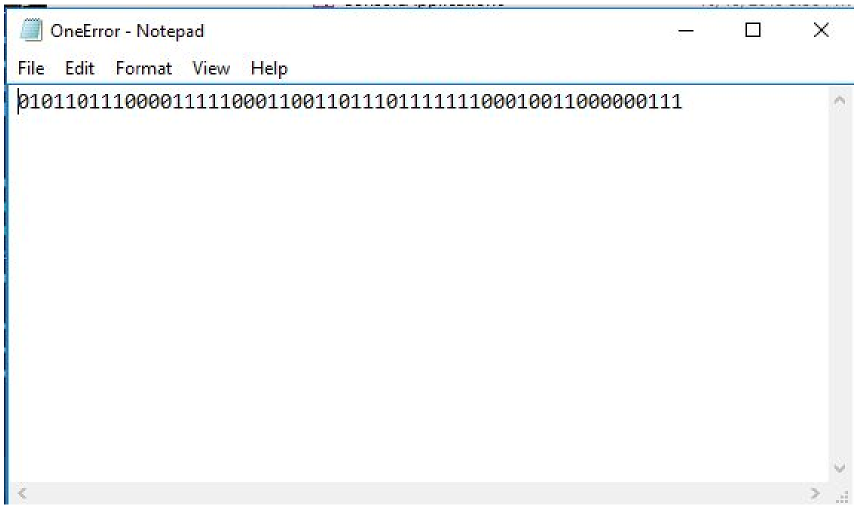
```
0101101|0101101|0101|NO_ERROR
1100001|1100001|1100|NO_ERROR
1111000|1111000|1110|NO_ERROR
1100110|1100110|1101|NO_ERROR
1111111|1111111|1111|NO_ERROR
1111000|1111000|1110|NO_ERROR
1001100|1001100|1001|NO_ERROR
0000111|0000111|0001|NO_ERROR
```

Next Page ---->

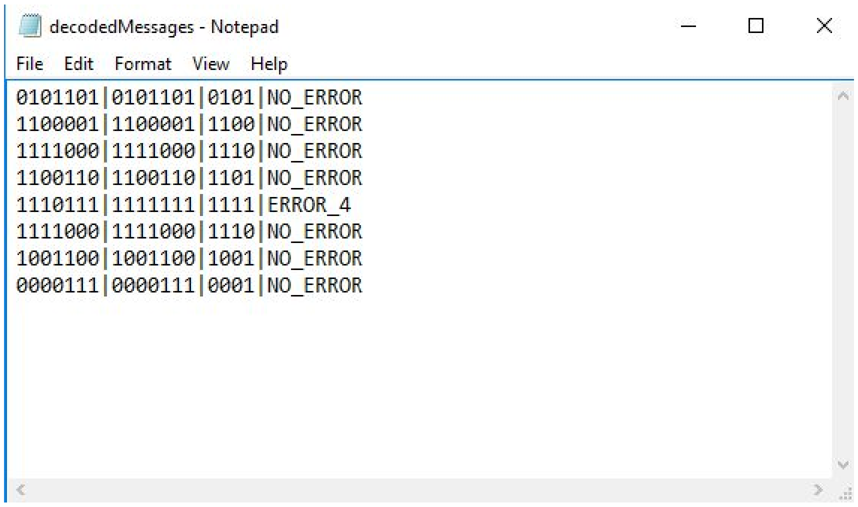
Test Case 3:



```
c:\users\sabre\documents\visual studio 2013\Projects\ConsoleApplication6\Debug\ConsoleAppli
Select from the following menu:
1) Encode a message
2) Decode a message
3) Exit
Please make a selection: 2
Enter File Name for the messages to be decoded
OneError.txt
Encoded | Corrected | Decoded | Status
0101101|0101101|0101|NO_ERROR
1100001|1100001|1100|NO_ERROR
1111000|1111000|1110|NO_ERROR
1100110|1100110|1101|NO_ERROR
1110111|1111111|1111|ERROR_4
1111000|1111000|1110|NO_ERROR
1001100|1001100|1001|NO_ERROR
0000111|0000111|0001|NO_ERROR
Messages are successfully decoded, check decoded messages file
Press any key to continue . . .
```



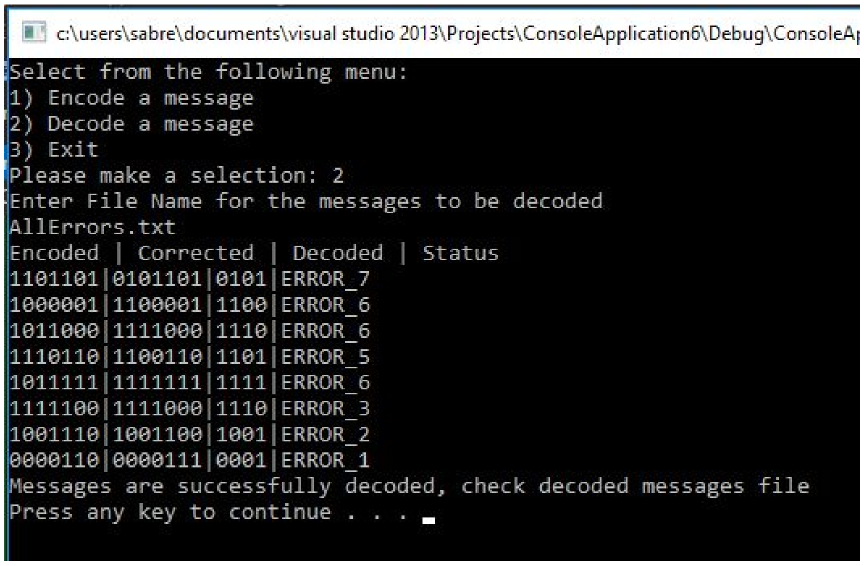
```
01011011100001111100011001101110111111110001001100000111
```



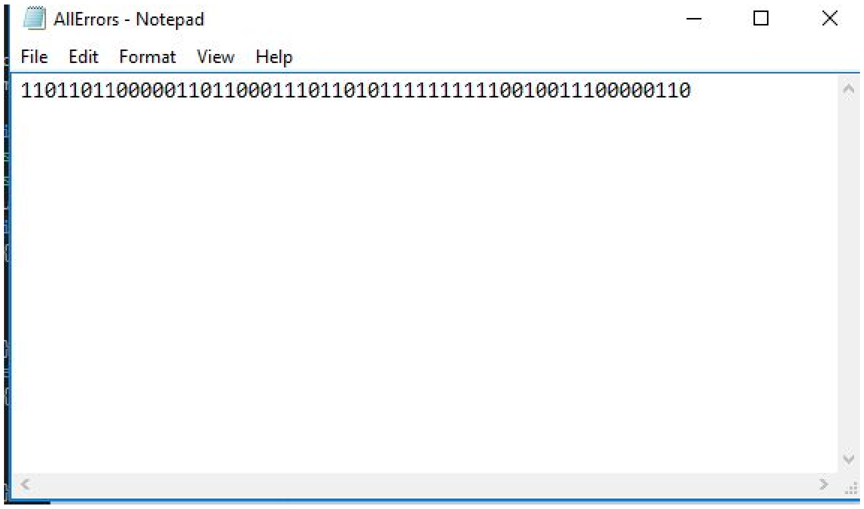
```
0101101|0101101|0101|NO_ERROR
1100001|1100001|1100|NO_ERROR
1111000|1111000|1110|NO_ERROR
1100110|1100110|1101|NO_ERROR
1110111|1111111|1111|ERROR_4
1111000|1111000|1110|NO_ERROR
1001100|1001100|1001|NO_ERROR
0000111|0000111|0001|NO_ERROR
```

Next Page ---->

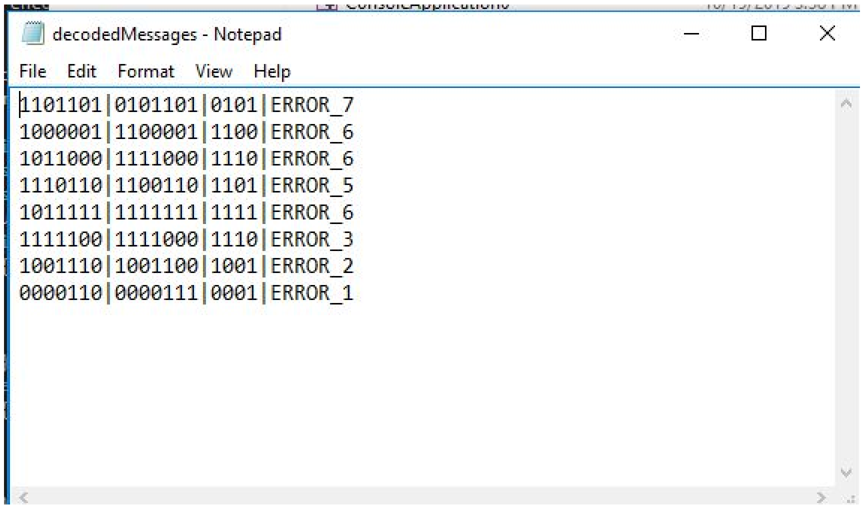
Test Case 4:



```
c:\users\sabre\documents\visual studio 2013\Projects\ConsoleApplication6\Debug\ConsoleAp
Select from the following menu:
1) Encode a message
2) Decode a message
3) Exit
Please make a selection: 2
Enter File Name for the messages to be decoded
AllErrors.txt
Encoded | Corrected | Decoded | Status
1101101|0101101|0101|ERROR_7
1000001|1100001|1100|ERROR_6
1011000|1111000|1110|ERROR_6
1110110|1100110|1101|ERROR_5
1011111|1111111|1111|ERROR_6
1111100|1111000|1110|ERROR_3
1001110|1001100|1001|ERROR_2
0000110|0000111|0001|ERROR_1
Messages are successfully decoded, check decoded messages file
Press any key to continue . . .
```



```
AllErrors - Notepad
File  Edit  Format  View  Help
11011011000001101100011101101011111111110010011100000110
```



```
decodedMessages - Notepad
File  Edit  Format  View  Help
1101101|0101101|0101|ERROR_7
1000001|1100001|1100|ERROR_6
1011000|1111000|1110|ERROR_6
1110110|1100110|1101|ERROR_5
1011111|1111111|1111|ERROR_6
1111100|1111000|1110|ERROR_3
1001110|1001100|1001|ERROR_2
0000110|0000111|0001|ERROR_1
```

Next Page ---->

User Guide

This software can be used by Computer Engineers in verifying if any data is corrupted over transmission. Moreover, the software can also be used to encode any data with even parity before transmission in order to prevent data loss in the case that gets corrupted. It would provide them with the Status of an encoded message whether it is corrupted or not, and then would give them the corrected message as well as the original message before the even parity. The software also stores the results in appropriate files with labels for encoding or decoding so that the results are recorded.

Next Page ---->