1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт компьютерных наук и технологий
5. **Кафедра «Информационная безопасность компьютерных систем»**

**Лабораторная работа 4**

1. «Методы надежной передачи данных»
2. по дисциплине «Практикум по информационной безопасности»

Выполнил:

студент гр. 13558/1 Никулкин В. А.

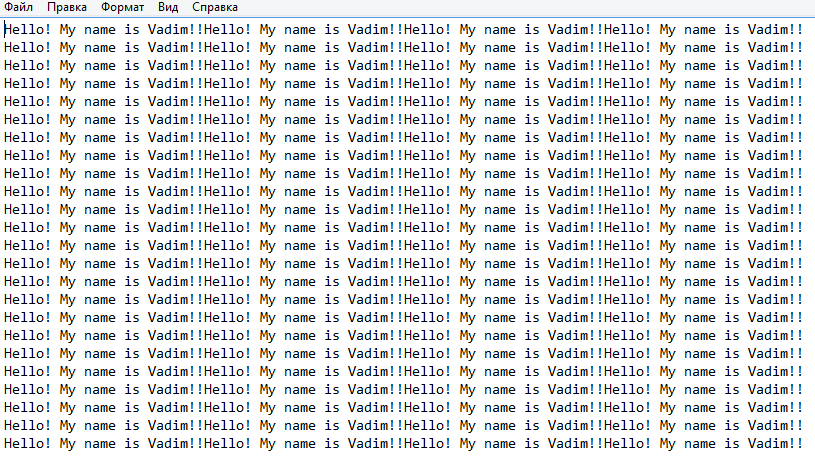
Проверил:

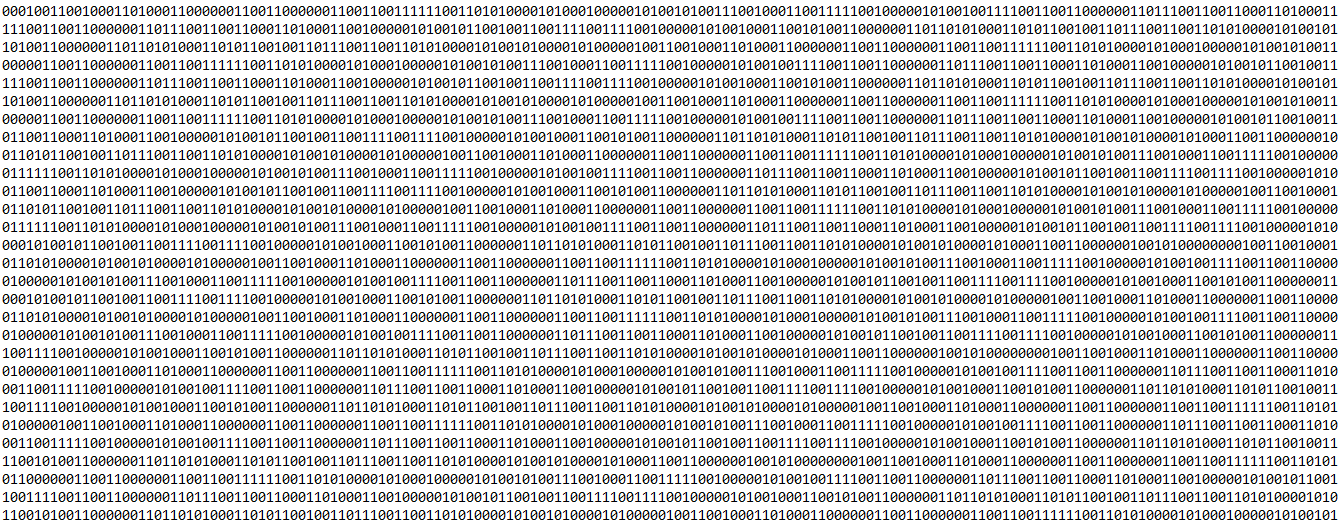
ассистент преподавателя Алексеев. И. В.

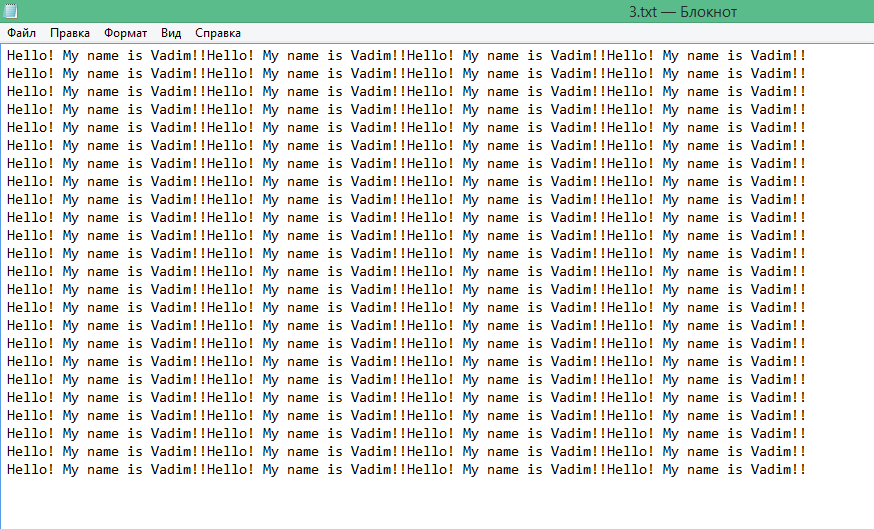
1. Санкт-Петербург
2. 2017
3. **Цель работы**

Ознакомиться с методом Хемминга помехоустойчивого кодирования, позволяющим обнаруживать и автоматически исправлять ошибки, возникающие при хранении и передаче информации.

1. **Ход работы**
2. 1) Был создан текстовый файл “1.txt”. В него был записан некоторый набор символов.



1. 2) Далее файл был закодирован при помощи программы осуществляющей кодирование Хэмминга. Закодированное сообщение было помещено в файл “2.txt”
2. 
   * 1. 3) Далее были внесены однобитовые ошибки, при этом при расшифровке целостность сообщения не была нарушена



4) При внесении групповых ошибок была нарушена целостность сообщения, и оно было неправильно декодировано

5) Пример расчета кода Хемминга для 16-разрядного числа:

Возьмем, например, число 0100010000111101. Прежде всего, необходимо вставить контрольные биты. Они вставляются в строго определённых местах — это позиции с номерами, равными степеням двойки. В нашем случае (при длине информационного слова в 16 бит) это будут позиции 1, 2, 4, 8, 16. Соответственно, у нас получилось 5 контрольных бит.

Получается следующее, \_\_0\_100\_0100001\_ 11101

Теперь необходимо вычислить значение каждого контрольного бита. Значение каждого контрольного бита зависит от значений информационных бит, но не от всех, а только от тех, которые этот контрольных бит контролирует. Для того чтобы понять, за какие биты отвечает каждых контрольный бит необходимо понять очень простую закономерность: контрольный бит с номером N контролирует все последующие N бит через каждые N бит, начиная с позиции N. Теперь, берём каждый контрольный бит и смотрим, сколько среди контролируемых им битов единиц, получаем некоторое целое число и, если оно чётное, то ставим ноль, в противном случае ставим единицу.

Получаем следующее 10011000010001011101

6) Блок-схема алгоритма вычисления помехоустойчивого кодирования\

Пока счетчик не станет равен числу блоков

Копирование элементов из буфферной строки в ту, которая будет содержать закодированное сообщение

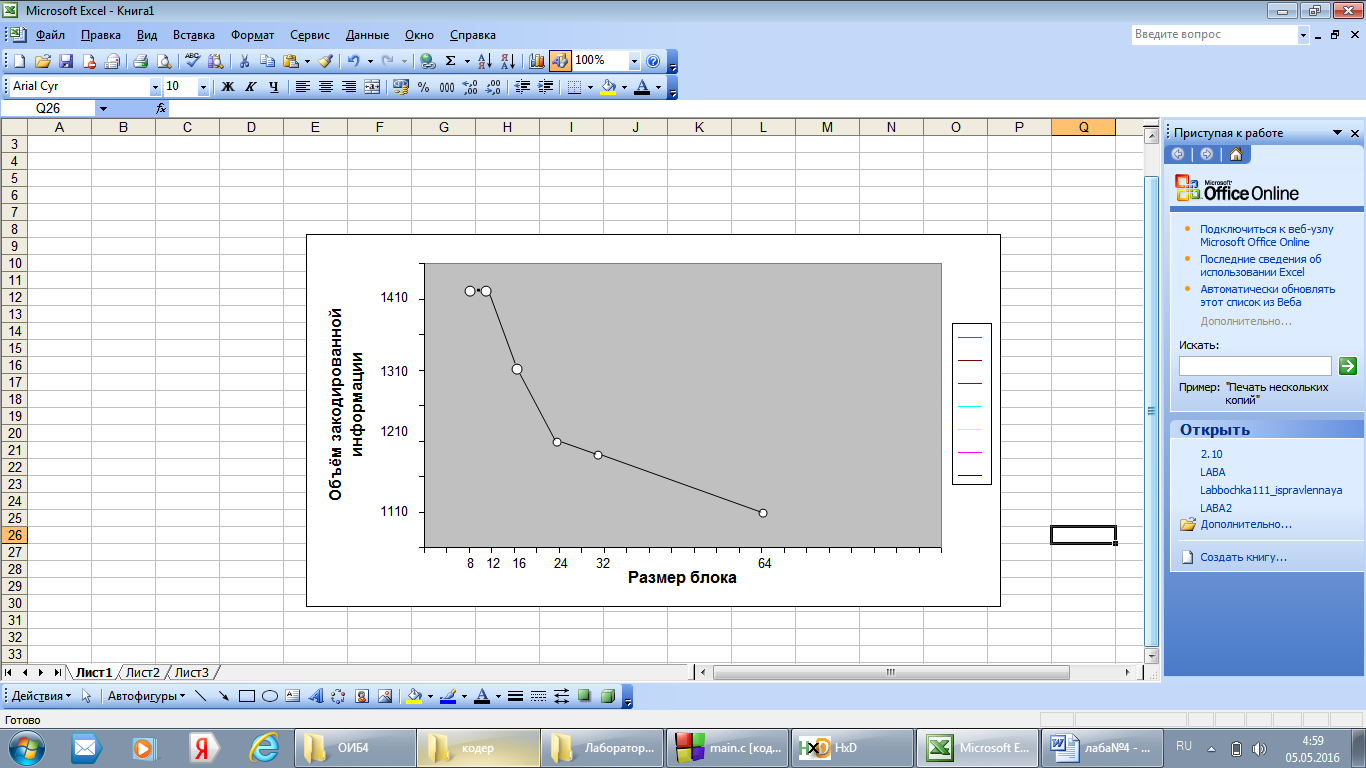
Контрольные биты вычисляются по следующему алгоритму: Значение каждого контрольного бита зависит от значений информационных бит, но не от всех, а только от тех, которые этот контрольных бит контролирует. Для того чтобы понять, за какие биты отвечает каждых контрольный бит необходимо понять очень простую закономерность: контрольный бит с номером N контролирует все последующие N бит через каждые N бит, начиная с позиции N. Теперь, берём каждый контрольный бит и смотрим, сколько среди контролируемых им битов единиц, получаем некоторое целое число и, если оно чётное, то ставим ноль, в противном случае ставим единицу.

Копирование битов из изначальной строки в буфферную, на месте контрольных битов остаются пока нули.

Вычисление длины блока с контрольными битами, затем создания буфферной строки этого размера.

Программ получает на вход изначальное сообщение, размер логического блока, количество логических блоков

7) График зависимости изменения объёма закодированной информации от размера кодируемых блоков.(если изначальное сообщение 1000 символов)



8) Листинг программы:

#include <stdio.h>

#include <stdlib.h>

#include <io.h>

struct code{

char \*data;

int bits;

};

struct code create\_zero(int n)

{

struct code c;

int i = n/8 + (n % 8 ? 1 : 0);

c.data = (char\*)calloc(i, sizeof(char));

c.bits = n;

return(c);

}

void set\_bit(int index, int value, struct code c)

{

int bait = index / 8;

int index\_bit = index % 8;

int a = 1 << index\_bit;

if(value == 1)

c.data[bait] |= a;

else

c.data[bait] &= ~a;

}

int get\_bit(int index, struct code c)

{

int bait = index / 8;

int index\_bit = index % 8;

int a = (c.data[bait] >> index\_bit) & 1;

return(a);

}

int length(int block)

{

int i = 1;

int len = block;

while(i <= block)

{

i \*= 2;

++len;

}

return len;

}

struct code encode(struct code c, int block, int number\_of\_blocks)

{

int i = 1;

int j;

int size\_block = length(block);

int l = 1;

int original\_pos = 0;

int encoded\_pos = 0;

int counter1 = 0;

struct code d = create\_zero(size\_block \* number\_of\_blocks);

while(l < number\_of\_blocks + 1)

{

encoded\_pos = 0;

i = 1;

int counter = 0;

struct code a = create\_zero(size\_block);

while(1)

{

if(counter == block)

break;

if(encoded\_pos == i - 1)

{

i += i;

++ encoded\_pos;

continue;

}

set\_bit(encoded\_pos, get\_bit(original\_pos, c), a);

++ original\_pos;

++ encoded\_pos;

++counter;

}

int number\_of\_control = size\_block - block;

for(j = 0; j < number\_of\_control; ++j)

{

int sum = 0;

int skip = 0;

int step = 0;

for(int k = (1 << j) - 1; k < size\_block; ++k)

{

++step;

if(!skip)

sum += get\_bit(k, a);

if(step % (1 << j) == 0)

skip = (skip + 1) % 2;

}

if(sum % 2 == 0)

set\_bit((1 << j) - 1, 0, a);

else

set\_bit((1 << j) - 1, 1, a);

}

for(counter1; counter1 < size\_block \* l; ++counter1 )

set\_bit(counter1, get\_bit(counter1 % size\_block, a), d);

++l;

free(a.data);

}

return(d);

}

struct code decode(struct code d, int block)

{

int size\_block = length(block);

int number\_of\_blocks = d.bits / size\_block + (d.bits % size\_block ? 1 : 0);

int original\_pos = 0;

int encoded\_pos = 0;

int counter1 = 0;

int l = 1;

int counter2 = 0;

int i;

int counter3 = 0;

int j;

int counter4 = 0;

int counter5 = 0;

struct code v = create\_zero(block \* number\_of\_blocks);

while(l < number\_of\_blocks + 1)

{

encoded\_pos = 0;

i = 1;

int counter = 0;

struct code a = create\_zero(size\_block);

struct code q = create\_zero(size\_block);

for(int p = 0; p < size\_block; ++p)

{

set\_bit(p, get\_bit(counter4, d), q);

++counter4;

}

while(1)

{

if(counter == size\_block)

break;

set\_bit(encoded\_pos, get\_bit(original\_pos, d), a);

++ original\_pos;

++ encoded\_pos;

++counter;

}

int number\_of\_control = size\_block - block;

int h = 0;

int z = 1;

while(1)

{

if(h == number\_of\_control)

break;

set\_bit(z - 1, 0, a);

++h;

z += z;

}

for(j = 0; j < number\_of\_control; ++j)

{

int sum = 0;

int skip = 0;

int step = 0;

for(int k = (1 << j) - 1; k < size\_block; ++k)

{

++step;

if(!skip)

sum += get\_bit(k, a);

if(step % (1 << j) == 0)

skip = (skip + 1) % 2;

}

if(sum % 2 == 0)

set\_bit((1 << j) - 1, 0, a);

else

set\_bit((1 << j) - 1, 1, a);

}

counter1 = 0;

for(int u = 1; u < size\_block; u += u)

{

if(get\_bit(u - 1, a) != get\_bit(u - 1, q))

counter1 += u;

}

if(counter1 == 0)

set\_bit(counter1,get\_bit(counter1, a), a);

else

{

if(get\_bit(counter1 - 1, a) == 0)

set\_bit(counter1 - 1, 1, a);

else

set\_bit(counter1 - 1, 0, a);

}

counter2 = 0;

i = 1;

counter3 = 0;

struct code buf = create\_zero(block);

while(counter2 < a.bits)

{

if(counter2 == i - 1)

{

i = 2 \* i;

++ counter2;

continue;

}

set\_bit(counter3, get\_bit(counter2, a), buf);

++counter3;

++counter2;

}

counter2 = 0;

while(counter2 < buf.bits)

{

set\_bit(counter5, get\_bit(counter2, buf), v);

++counter5;

++counter2;

}

++l;

free(a.data);

free(q.data);

}

free(d.data);

return (v);

}

int main()

{

FILE \*in = fopen("1.txt", "rb");

FILE \*out = fopen("2.txt", "wb");

FILE \*result = fopen("3.txt", "wb");

struct code c;

char symbol;

fseek(in, 0, SEEK\_END);

int longth = ftell(in);

fseek(in, 0, SEEK\_SET);

c.data = (char\*) malloc(sizeof(char) \* longth);

fread (c.data, 1, longth, in);

c.bits = longth \* 8;

c.data[longth] = '\0';

int k = 0;

int block = 8;

int len = longth \* 8 / block + (longth \* 8 % block ? 1 : 0);

int size\_block = length(block);

struct code d;

d = encode(c, block, len);

struct code v = create\_zero(block \* len);

for(int p = 0; p < d.bits; ++p)

fprintf(out, "%d", get\_bit(p, d));

fclose(out);

system("2.txt");

out = fopen("2.txt", "rb");

for(int p = 0; p < d.bits; ++p)

{

int bit = fgetc(out) - '0';

set\_bit(p, bit, d);

}

v = decode(d, block);

v.data[v.bits / 8 + (v.bits % 8 ? 1 : 0)] = '\0';

fprintf(result, "%s", v.data);

free(d.data);

free(c.data);

fclose(result);

fclose(out);

fclose(in);

return 0;

}

9) Ответы на контрольные вопросы:

1. Расстояние Хемминга – минимальное число разрядов с неодинаковыми значениями кодовых слов.
2. Преимущество кодов Рида-Соломона в том, что они способны исправлять групповые ошибки.
3. К информационному блоку из k символов приписываются 2t проверочных символов, при вычислении каждого из которых используются все k символов исходного блока. К исходному блоку приписываются 2t нулей, получается полином S. Этот полином делится на пораждающий полином G, находится остаток R, S = QG + R, где Q – частное. Этот остаток и приписывается к исходному блоку символов.
4. Размер кодируемого блока влияет на количество контрольных бит, которых он будет содержать. Чем больше размер блока, тем эффективнее кодируется информация.
5. Номера контрольных бит – степени двойки. 0 – контрольный бит, 1 – информационный бит.

001011101111111011111

**Вывод**

Для проверки целостности информации при передаче и исправлении ошибок без повторной передачи, используется помехоустойчивое кодирование. Такие коды способны закодировать сообщение и после передачи определить, возникла ли ошибка, и по возможности, восстановить её.