Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Защита информации и надёжность информационных систем

Студент: Лопатнюк П.В.

ФИТ 3 курс 1 группа

Преподаватель: Нистюк О.А.

Минск 2025

**Лабораторная работа № 4**

ИЗБЫТОЧНОЕ КОДИРОВАНИЕ ДАННЫХ В ИНФОРМАЦИОННЫХ СИСТЕМАХ. КОД ХЕММИНГА

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

**Задачи:**

1. Закрепить теоретические знания по использованию методов помехоустойчивого кодирования для повышения надежности передачи и хранения в памяти компьютера двоичных данных.
2. Разработать приложение для кодирования/декодирования двоичной информации кодом Хемминга с минимальным кодовым расстоянием 3 или 4.
3. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.

# Теоретические сведения

Надёжность системы – характеристика способности программного, аппаратного, аппаратно-программного средства выполнить при определённых 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.

Как следует из вышеприведенного определения, 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).

Избыточное кодирование направлено на повышение надежности передачи и хранения данных за счет обнаружения и исправления ошибок, возникающих из-за помех в каналах связи, сбоев оборудования или программных ошибок.

Функциональная сущность:

* Обнаружение ошибок – позволяет выявить факт искажения данных.
* Исправление ошибок – восстанавливает исходные данные без повторной передачи.
* Контроль целостности – обеспечивает достоверность информации при хранении и передаче.

Применение в информационных системах:

* Передача данных (Wi-Fi, сотовая связь, оптоволокно).
* Хранение информации (RAID, ECC-память, SSD).
* Криптография (контроль подлинности сообщений).

При использовании избыточных кодов исходные данные делятся на блоки из k битов (называются информационными битами). В процессе кодирования каждый k-битный блок данных преобразуется, как было отмечено выше, в блок из n битов (кодовое слово). Число k часто называется размерностью кода. Таким образом, к каждому блоку данных в процессе кодирования присоединяются r = n – k битов, которые называют избыточными битами (redundant bits), битами четности (parity bits) или контрольными битами (check bits); новой информации они не несут.

Для обозначения описанного кода обычно пользуются записью (n, k) и говорят, что данный код использует n символов для передачи (хранения) k символов сообщения. Отношение числа битов данных к общему числу битов k/n именуется степенью кодирования (code rate) – доля кода, которая приходится на полезную информацию. Еще одним важным параметром кода является расстояние Хемминга (d), которое показывает, что два кодовых слова различаются по крайней мере в d позициях.

Влияние на время передачи:

* Увеличение n = k + r приводит к росту объема передаваемых данных.
* Для сохранения скорости передачи требуется большая пропускная способность канала.
* Однако снижается вероятность повторной передачи из-за ошибок, что компенсирует накладные расходы.

Надежность системы (способность выполнять функции без сбоев):

* Избыточное кодирование повышает надежность за счет коррекции ошибок.
* Например, ECC-память исправляет 1-битные ошибки, предотвращая сбои ПО.

Достоверность результата (правильность выходных данных):

* Контрольные биты позволяют отличить корректные данные от искаженных.
* В системах реального времени (авионика, медоборудование) это критично.

Типы ошибок, которые помогает устранить:

* Аппаратные сбои (помехи в сети, повреждение носителей).
* Программные ошибки (случайная модификация данных в ОЗУ).

# Практическое задание

1. На основе информационного сообщения, представленного символами русского/английского алфавитов, служебными символами и цифрами, содержащегося в некотором текстовом файле, сформировать информационное сообщение в двоичном виде; длина сообщения в бинарном виде должна быть не менее 16 символов. Для выполнения этого задания можно использовать коды ASCII символов алфавита либо результаты лабораторной работы № 3.

Исходное сообщение:

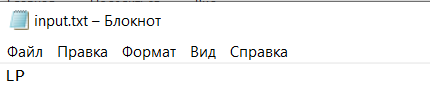


Рисунок 1.1 – Оригинальное сообщение в файле

Функции для считывания из файла:

|  |
| --- |
| namespace Lab4  {  class FileReader  {  public static string ReadTextFromFile(string filePath)  {  try  {  FileInfo fileInfo = new FileInfo(filePath);  return File.ReadAllText(filePath);  }  catch (Exception ex)  {  Console.WriteLine($"Ошибка при чтении файла: {ex.Message}");  return string.Empty;  }  }  public static int[] TextToBinaryArray(string text)  {  List<int> binaryArray = new List<int>();  foreach (char c in text)  {  int asciiValue = (int)c;  string binaryValue = Convert.ToString(asciiValue, 2).PadLeft(8, '0');  foreach (char bit in binaryValue)  {  binaryArray.Add(int.Parse(bit.ToString()));  }  }  return binaryArray.ToArray();  }  }  } |

Листинг 1.1 – Метод считывания из файла

Результат преобразования:

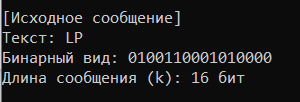


Рисунок 1.2 – Результат преобразования сообщения

Для примера рассмотрим следующие данные:

* Исходные данные (двоичные): Xk = 1101.
* Количество информационных бит: k = 4.

1. Для полученного информационного слова построить проверочную матрицу Хемминга (значение минимального кодового расстояния согласовать с преподавателем).

После получения информационного слова используем следующий метод для построения проверочной матрицы:

|  |
| --- |
| public static int[,]? GenerateMatrix(int k)  {  Console.WriteLine("Длина информационного слова: " + k);  int r = CalculateRedundantBits(k);  Console.WriteLine("Длина избыточного слова: " + r);  int n = k + r;  Console.WriteLine("Длина кодового слова: " + n);  int[,] iMatrix = CreateIMatrix(r);  int[,]? pMatrix = CreatePMatrix(k, r);  if (pMatrix == null) return null;  int[,] Matrix = ConcatenateMatrices(pMatrix, iMatrix);  return Matrix;  } |

Листинг 2.1 – Генерация матрицы

В методе используются следующие функции:

1. Подсчёт длины информационного слова:

|  |
| --- |
| private static int CalculateRedundantBits(int k)  {  return (int)Math.Ceiling(Math.Log2(k + Math.Ceiling(Math.Log2(k)) + 1));  } |

Листинг 2.2 – Вычисление избыточных символов

1. Генерация единичной матрицы:

|  |
| --- |
| public static int[,] CreateIMatrix(int r)  {  if (r <= 0)  {  throw new ArgumentException("Размер матрицы должен быть положительным числом");  }  int[,] identityMatrix = new int[r, r];  Console.WriteLine($"Матрица I:");  for (int i = 0; i < r; i++)  {  for (int j = 0; j < r; j++)  {  if (i == j) identityMatrix[i, j] = 1;  else identityMatrix[i, j] = 0;  Console.Write(identityMatrix[i, j] + " ");  }  Console.WriteLine();  }  return identityMatrix;  } |

Листинг 2.3 – Генерация единичной матрицы

1. Генерация порождающей матрицы:

|  |
| --- |
| private static int[,]? CreatePMatrix(int k, int r)  {  int totalColumns = 0;  for (int p = 2; p <= r; p++)  {  totalColumns += Combinations(r, p);  }  Console.WriteLine($"Общее количество возможных столбцов: {totalColumns}");  if (totalColumns < k)  {  Console.WriteLine("Ошибка: количество возможных столбцов меньше k");  return null;  }  var allColumns = GenerateAllColumns(r);  var selectedColumns = SelectRandomColumns(allColumns, k);  int[,] matrix = new int[r, k];  Console.WriteLine($"Матрица P:");  for (int i = 0; i < r; i++)  {  for (int j = 0; j < k; j++)  {  matrix[i, j] = selectedColumns[j][i];  Console.Write(matrix[i, j] + " ");  }  Console.WriteLine();  }  return matrix;  } |

Листинг 2.4 – Генерация матрицы

Здесь мы используем функции для генерации битовых последовательностей с весом более 2:

|  |
| --- |
| static int Combinations(int n, int k)  {  if (k > n) return 0;  if (k == 0 || k == n) return 1;  int result = 1;  for (int i = 1; i <= k; i++)  {  result = result \* (n - k + i) / i;  }  return result;  }  static List<int[]> SelectRandomColumns(List<int[]> allColumns, int k)  {  var random = new Random();  return allColumns.OrderBy(x => random.Next()).Take(k).ToList();  }  static List<int[]> GenerateAllColumns(int r)  {  var columns = new List<int[]>();  for (int p = 2; p <= r; p++)  {  var combinations = GenerateCombinations(r, p);  foreach (var combination in combinations)  {  int[] column = new int[r];  foreach (int index in combination)  {  column[index] = 1;  }  columns.Add(column);  }  }  return columns;  }  static IEnumerable<int[]> GenerateCombinations(int n, int k)  {  int[] result = new int[k];  Stack<int> stack = new Stack<int>();  stack.Push(0);  while (stack.Count > 0)  {  int index = stack.Count - 1;  int value = stack.Pop();  while (value < n)  {  result[index++] = value++;  stack.Push(value);  if (index == k)  {  yield return result.ToArray();  break;  }  }  }  } |

Листинг 2.5 – Генерация битовых последовательностей

Результатом является:

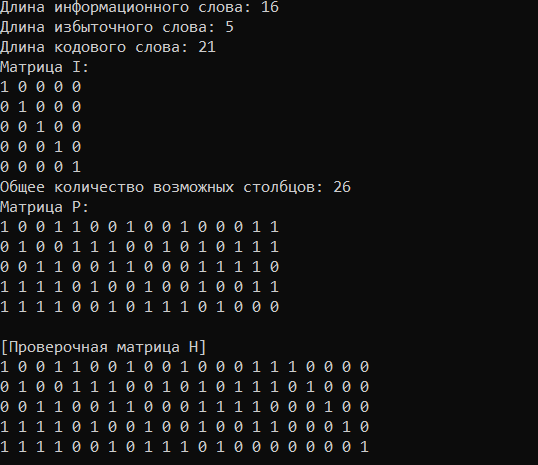


Рисунок 1.5 – Результат кодирования

Нам нужно достаточное количество проверочных бит (r), чтобы определить позицию одиночной ошибки во всём кодовом слове (n бит) или указать на отсутствие ошибки. Общее количество бит в кодовом слове равно:  
n = k + r.

Условие: 2^r ≥ n + 1 (что равносильно 2^r ≥ k + r + 1).  
 Проверим значения для r:  
 Если r = 3: 2^3 ≥ 4 + 3 + 1 ⇒ 8 ≥ 8 (Истина)  
 Следовательно, нам нужно r = 3 проверочных бита.  
 Общая длина кодового слова будет:  
 n = k + r = 4 + 3 = 7 бит.

Проверочная матрица имеет вид H = [P | Ir],  
где Ir — это единичная матрица размера 3 x 3,  
а P — это матрица размера 3 x 4. Столбцы матрицы P выбираются из всех возможных 3-битных векторов с весом (числом единиц) не менее 2. Таких столбцов должно быть ровно 4.

Возможные 3-битные векторы с весом ≥ 2:  
[011], [101], [110], [111]

Составим матрицу P:  
[ 0 1 1 1 ]  
[ 1 0 1 1 ]  
[ 1 1 0 1 ]

Единичная матрица Ir (размер 3 x 3):  
[ 1 0 0 ]  
[ 0 1 0 ]  
[ 0 0 1 ]

Итоговая проверочная матрица H = [P | I]:  
[ 0 1 1 1 | 1 0 0 ]  
[ 1 0 1 1 | 0 1 0 ]  
[ 1 1 0 1 | 0 0 1 ]

1. Используя построенную матрицу, вычислить избыточные символы (слово Xr).

Используем следующую функцию для вычисления избыточных символов, а также получения полного кодового слова:

|  |
| --- |
| public static int[] SolveHamming(int[,] h, int[] knownBits, out int[] unknownBits)  {  int k = knownBits.Length;  int r = CalculateRedundantBits(k);  unknownBits = new int[r];  if (h.GetLength(0) != r || h.GetLength(1) != r + k || knownBits.Length != k)  {  throw new ArgumentException("Неверные размерности матриц или вектора известных битов.");  }  int[] x = new int[r + k];  for (int i = 0; i < k; i++)  {  x[i] = knownBits[i];  }  for (int i = 0; i < r; i++)  {  int x\_ri = 0;  for (int j = 0; j < k; j++)  {  x\_ri ^= h[i, j] \* x[j];  }  unknownBits[i] = x\_ri;  x[k + i] = x\_ri;  }  return x;  } |

Листинг 3.1 – Решение Хэмминга

Получем следующий результат:

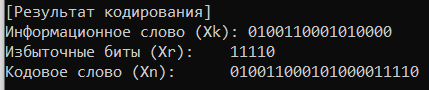


Рисунок 1.5 – Результат кодирования

Для систематического кода, где H = [P | I], кодовое слово Xₙ имеет структуру Xₙ = [Xₖ | Xᵣ]. То есть, информационные биты идут первыми, за ними проверочные.

Основное уравнение линейного кода:  
H · Xₙᵗ = 0,  
где Xₙᵗ — это кодовое слово в виде вектора-столбца.

Пусть Xₖ = 1010. Тогда Xₙᵗ имеет следующий вид:

markdown

КопироватьРедактировать

[ 0 1 1 1 | 1 0 0 ] [ 1 ]

[ 1 0 1 1 | 0 1 0 ] \* [ 0 ] = 0

[ 1 1 0 1 | 0 0 1 ] [ 1 ] [ 0 ]

[ xr₁ ]

[ xr₂ ]

[ xr₃ ]

В результате решения получаем Xᵣ = [1 1 0].

Формируем полное кодовое слово:  
Xₙ = [Xₖ | Xᵣ] = [1 0 1 0 | 1 1 0]

1. Принять исходное слово со следующим числом ошибок: 0, 1, 2. Позиция ошибки определяется (генерируется) случайным образом.

Создаём три варианта принятого слова и вносим ошибки:

|  |
| --- |
| int[] Yn1 = new int[Xn.Length];  int[] Yn2 = new int[Xn.Length];  int[] Yn3 = new int[Xn.Length];  Xn.CopyTo(Yn1, 0);  Xn.CopyTo(Yn2, 0);  Xn.CopyTo(Yn3, 0);  Hemming.ChangeValue(Yn2); // 1 ошибка  Hemming.ChangeValue(Yn3); // 2 ошибки  Hemming.ChangeValue(Yn3); |

Листинг 4.1 – Вносим ошибки

Пусть при передаче произошла ошибка в 3-й позиции кодового слова Xₙ.  
 Переданное слово: Xₙ = 1011010  
 Принятое слово: Yₙ = 1001010 (3-й бит инвертирован: 1 → 0)

Вычислим синдром ошибки по формуле:  
S = H × Yₙᵗ

Матрица H:

css

КопироватьРедактировать

[ 0 1 1 1 | 1 0 0 ][ 1 0 1 1 | 0 1 0 ][ 1 1 0 1 | 0 0 1 ]

Подставим Yₙ = [1 0 0 1 0 1 0]

1. Для полученного слова Yn = Yk, Yr, используя уже известную проверочную матрицу Хемминга, вновь вычислить избыточные символы (обозначим их Yr’), используя выражение (4.6).
2. Вычислить и проанализировать синдром. В случае, если анализ синдрома показал, что информационное сообщение было передано с ошибкой (или 2 ошибками), сгенерировать унарный вектор ошибки Еn = е1, е2, …, еn и исправить одиночную ошибку, используя формулу (4.7); проанализировать ситуацию при возникновении ошибки в 2 битах.

|  |
| --- |
| public static void CheckYn(int[] Yn, int k, int[,] H)  {  Console.WriteLine("Поступило слово:");  foreach (var x in Yn) Console.Write(x);  Console.WriteLine();  (int[] Yk, int[] Yr) = SplitArray(Yn, k);  Console.WriteLine("Yk:");  foreach (var x in Yk) Console.Write(x);  Console.WriteLine();  Console.WriteLine("Yr:");  foreach (var x in Yr) Console.Write(x);  Console.WriteLine();  Console.WriteLine("Вычисляем Yr':");  int[] Yrq;  int[] Ynq = SolveHamming(H, Yk, out Yrq);  foreach (var x in Yrq) Console.Write(x);  Console.WriteLine();  int[] S = XorVectors(Yr, Yrq);  Console.WriteLine("Синдром:");  foreach (var x in S) Console.Write(x);  Console.WriteLine();  bool isZeroVector = true;  foreach (int element in S)  {  if (element != 0)  {  isZeroVector = false;  break;  }  }  if (isZeroVector)  {  Console.WriteLine("Синдром нулевой, значит ошибок нет");  return;  }  if (S.Length != H.GetLength(0))  {  throw new ArgumentException("Длина вектора должна быть равна количеству строк матрицы.");  }  for (int j = 0; j < H.GetLength(1); j++)  {  bool match = true;  for (int i = 0; i < H.GetLength(0); i++)  {  if (H[i, j] != S[i])  {  match = false;  break;  }  }  if (match)  {  Console.WriteLine($"Найден совпадающий столбец: {j+1}");  int[] E = new int[Yn.Length];  for (var e = 0; e < E.Length; e++) E[e] = 0;  E[j] = 1;  Console.WriteLine($"Вектор ошибки:");  for (var e = 0; e < E.Length; e++) Console.Write(E[e]);  Console.WriteLine();  Console.WriteLine($"Исправленный Yn:");  int[] YnFIX = XorVectors(Yn, E);  for (var i = 0; i < YnFIX.Length; i++) Console.Write(YnFIX[i]);  Console.WriteLine();  return;  }  }  Console.WriteLine("Совпадающих столбцов не найдено");  } |

Листинг 6.1 – Анализ синдрома

Получем следующие результаты:

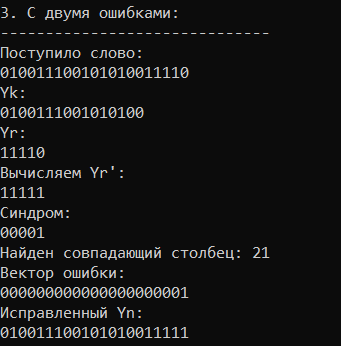
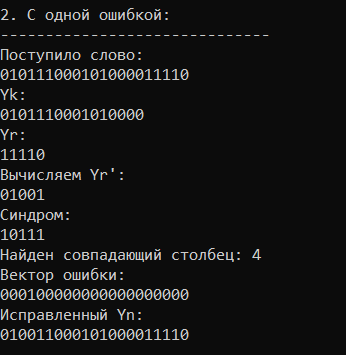
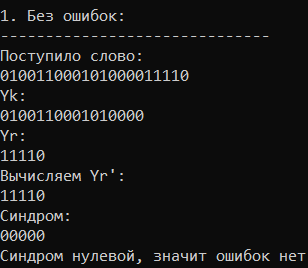


Рисунок 1.5 – Результат кодирования

Получаем принятое слово Yₙ = 1001010.  
Вычислим синдром ошибки S. Синдром вычисляется по формуле:  
S = H × Yₙᵗ (mod 2)

Вектор Yₙ в виде строки: [1 0 0 1 0 1 0]

Подставим в уравнение и произведем умножение:  
S = [1 1 0]

Синдром S не равен нулевому вектору [0 0 0], следовательно, в принятом слове Yₙ есть ошибка.

Поскольку мы рассматриваем одиночную ошибку, синдром S точно совпадает с одним из столбцов матрицы H, и его номер указывает на позицию ошибочного бита.

Сравниваем синдром S = [1 1 0] со столбцами матрицы H — видно, что он совпадает с третьим столбцом. Это означает, что ошибка произошла в 3-й позиции слова Yₙ.

Исправим ошибку: инвертируем бит в 3-й позиции:  
Yₙ = 1 0 0 1 0 1 0  
→ инвертируем 3-й бит (0 → 1)  
→ исправленное слово Yₙ' = 1 0 1 1 0 1 0

# Вывод

В ходе лабораторной работы были углублены теоретические знания в области теории информации, а также разработано программное средство для вычисления энтропии по Шеннону и оценки количества информации для латышского, таджикского и бинарного алфавитов. Были выполнены расчеты, позволяющие проанализировать объем информации в зависимости от типа алфавита. Дополнительно исследовано, как вероятность ошибок влияет на передаваемую информацию. Результаты показали, что формула эффективной энтропии применима исключительно для бинарного алфавита, поскольку только в двоичной системе возможно однозначное изменение символа на противоположный при возникновении ошибки, что делает данный метод неприменимым для многосимвольных алфавитов.