Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Избыточное кодирование данных в информационных системах. Код Хемминга

Студент: Стрелковская В. А.

ФИТ 3 курс 2 группа

Преподаватель: Хартанович А. А.

Минск 2025

# Теоретические сведения

**Цель**:приобретение практических навыков кодирования/декодирования двоичных данных при использовании кода Хемминга.

**Надежность системы** – характеристика способности программного, аппаратного, аппаратно-программного средства выполнить при определенных условиях требуемые функции в течение конкретного периода времени.

**Достоверность работы системы** (устройства) – свойство, характеризующее истинность конечного (выходного) результата работы (выполнения программы), определяемое способностью средств контроля фиксировать правильность или ошибочность работы.

**Ошибка устройства** – неправильное значение сигнала (бита – в цифровом устройстве) на внешних выходах устройства или отдельного его узла, вызванное технической неисправностью, или воздействующими на него помехами (преднамеренными либо непреднамеренными), или иным способом.

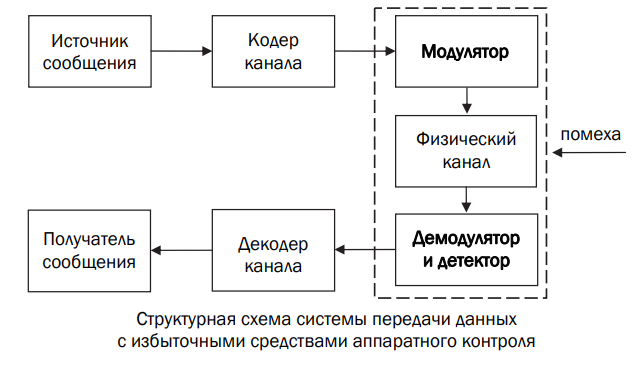
**Ошибка программы** – проявляется в не соответствующем реальному (требуемому) промежуточном или конечном значении (результате) вследствие неправильно запрограммированного алгоритма или неправильно составленной программы.

Надежность есть внутреннее свойство объекта, заложенное в него при изготовлении и проявляющееся во время эксплуатации. Вторая особенность надежности состоит в том, что она проявляется во времени. И третья особенность надежности выражается по-разному при различных условиях эксплуатации и различных режимах применения объекта (информационной системы в целом, отдельного ее блока, канала передачи сообщения, оперативной или внешней памяти компьютера).

**Надежность** – комплексное свойство, включающее в себя единичные свойства: безотказность, ремонтопригодность, сохраняемость, долговечность.

**Безотказность** – это свойство технического объекта непрерывно сохранять работоспособное состояние в течение некоторого времени.

**Ремонтопригодность** – это свойство технического объекта, заключающееся в приспособленности к поддержанию и восстановлению работоспособного состояния путем технического обслуживания.



Информационное слово Хk (длина сообщения – k символов) в избыточное сообщение (кодовое слово) Xn длиной n символов (n > k).

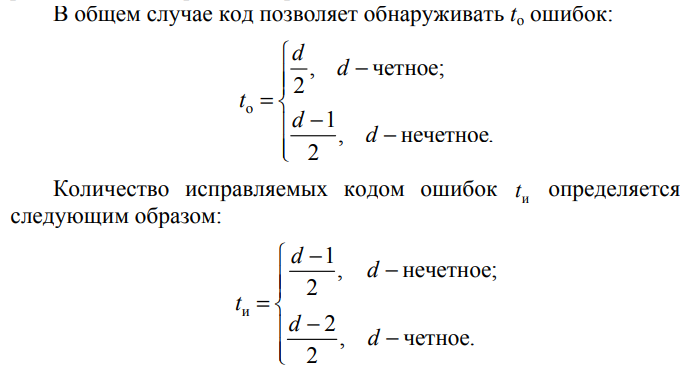
**Кратность ошибки** – количество ошибок.

Число k часто называется размерностью кода.

Число r называется избыточными битами/битами чётности/контрольными битами.

Отношение числа битов данных к общему числу битов k/n именуется степенью кодирования (code rate) – **доля кода**, которая приходится на полезную информацию.

**Расстояние Хемминга** (d), которое показывает, что два кодовых слова различаются по крайней мере в d позициях.



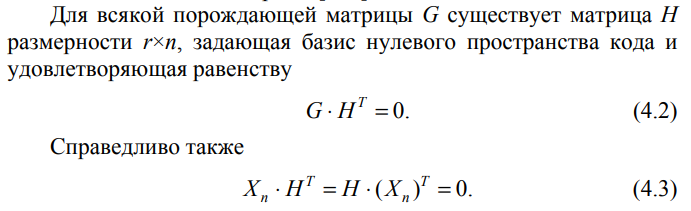
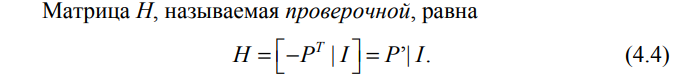
Код Хемминга относится к классу линейных блочных кодов. **Линейные блочные коды** – это класс кодов с контролем четности, которые можно описать парой чисел (n, k).

Для формирования r проверочных символов (кодирования), т. е. вычисления проверочного слова Xr, используется порождающая матрица G: совокупность базисных векторов будем далее записывать в виде матрицы G размерностью k×n с единичной подматрицей (I) в первых k строках и столбцах:



Более точно матрица G называется порождающей матрицей линейного корректирующего кода в приведенно-ступенчатой форме.

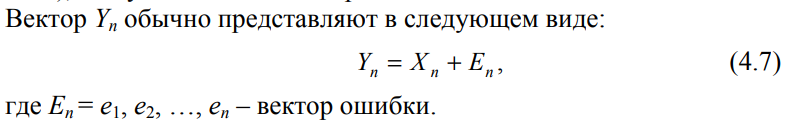
**Кодирование** заключается в умножении вектора сообщения Хk длиной k на порождающую матрицу по правилам матричного умножения (все операции выполняются по модулю 2).



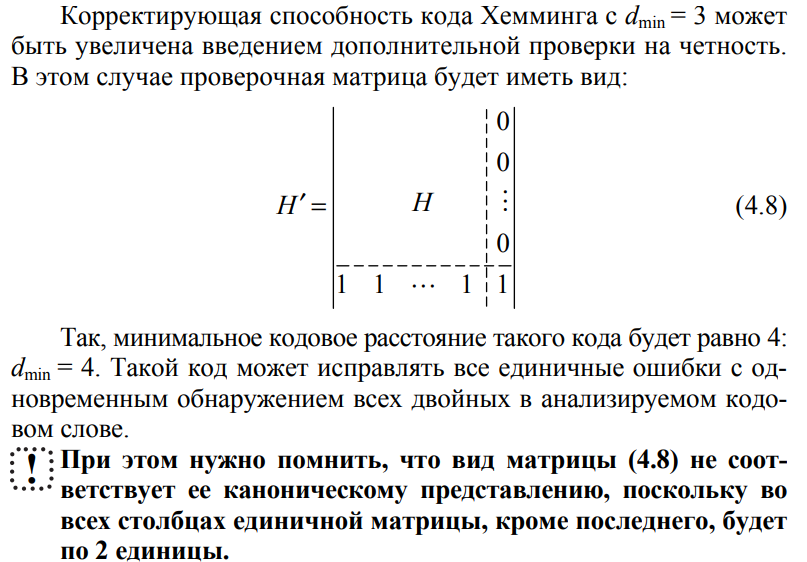
В коде Хемминга с минимальным кодовым расстоянием dmin = 3 проверочная матрица Н имеет классический вид и состоит из двух подматриц: P’ размером k×r и I размером r×r соответственно.

Результат умножения сообщения на выходе канала передачи (Yn) или (что равнозначно) сообщения, считываемого из памяти, на проверочную матрицу (Н) называется **синдромом** **(вектором ошибки) S:**





**Важно запомнить, что в силу выражений (4.3)–(4.7) ненулевой синдром всегда равен сумме по модулю 2 тех векторстолбцов матрицы Н, номера которых соответствуют номерам ошибочных битов в слове Yn.**

****

# Практическое задание

# 2.1 Преобразование в бинарный вид

Для выполнения вычислений необходимо преобразовать сообщение в бинарный вид.. Представлено на рисунке 2.1.

// Исходное сообщение

string message = "TEST";

string binaryMessage = StringToBinary(message);

Console.WriteLine($"Бинарное представление: {binaryMessage}");

// Вычисляем длины

int k = binaryMessage.Length;

int r = CalculateRedundantBits(k);

int n = k + r;

Console.WriteLine($"Длина исходного сообщения (k): {k}");

Console.WriteLine($"Количество избыточных битов (r): {r}");

static string StringToBinary(string text)

{

StringBuilder sb = new StringBuilder();

foreach (char c in text)

sb.Append(Convert.ToString(c, 2).PadLeft(8, '0'));

return sb.ToString();

}

static int CalculateRedundantBits(int m)

{

int r = 0;

while (Math.Pow(2, r) < m + r + 1)

r++;

return r;

}

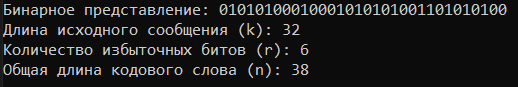


Рисунок 2.1 – Представление в бинарном виде

Далее выполняем построение матрицы Хемминга. Представлено на рисунке 2.2.

// Строим проверочную матрицу Хемминга

int[,] H = BuildHammingMatrix(n, r);

Console.WriteLine("\nПроверочная матрица Хемминга:");

PrintMatrix(H);

// Функция для построения матрицы Хемминга

static int[,] BuildHammingMatrix(int n, int r)

{

int[,] H = new int[r, n];

// Строим матрицу A (первые r столбцов)

int col = 0;

for (int i = 1; col < n - r; i++)

{

// Генерация столбца в двоичной форме

string binary = Convert.ToString(i, 2).PadLeft(r, '0');

int onesCount = CountOnes(binary);

// Если столбец содержит меньше 2 единиц, пропускаем его

if (onesCount < 2)

continue;

// Заполняем столбец в матрице

for (int j = 0; j < r; j++)

{

H[j, col] = binary[j] - '0';

}

col++;

}

// Заполняем единичную матрицу в правой части

for (int i = 0; i < r; i++)

{

for (int j = 0; j < r; j++)

{

H[i, n - r + j] = (i == j) ? 1 : 0;

}

}

return H;

}

static int CountOnes(string binary)

{

int count = 0;

foreach (char c in binary)

{

if (c == '1')

count++;

}

return count;

}

static void PrintMatrix(int[,] matrix)

{

for (int i = 0; i < matrix.GetLength(0); i++)

{

for (int j = 0; j < matrix.GetLength(1); j++)

Console.Write(matrix[i, j] + " ");

Console.WriteLine();

}

}

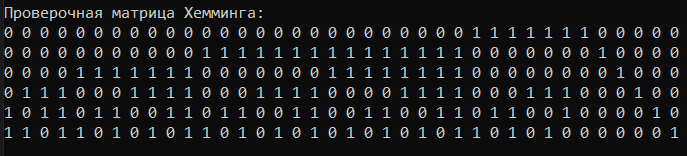


Рисунок 2.2 – Матрица Хемминга

Для преобразования информационного слова Xk в кодовое слово Xn необходимо найти избыточные символы. Для этого используется функция, представленная на рисунке 2.3.

//вычисление избыточных символов

CalculateRedundantSymbols(binaryMessage, H, r);

static void CalculateRedundantSymbols(string binaryMessage, int[,] H, int r)

{

int dataLength = binaryMessage.Length;

// Подготовка вектора для бит данных (в виде массива целых чисел)

int[] dataBits = new int[dataLength];

for (int i = 0; i < dataLength; i++)

{

dataBits[i] = binaryMessage[i] - '0';

}

// Подготовка массива для избыточных символов

int[] redundantSymbols = new int[r];

// Вычисление избыточных символов с использованием матрицы Хемминга

for (int i = 0; i < r; i++)

{

int sum = 0;

for (int j = 0; j < dataLength; j++)

{

// Учитываем только соответствующие части H

if (H[i, j] == 1)

{

sum += dataBits[j];

}

}

// Избыточный символ - это бит четности

redundantSymbols[i] = sum % 2;

}

// Вывод вычисленных избыточных символов

Console.WriteLine("\nВычисленные избыточные символы:");

for (int i = 0; i < r; i++)

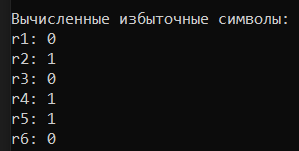
{

Console.WriteLine($"r{i + 1}: {redundantSymbols[i]}");

}

}

Рисунок 2.3 – Функция вычисления избыточных символов



Результат

Следующий шаг – генерация кодового слова с избыточными символами. Функции представлены на рисунке 2.4.

// Генерация кодового слова с избыточными битами

string encodedMessage = EncodeHamming(binaryMessage, n, r);

Console.WriteLine($"\nЗакодированное сообщение: {encodedMessage}");

static string EncodeHamming(string data, int n, int r)

{

// Создаем массив для кодированного сообщения

char[] code = new char[n];

int dataIdx = 0;

// Заполняем массив кодированного сообщения

for (int i = 0; i < n; i++)

{

if ((i & (i + 1)) == 0) // Проверка, является ли позиция избыточным битом

code[i] = '0'; // Позиция избыточного бита

else

code[i] = data[dataIdx++]; // Заполнение битами данных

}

// Вычисляем избыточные символы

int[] redundantSymbols = new int[r];

for (int i = 0; i < r; i++)

{

int sum = 0;

for (int j = 0; j < n; j++)

{

if (((j + 1) & (1 << i)) != 0) // Проверка, включает ли текущая позиция избыточный бит

sum += (code[j] - '0');

}

redundantSymbols[i] = sum % 2; // Избыточный символ - это бит четности

}

// Формируем строку из исходных данных

StringBuilder encodedMessage = new StringBuilder(data);

// Добавляем избыточные символы в зеркальном порядке

for (int i = r - 1; i >= 0; i--)

{

encodedMessage.Append(redundantSymbols[i]);

}

return encodedMessage.ToString(); // Возвращаем итоговое закодированное сообщение

}

Рисунок 2.4 – Функция генерации кодового слова



Далее было выполнено вычисление сообщения с ошибкой. Представлено на рисунке 2.5.

// Вносим случайную ошибку

Random rnd = new Random();

int errorCount = rnd.Next(3);

string corruptedMessage = IntroduceErrors(encodedMessage, errorCount);

Console.WriteLine($"Сообщение с {errorCount} ошибками: {corruptedMessage}");

static string IntroduceErrors(string data, int numErrors)

{

Random rnd = new Random();

char[] corrupted = data.ToCharArray();

for (int i = 0; i < numErrors; i++)

{

int errorPos;

do { errorPos = rnd.Next(data.Length); }

while (corrupted[errorPos] == 'X');

corrupted[errorPos] = corrupted[errorPos] == '0' ? '1' : '0';

}

return new string(corrupted);

}

Рисунок 2.5 – Вычисление сообщения с ошибкой

На данном этапе есть 3 варианта, для каждого из который вычисляются новые избыточные символы:

1. 0 ошибок

CalculateRedundantSymbolsWithErrors(corruptedMessage, H, r);

static void CalculateRedundantSymbolsWithErrors(string receivedMessage, int[,] H, int r)

{

int totalLength = receivedMessage.Length;

// Подготовка вектора для закодированных бит (в виде массива целых чисел)

int[] receivedBits = new int[totalLength];

for (int i = 0; i < totalLength; i++)

{

receivedBits[i] = receivedMessage[i] - '0';

}

// Подготовка массива для новых избыточных символов

int[] newRedundantSymbols = new int[r];

// Вычисление новых избыточных символов с использованием матрицы Хемминга

for (int i = 0; i < r; i++)

{

int sum = 0;

for (int j = 0; j < totalLength; j++)

{

// Учитываем только соответствующие части H

if (H[i, j] == 1)

{

sum += receivedBits[j];

}

}

// Новый избыточный символ - это бит четности

newRedundantSymbols[i] = sum % 2;

}

// Вывод вычисленных новых избыточных символов

Console.WriteLine("\nВычисленные новые избыточные символы (Yr'):");

for (int i = 0; i < r; i++)

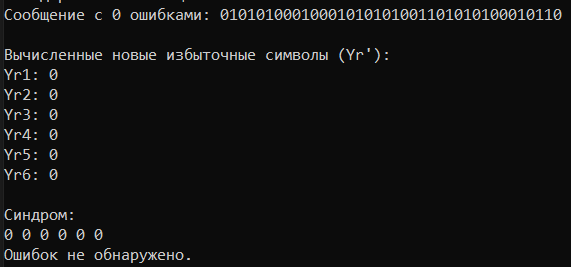
{

Console.WriteLine($"Yr{i + 1}: {newRedundantSymbols[i]}");

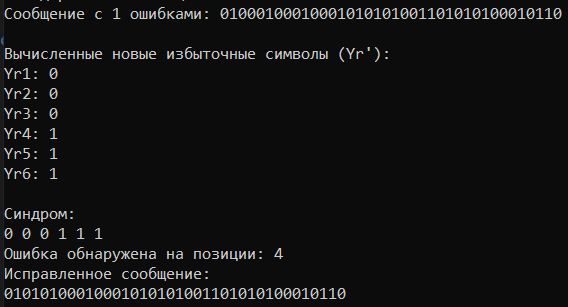
}

}

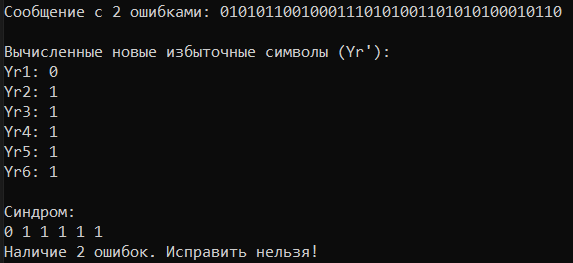
Рисунок 2.6 – Функция вычисления новых избыточных символов

****

1. 1 ошибка



1. 2 шибки



Также было необходимо выполнить вычисление синдрома и исправление ошибки. Функция для вычисления представлена на рисунке 2.7:

static void AnalyzeSyndrome(string receivedMessage, int[,] H, int r, int errorCount)

{

// Вычисляем новые избыточные символы

int totalLength = receivedMessage.Length;

int[] receivedBits = new int[totalLength];

for (int i = 0; i < totalLength; i++)

{

receivedBits[i] = receivedMessage[i] - '0';

}

// Вычисляем синдром

int[] syndrome = new int[r];

for (int i = 0; i < r; i++)

{

int sum = 0;

for (int j = 0; j < totalLength; j++)

{

if (H[i, j] == 1)

{

sum += receivedBits[j];

}

}

syndrome[i] = sum % 2;

}

// Вывод синдрома

Console.WriteLine("\nСиндром:");

for (int i = 0; i < r; i++)

{

Console.Write(syndrome[i] + " ");

}

Console.WriteLine();

// Проверка на количество ошибок

if (errorCount > 1)

{

Console.WriteLine("Нельзя решить");

return; // Завершаем выполнение функции

}

// Проверка на наличие одной ошибки

int errorPos = 0;

for (int i = 0; i < r; i++)

{

if (syndrome[i] == 1)

{

errorPos += (1 << i);

}

}

for (int j = 0; j < totalLength; j++)

{

// Проверяем, совпадает ли синдром с j-м столбцом H

bool match = true;

for (int i = 0; i < r; i++)

{

if (H[i, j] != syndrome[i])

{

match = false;

break;

}

}

if (match)

{

errorPos = j + 1; // +1 для 1-индексированной позиции

break;

}

}

// Исправление ошибки, если она найдена

if (errorPos > 0)

{

Console.WriteLine($"Ошибка обнаружена на позиции: {errorPos}");

receivedBits[errorPos - 1] ^= 1; // Исправление бита

// Вывод исправленного сообщения

Console.WriteLine("Исправленное сообщение:");

foreach (var bit in receivedBits)

{

Console.Write(bit);

}

Console.WriteLine(); // Переход на новую строку

}

else

{

Console.WriteLine("Ошибок не обнаружено.");

}

}

Рисунок 2.7 – Функция вычисления синдрома и исправления ошибки

**Вывод:** В данной лабораторной работе был реализован и протестирован алгоритм кодирования и декодирования кода Хемминга.

Результат работы показал, что код Хемминга эффективно решает устранение одной ошибки. Происходит корректное определение и исправление ошибки.

Также было протестировано внесение разного количества ошибок и рассмотрено их исправление. Программа корректно вычисляет синдром и исправляет одиночные ошибки, выводит сообщение при отсутствии ошибок и наличии 2 ошибок.