1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —

Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 10**

1. «Методы надежного хранения и передачи информации»
2. по дисциплине «Основы информационной безопасности»
3. Выполнил
4. студент гр. 4851001/20001 Козлов О. И.

<*подпись*>

1. Преподаватель
2. профессор Калинин М. О.

<*подпись*>

1. Санкт-Петербург
2. 2023

Оглавление

[1) Цель работы 3](#_Toc130995358)

[2) Постановка задачи 3](#_Toc130995359)

[3) Ход работы 4](#_Toc130995360)

[4) Контрольные вопросы 16](#_Toc130995361)

[5) Выводы 18](#_Toc130995362)

[*Приложение* 19](#_Toc130995363)

## Цель работы

ознакомление с методом Хемминга помехоустойчивого кодирования, позволяющим обнаруживать и автоматически исправлять ошибки, возникающие при хранении и передаче информации в ненадежных средах.

## Постановка задачи

1. Создать текстовый файл размером 100 символов, сохранить его как эталон.

2. Реализовать программу, выполняющую для произвольного текстового файла функции кодирования и декодирования Хемминга. Размер текстового блока, обрабатываемого в рамках одного цикла кодирования, должен задаваться в виде аргумента командной строки.

3. Закодировать текстовый файл. Зафиксировать и сравнить размеры исходного и закодированного файлов для различных размеров кодируемых блоков: 8, 12, 16, 24, 32, 48, 64 разряда.

4. С помощью НЕХ-редактора (например, HIEW) в произвольных местах закодированного файла внести одиночные изменения и зафиксировать позицию в файле и исходное значение измененного бита, чтобы впоследствии можно было проверить данные при декодировании.

5. Запустить разработанную программу на декодирование модифицированного закодированного файла. Сопоставить содержимое декодированного файла с эталоном, отметить совпадения и расхождения. Проверить восстановление измененных битов при декодировании.

6. С помощью НЕХ-редактора в произвольных местах закодированного файла внести групповые изменения. Данные изменения моделируют массовые ошибки при передаче или хранении информации. Для внесенных изменений необходимо зафиксировать позицию в файле и новое значение измененных битов, чтобы впоследствии можно было проверить факт восстановления данных при декодировании файла. Повторить выполнение п. 5.

## Ход работы

*Расчет числа*

Для расчёта кода Хемминга для 16 – го разрядного числа необходимо расписать схему смещения информационных и контрольных бит

|  |  |  |
| --- | --- | --- |
| Номер бит | Разложение номера бита по степеням | Тип бита |
| 1 |  | *Контрольный (контролирует 3, 5, 7, 9, 11, 13, 15)* |
| 2 |  | *Контрольный (контролирует 3, 6, 7, 10, 11, 14, 15)* |
| 3 | =1+2 | Информационный (контролируется битами 1, 2) |
| 4 |  | Контрольный (контролирует 5, 6, 7, 12, 13, 14, 15) |
| 5 | =1+4 | Информационный (контролируется битами 1, 4) |
| 6 | =2+4 | Информационный (контролируется битами 2, 4) |
| 7 | =1+2+4 | Информационный (контролируется битами 1, 2, 4) |
| 8 |  | *Контрольный (контролирует 9, 10, 11, 12, 13, 14, 15)* |
| 9 | =1+8 | Информационный (контролируется битами 1, 8) |
| 10 | =2+8 | Информационный (контролируется битами 2, 8) |
| 11 | =1+2+8 | Информационный (контролируется битами 1, 2, 8) |
| 12 | =4+8 | Информационный (контролируется битами 4, 8) |
| 13 | =1+4+8 | Информационный (контролируется битами 1, 4, 8) |
| 14 | =2+4+8 | Информационный (контролируется битами 2, 4, 8) |
| 15 | =1+2+4+8 | Информационный (контролируется битами 1, 2, 4, 8) |
| 16 |  | *Контрольный (контролирует 17, 18, 19, 20, 21)* |
| 17 | =1+16 | Информационный (контролируется битами 1, 16) |
| 18 | =2+16 | Информационный (контролируется битами 2, 16) |
| 19 | =1+2+16 | Информационный (контролируется битами 1, 2, 16) |
| 20 | =4+16 | Информационный (контролируется битами 4, 16) |
| 21 | =1+4+16 | Информационный (контролируется битами 1, 4, 16) |

*Таблица 1 ­– Схема размещения информационных и контрольных битов*

Возьмем число 1010111011111111.

Для него код Хемминга будет выглядеть следующим образом:

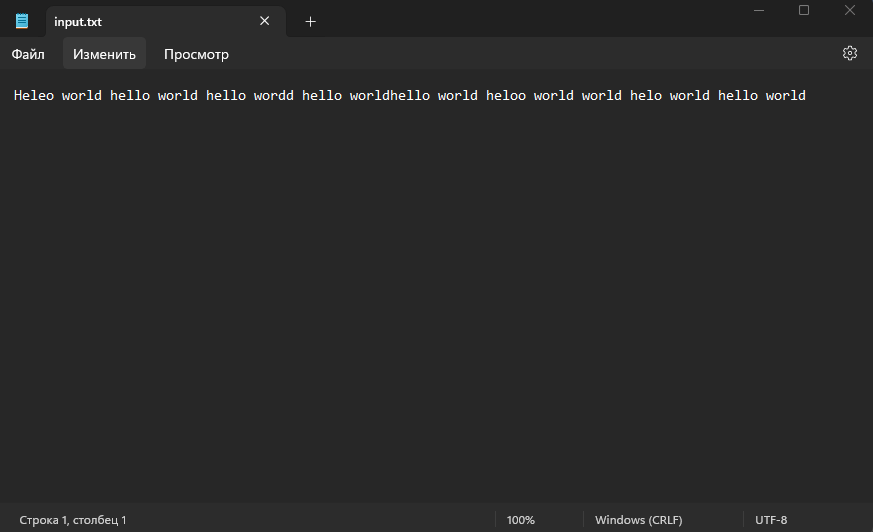
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |

* Для 1 контрольного бита сумма равна: 9. Контрольный бит равен 1.
* Для 2 контрольного бита сумма равна: 9. Контрольный бит равен 1.
* Для 4 контрольного бита сумма равна: 7. Контрольный бит равен 1.
* Для 8 контрольного бита сумма равна: 6. Контрольный бит равен 0.
* Для 16 контрольного бита сумма равна: 5. Контрольный бит равен 1.

Итоговое число: 111101101110111111111.

*Программа кодирования и декодирования*

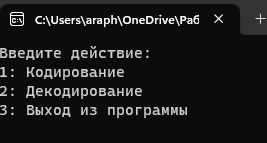
Для реализации программы был создан текстовый файл:

**

*Рисунок 1 – Текстовый файл, используемый в программе*

Далее был реализована программа-кодер и программа-декодер.

Функция Menu () используется для реализации меню и выбора действия.



*Рисунок 2 – Меню в программе*

*Кодирование:*

Кодирование происходит благодаря функции Coder(). У пользователя запрашивается размер кодируемого блока и название файла, который требуется закодировать.

Далее символы из этого текста представляются в двоичном виде (функция TextToBit()) и записывается в дополнительный файл с название Binary.txt. В конце работы программы он удаляется.

Функция AddControlBits() разделяет двоичное представление текста на блоки, размер которых указывает пользователь и с помощью кода Хемминга добавляет контрольные биты. Сначала в дополнительном файле с названием Control.txt. Для начала используется массив с названием NumberBits, где на месте контрольных бит записывается 0, а на места информационных бит считывается числа из текста в бинарном представлении.

Полный размер блока рассчитывается с помощью функции FindOutSize().

Далее для полного блока идет расчет каждого контрольного бита. Так полные блоки считываются до тех пор, пока исходный файл не закончится. Если он закончился, а на полный блок символов не хватило, тогда на место пропуска записывается 0.

Функция BitsToSymbol переводит получившиеся биты в символы. Считывается 8 символов бинарного кода и представляются в символ. А с помощью функции WriteToFile() данные символы записываются в файл с названием Coder.txt. Таким образом исходный текст закодировался в конечный файл. И функция Coder() завершает работу.

*Декодирование:*

Кодирование происходит благодаря функции Decoder(). У пользователя запрашивается размер кодируемого блока и название файла, который требуется декодировать.

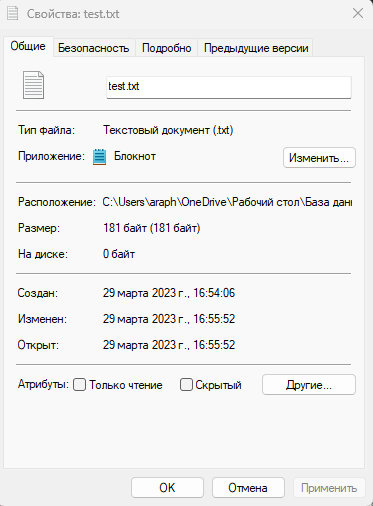
Далее символы из этого текста представляются в двоичном виде (функция TextToBit()) и записывается в дополнительный файл с название Control.txt. В конце работы программы он удаляется.

С помощью функции FindOutSize() находится полный размер блока и записывается в переменную Size. Благодаря функции DeleteControlBits() из файла Control.txt считывается полный блок и проверяются контрольные биты на правильность благодаря функции CheckCorrectness(). Если были допущены ошибки и контрольные биты считаны неправильно, то функция исправляет одиночные ошибки. Контрольные биты удаляются, и оставшиеся биты записываются в файл с названием Binary.txt.

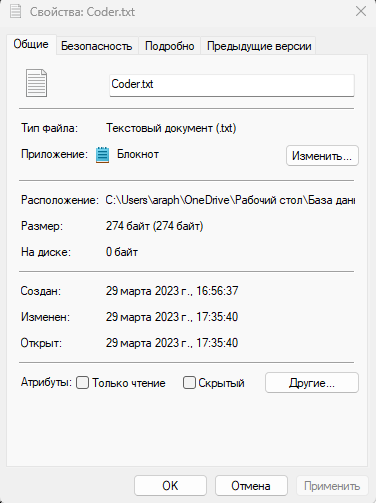
Функция BitsToSymbol переводит получившиеся биты в символы. Считывается 8 символов бинарного кода и представляются в символ. А с помощью функции WriteToFile() данные символы записываются в файл с названием Coder.txt. Таким образом исходный текст закодировался в конечный файл. И функция Decoder() завершает работу.

*Кодирование текстового файла:*

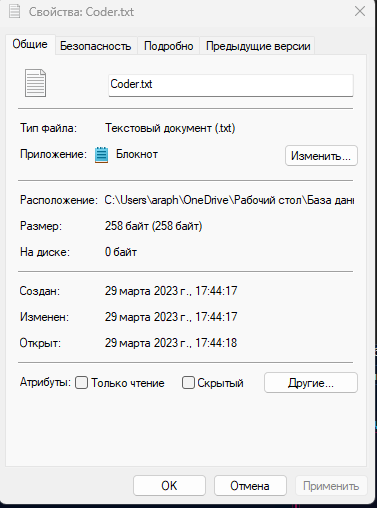
Исходный файл имеет следующий размер:

**

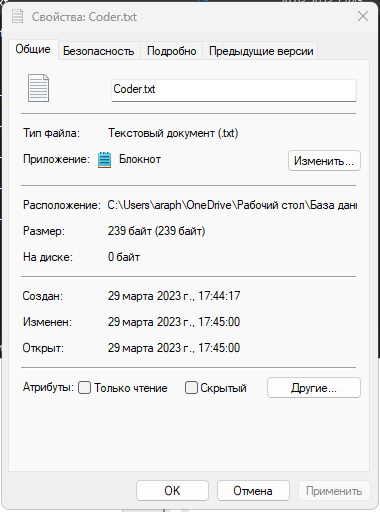
*Рисунок 3 – Начальный размер файла*



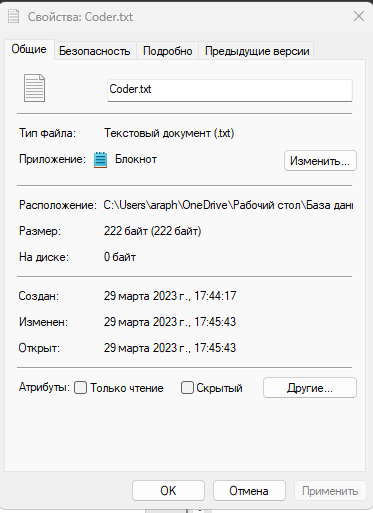
*Рисунок 4 – Размер закодированного файла с размером блока 8*



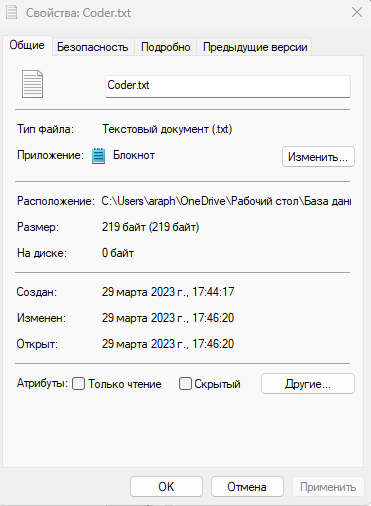
*Рисунок 5 – Размер закодированного файла с размером блока 12*



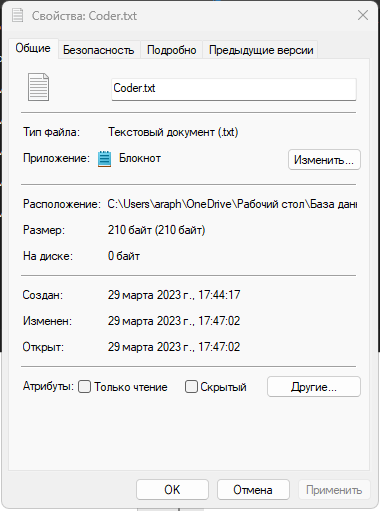
*Рисунок 6 – Размер закодированного файла с размером блока 16*



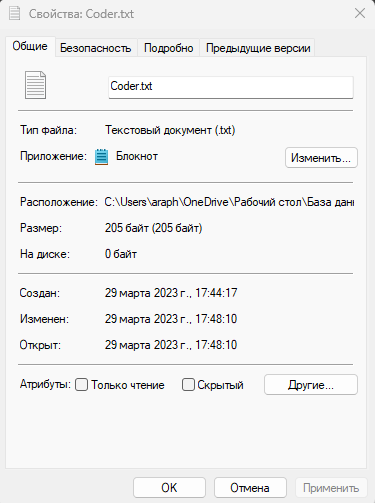
*Рисунок 7 – Размер закодированного файла с размером блока 24*



*Рисунок 8 – Размер закодированного файла с размером блока 32*



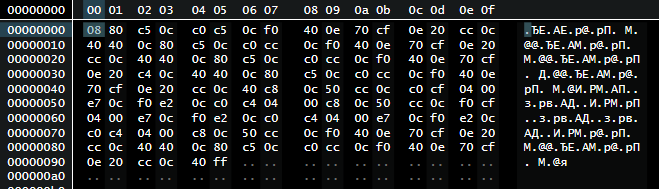
*Рисунок 9 – Размер закодированного файла с размером блока 48*



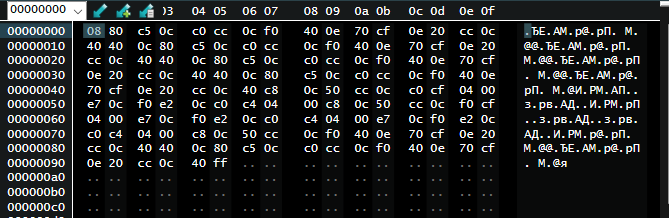
*Рисунок 10 – Размер закодированного файла с размером блока 64*

*График 1 - Зависимость размера файла от размера блока*

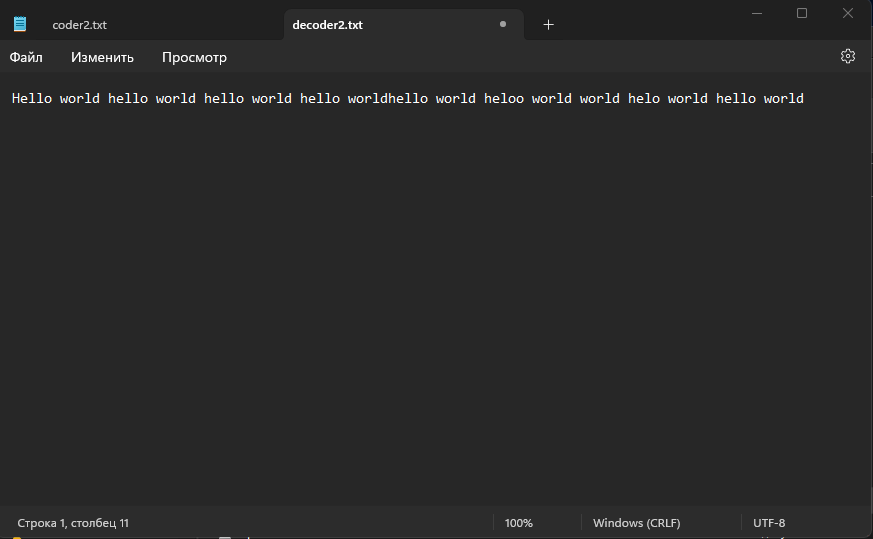
*Одиночные изменения закодированного сообщения в HEX коде*

**

*Рисунок 11 – Исходное закодированное сообщение в HEX редакторе*

**

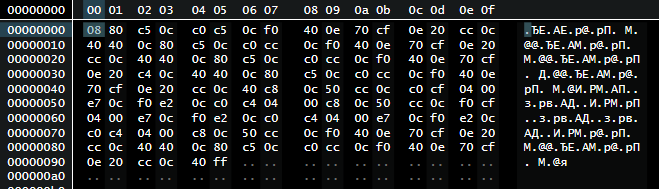
*Рисунок 12 – Изменённое закодированное сообщение при помощи HEX редактора*

**

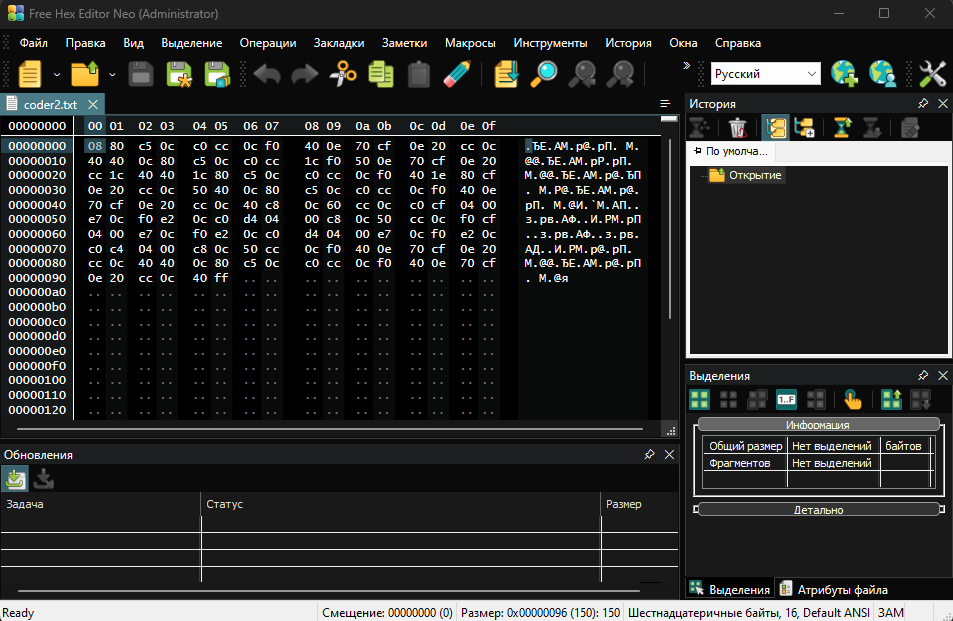
*Рисунок 13 – Полученное закодированное сообщение*

При внесении одиночных побитовых изменений и после запуска программы-декодера в выходном файле было получено закодированное сообщение.

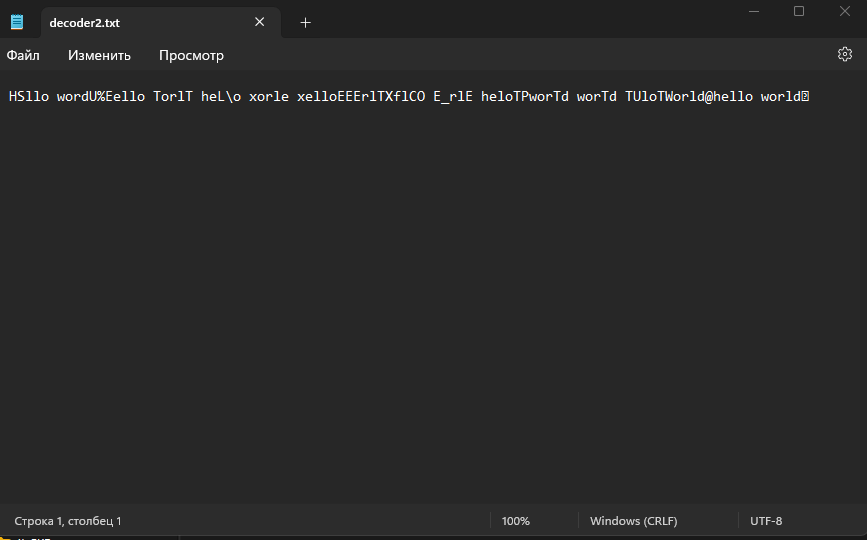
*Множественные изменения закодированного сообщения в HEX коде*

**

*Рисунок 14 – Исходное закодированное сообщение в HEX редакторе*

**

*Рисунок 15 – Изменённое закодированное сообщение при помощи HEX редактора*

**

*Рисунок 16 – Полученное закодированное сообщение*

При внесении групповых изменений моделируются массовые ошибки при передаче и хранении информации. При запуске программы-декодера исходного сообщения не было получено, однако разобрать части слов можно.

## Контрольные вопросы



Расстояние Хемминга — число позиций, в которых соответствующие символы двух слов одинаковой длины различны.



Этот метод применятся для исправления групповых ошибок. Код Рида-Соломона исправляющий t ошибок, требует 2t проверочных символов и с его помощью исправляются произвольный пакеты ошибок длиной t или меньше. Код Рида – Соломона является одним из наиболее мощных кодов, исправляющий многократные пакеты ошибок. Применяется в каналах, где пакеты ошибок могут образовываться столь часто, что их уже нельзя исправлять с помощью кодов, исправляющих одиночные ошибки.

При несистематическом кодировании информационное слово умножается на некий неприводимый полином в поле Галуа. Полученное закодированное слово полностью отличается от исходного и для извлечения информационного слова нужно выполнить операцию декодирования и уже потом можно проверить данные на содержание ошибок. Такое кодирование требует большие затраты ресурсов только на извлечение информационных данных, при этом они могут быть без ошибок.

При систематическом кодировании к информационному блоку из k символов приписываются 2t проверочных символов, при вычислении каждого проверочного символа используются все k символов исходного блока. В этом случае нет затрат ресурсов при извлечении исходного блока, если информационное слово не содержит ошибок, но кодировщик/декодировщик должен выполнить k(n-k) операций сложения и умножения для генерации проверочных символов. Кроме того, так как все операции проводятся в поле Галуа, то сами операции кодирования/декодирования требуют много ресурсов и времени.



Восьмой бит является контрольным битом. Если все биты расставлены верно, а ошибка выдается только в восьмом бите, значит ошибка именно в этом бите.

Если этот бит последний, то контролировать ошибку никак не получится. Если не последний, то требуется сосчитать сумму битов, которые контролирует этот бит. Если это число четно, то ставится ноль, если нечетно, то ставится единица.



Размер кодируемого блока в коде Хемминга отвечает за количество информационных бит. От этого зависит и количество контрольных бит.

Чем больше кодируемый блок, тем выше вероятность возникновения ошибки, которую нельзя исправить методом Хемминга. Так же от размера кодируемого блока зависит размер закодированного сообщения. Чем больше блок, тем меньше размер.

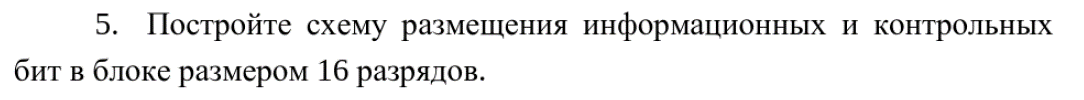


Схема размещения информационных и контрольных бит в блоке размером 16 разрядов была построена выше для расчета кода Хемминга для произвольного числа вручную.

## Выводы

В данной работе было изучено помехоустойчивое кодирование (метод Хемминга), с помощью которого удалось передать данные без ошибок. В случае возникновения одиночных ошибок они были исправлены. Код Хемминга является самоконтролирующиеся и самокорректирующимся кодом. Разработанный алгоритм не только находил ошибки, но и исправлял их.

## *Приложение*

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <math.h>

#include <malloc.h>

#include <Windows.h>

#include <locale.h>

#define CODER 1

#define DECODER 2

#define EXIT 3

int FindOutSize(int Block) {

int Size = 0;

if (Block == 8) Size = 12;

if (Block == 12) Size = 17;

if (Block == 16)Size = 21;

if (Block == 24) Size = 29;

if (Block == 32) Size = 38;

if (Block == 48) Size = 54;

if (Block == 64) Size = 71;

return Size;

}

int CheckFiles(FILE\* File) {

if (File == NULL) {

printf("Ошибка при открытии файла\n");

return EXIT\_FAILURE;

}

return EXIT\_SUCCESS;

}

int TextToBit(char\* NameFile, char\* Name) {

char Sym = 0;

int Symbol = 0;

FILE\* InputFile = fopen(NameFile, "rb");

FILE\* UseFile = fopen(Name, "wb");

if (CheckFiles(UseFile)) return EXIT\_FAILURE;

if (CheckFiles(InputFile)) return EXIT\_FAILURE;

fseek(UseFile, 7, SEEK\_SET);

while (!feof(InputFile)) {

Symbol = fgetc(InputFile);

if (Symbol < 0) {

break;

}

for (int i = 0; i <= 7; i++) {

Sym = (Symbol % 2) + '0';

fputc(Sym, UseFile);

Symbol = Symbol / 2;

if (i != 7) fseek(UseFile, -2, SEEK\_CUR);

else fseek(UseFile, -1, SEEK\_CUR);

}

fseek(UseFile, 15, SEEK\_CUR);

}

fclose(UseFile);

fclose(InputFile);

return EXIT\_SUCCESS;

}

int BitsToSymbol(int NumberBits[]) {

int Number = 0;

if (NumberBits[0] == 1)Number += 128;

if (NumberBits[1] == 1)Number += 64;

if (NumberBits[2] == 1)Number += 32;

if (NumberBits[3] == 1)Number += 16;

if (NumberBits[4] == 1)Number += 8;

if (NumberBits[5] == 1)Number += 4;

if (NumberBits[6] == 1)Number += 2;

if (NumberBits[7] == 1)Number += 1;

return Number;

}

int WriteToFile(char\* NameFile, char\* Name) {

int NumberBits[8] = { 0 }, Number;

char Symbol = 0;

FILE\* OutputFile = fopen(NameFile, "wb");

FILE\* UseFile = fopen(Name, "rb");

if (CheckFiles(UseFile)) return EXIT\_FAILURE;

if (CheckFiles(OutputFile)) return EXIT\_FAILURE;

while (!feof(UseFile)) {

memset(NumberBits, 0, sizeof(NumberBits));

for (int i = 0; i < 8; i++) {

if (!feof(UseFile)) Symbol = (char)fgetc(UseFile);

if (!feof(UseFile)) NumberBits[i] = (int)Symbol - '0';

}

Number = BitsToSymbol(NumberBits);

Symbol = (char)Number;

fputc(Symbol, OutputFile);

}

fclose(OutputFile);

fclose(UseFile);

return EXIT\_SUCCESS;

}

int AddControlBits(int Block) {

char Symbol = 0;

int Sum = 0, ControlBit = 0, Count = 0, NumberBits[71];

int Size = FindOutSize(Block);

memset(NumberBits, 0, sizeof(NumberBits));

FILE\* BinaryFile = fopen("Binary.txt", "rb");

FILE\* ControlFile = fopen("Control.txt", "wb");

if (CheckFiles(BinaryFile)) return EXIT\_FAILURE;

if (CheckFiles(ControlFile)) return EXIT\_FAILURE;

while (!feof(BinaryFile)) {

for (int i = 0; i < Size; i++) {

if (i == 0 || i == 1 || i == 3 || i == 7 || i == 15 || i == 31 || i == 63) NumberBits[i] = 0;

else {

Symbol = (char)fgetc(BinaryFile);

if (!feof(BinaryFile)) NumberBits[i] = (int)Symbol - '0';

else NumberBits[i] = 0;

}

}

for (int i = 0; i < Size; i++) {

if (i == 0 || i == 1 || i == 3 || i == 7 || i == 15 || i == 31 || i == 63) {

ControlBit = i + 1;

for (int j = 0; j < Size; j++) {

if (j < i) continue;

if (Count != ControlBit) {

Count++;

Sum = Sum + NumberBits[j];

}

if (Count == ControlBit) {

j += ControlBit;

Count = 0;

}

}

if (Sum % 2 != 0)NumberBits[i] = 1;

Sum = 0;

Count = 0;

}

}

for (int c = 0; c < Size; c++) {

Symbol = (char)NumberBits[c] + '0';

fputc(Symbol, ControlFile);

}

memset(NumberBits, 0, sizeof(NumberBits));

}

fclose(BinaryFile);

fclose(ControlFile);

return EXIT\_SUCCESS;

}

int CheckCorrectness(int NumberBits[], int Size) {

int ControlBit = 0, Count = 0, Sum = 0, Error = 0;

for (int i = 0; i < Size; i++) {

if (i == 0 || i == 1 || i == 3 || i == 7 || i == 15 || i == 31 || i == 63) {

ControlBit = i + 1;

for (int j = 0; j < Size; j++) {

if (j < i) continue;

if (Count != ControlBit) {

Count++;

if (j != i) Sum = Sum + NumberBits[j];

}

if (Count == ControlBit) {

j += ControlBit;

Count = 0;

}

}

if (Sum % 2 != 0 && NumberBits[i] != 1) {

Error += ControlBit;

}

else if (Sum % 2 == 0 && NumberBits[i] != 0) {

Error += ControlBit;

}

Sum = 0;

Count = 0;

}

}

if (Error != 0) {

if (NumberBits[Error] == 0) NumberBits[Error] = 1;

else NumberBits[Error] = 0;

}

return EXIT\_SUCCESS;

}

int DeleteConrolBits(int Block) {

char Symbol = 0;

int NumberBits[71] = { 0 };

int Size = FindOutSize(Block);

FILE\* UseFile = fopen("Binary.txt", "wb");

FILE\* File = fopen("Control.txt", "rb");

if (CheckFiles(UseFile)) return EXIT\_FAILURE;

if (CheckFiles(File)) return EXIT\_FAILURE;

while (!feof(File)) {

for (int i = 0; i < Size; i++) {

if (!feof(File))Symbol = (char)fgetc(File);

if (!feof(File))NumberBits[i] = (int)Symbol - '0';

else NumberBits[i] = 0;

}

CheckCorrectness(NumberBits, Size);

for (int j = 0; j < Size; j++) {

if (j == 0 || j == 1 || j == 3 || j == 7 || j == 15 || j == 31 || j == 63) continue;

else {

Symbol = (char)NumberBits[j] + '0';

fputc(Symbol, UseFile);

}

}

}

fclose(UseFile);

fclose(File);

return EXIT\_SUCCESS;

}

int Coder(void) {

int Block = 0;

char NameFile[50] = { 0 };

printf("Введите размер текстового блока:");

if (scanf("%d", &Block) == 0)

return EXIT\_FAILURE;

while ((getchar()) != '\n');

if (Block != 8 && Block != 16 && Block != 32 && Block != 64 && Block != 12 && Block != 48 && Block != 24) {

printf("Введен неправильный размер текстового блока\n");

return EXIT\_FAILURE;

}

printf("Введите имя исходного файла:");

if (scanf("%s", &NameFile) == 0)return EXIT\_FAILURE;

if (TextToBit(NameFile, "Binary.txt")) return EXIT\_FAILURE;

if (AddControlBits(Block))return EXIT\_FAILURE;

WriteToFile("Coder.txt", "Control.txt");

remove("Binary.txt");

remove("Control.txt");

return EXIT\_SUCCESS;

}

int Decoder(void) {

int Block = 0;

char NameFile[50] = { 0 };

printf("Введите размер текстового блока:");

if (scanf("%d", &Block) == 0)return EXIT\_FAILURE;

while ((getchar()) != '\n');

if (Block != 8 && Block != 16 && Block != 32 && Block != 64 && Block != 12 && Block != 48 && Block != 24) {

printf("Введен неправильный размер текстового блока\n");

return EXIT\_FAILURE;

}

printf("Введите имя файла, который хотите декодировать:");

if (scanf("%s", &NameFile) == 0)return EXIT\_FAILURE;

if (TextToBit(NameFile, "Control.txt")) return EXIT\_FAILURE;

if (DeleteConrolBits(Block)) return EXIT\_FAILURE;

WriteToFile("Output.txt", "Binary.txt");

remove("Binary.txt");

remove("Control.txt");

return EXIT\_SUCCESS;

}

void Menu(void) {

while (1) {

int Choice = 0;

printf("Введите действие:\n1: Кодирование\n2: Декодирование\n3: Выход из программы\n");

if (scanf("%d", &Choice) == 0)return;

while ((getchar()) != '\n');

if (Choice == CODER || Choice == DECODER || Choice == EXIT) {

if (Choice == EXIT) break;

if (Choice == CODER) {

Coder();

continue;

}

if (Choice == DECODER) {

Decoder();

continue;

}

}

else printf("Ошибка ввода, попробуйте еще раз\n");

}

}

int main(void) {

setlocale(LC\_ALL, "Rus");

Menu();

return EXIT\_SUCCESS;

}