Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Дисциплина «Защита информации и надежность информационных систем»

**Отчёт по лабораторной работе №4**

Избыточное кодирование данных в информационных системах.

Код Хемминга

Студент: Жук С.С.

ФИТ 3 курс 2 группа

Преподаватель: Савельева М.Г.

Минск 2025

**Содержание**

[1 Теоретические сведения 3](#_Toc193741857)

[2 Построение матрицы Хемминга и вычисление избыточных битов 6](#_Toc193741858)

[3 Информационное сообщение с различным количеством ошибок 9](#_Toc193741859)

[3.1 Отсутствие ошибок 10](#_Toc193741860)

[3.2 Одиночная ошибка 11](#_Toc193741861)

[3.3 Двойная ошибка 12](#_Toc193741862)

[Вывод 14](#_Toc193741863)

# **1 Теоретические сведения**

Надежность системы – характеристика способности аппаратно-программного средства выполнить требуемые функции в течение конкретного периода времени.

Достоверность работы системы – свойство, характеризующее истинность конечного выполнения программы, определяемое способностью средств контроля фиксировать правильность или ошибочность работы.

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

Ошибка программы – проявляется в не соответствующем реальному промежуточном или конечном значении вследствие неправильно составленной программы.

Надежность является комплексным свойством, включающим в себя: безотказность, ремонтопригодность, сохраняемость, долговечность.

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

Ремонтопригодность – свойство технического объекта, заключающееся в приспособленности к поддержанию и восстановлению работоспособного состояния путем технического обслуживания, ремонта. Большинство современных цифровых устройств содержат специальные средства, призванные автоматически восстанавливать работоспособность этих объектов при нарушении нормального функционирования. Эти специальные средства контроля называются избыточными.

Сначала осуществляется формирование данных в виде двоичных символов. Затем кодер канала вносит в принятую информационную последовательность некоторую избыточность (данный процесс называется кодированием или помехоустойчивым кодированием), которую декодер может использовать для исправления возникающих при передаче данных по каналу связи ошибок.

Простейшая структурная схема дополнена двумя интересующими нас блоками: кодером (канала), осуществляющим преобразование исходного сообщения (информационного слова) *Xk* (длина сообщения – *k* символов) в избыточное сообщение (кодовое слово) *Xn* длиной *n* символов (*n* > *k*), и декодером (канала).

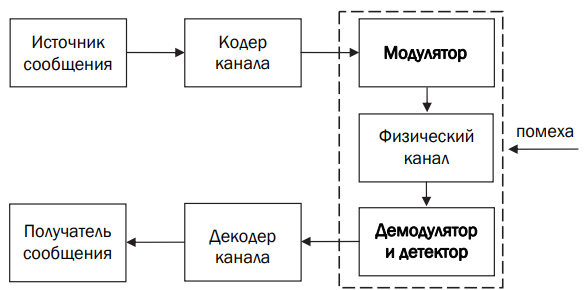


Рисунок 1.1 – Структурная схема системы передачи данных с избыточными средствами аппаратного контроля

Изначальной причиной нарушения нормальной работы цифрового устройства являются технические неисправности, возникающие внутри узлов или блоков устройства либо в каналах связи между ними.

Дефекты или неисправности могут приводить либо к кратковременному нарушению достоверности работы устройства (сбой), либо к полной и окончательной потере достоверности (отказ).

В каждом из этих случаев следствием неисправности являются ошибки в информации (информационные ошибки). Чаще всего причиной ошибок бывают внешние помехи, как это показано на рисунке. Количество таких ошибок (количество ошибочных двоичных символов) принято называть кратностью ошибки. При использовании избыточных кодов исходные данные делятся на блоки из *k* битов (информационные биты). В процессе кодирования каждый *k*-битный блок данных преобразуется, в блок из *n* битов (кодовое слово). Число *k* – размерность 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. Во всех этих простых математических выражениях *d* соответствует минимальному кодовому расстоянию Хемминга.

В теореме Шеннона не говорится о том, как нужно строить необходимые помехоустойчивые коды. Однако в ней указывается на принципиальную возможность кодирования, при котором может быть обеспечена сколь угодно высокая надежность передачи.

Код Хемминга относится к классу линейных блочных кодов.

Линейные блочные коды – 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* – порождающая матрица линейного корректирующего кода в приведенно-ступенчатой форме. Кодовые слова являются линейными комбинациями строк матрицы *G* (кроме слова, состоящего из нулевых символов).

Кодирование заключается в умножении вектора сообщения *Хk* длиной *k* на порождающую матрицу по правилам матричного умножения (все операции выполняются по модулю 2). Очевидно, что при этом первые *k* символов кодового слова равны соответствующим символам сообщения, а последние *r* символов образуются как линейные комбинации первых.

Для всякой порождающей матрицы *G* существует матрица *Н* размерности *r*×*n*, задающая базис нулевого пространства кода.

Справедливо также

(1.1)

В последнем выражении символ «*T*» означает транспонирование, а *Xn* = *x*1, *x*2, …, *xn*.

Матрица Н, называемая проверочной, равна

(1.2)

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. В последнем выражении *I* – единичная матрица порядка *r* (*r*×*r*).

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

(1.3)

где *Yn* = *y*1, *y*2, …, *yn* – принятый вектор (сообщение на выходе канала), полученный после передачи либо считывания из памяти. Вектор *Yn* обычно представляют в следующем виде:

(1.4)

где *Еn* = *е*1, *е*2, …, *еn* – вектор ошибки.

Синдром – результат проверки четности, выполняемой над сообщением *Yn* для определения его принадлежности заданному набору кодовых слов. При положительном результате проверки синдром *S* равен 0, т. е. *Yn* = *Хn*. Если *Yn* содержит ошибки, которые можно исправить, то синдром имеет определенное ненулевое значение, что позволяет обнаружить и исправить конкретную ошибочную комбинацию.

Важно запомнить, что в силу выражений (1.1) – (1.4) ненулевой синдром всегда равен сумме по модулю 2 тех вектор-столбцов матрицы *Н*, номера которых соответствуют номерам ошибочных битов в слове *Yn*.

Корректирующая способность кода Хемминга с *d*min = 3 может быть увеличена введением дополнительной проверки на четность: минимальное кодовое расстояние такого кода будет равно 4: *d*min = 4. Такой код может исправлять все единичные ошибки с одновременным обнаружением всех двойных в анализируемом кодовом слове. При этом нужно помнить, что вид этой матрицы не соответствует ее каноническому представлению, поскольку во всех столбцах единичной матрицы, кроме последнего, будет по 2 единицы. Для придания матрице канонического вида необходимо сложить посимвольно все строки между собой и результат сложения записать в последнюю строку (под горизонтальной линией)

# **2 Построение матрицы Хемминга и вычисление избыточных битов**

Матрица Хемминга – способ представления кодов Хемминга для обнаружения и исправления ошибок в процессе передачи данных. Код Хемминга – метод, который добавляет избыточные биты в передаваемое сообщение, чтобы можно было обнаружить и исправить одиночные ошибки (ошибки, вызванные повреждением одного бита в передаче).

Для начала необходимо прочитать данные из файла. После каждый символ текста преобразуется в его двоичное представление (в формате ASCII) с помощью функции, описанной ниже, toBinary. Переменные *k* и *r* хранят количество информационных и избыточных битов соответственно, *n* – общая длина кодового слова, *d* – минимальное расстояние Хемминга. Код представлен в листинге 2.1.

|  |
| --- |
| const inputData = fs.readFileSync(‘Lab4.txt’, ‘ascii’); const binaryMessage = *mylib*.toBinary(inputData); const k = binaryMessage.length; const r = *Math*.floor(*Math*.log2(k) + 1); const n = r + k; const d = 4;  const hammingMatrix = *mylib*.getHammingMatrix(k, r); |

Листинг 2.1 – Вывод в главной функции main

Далее рассмотрим две функции. Первая функция преобразует строку в её бинарное представление. В этой функции необходимо разбить строку на массив символов, затем для каждого символа вычисляем его код ASCII, конвертируем его в двоичную строку и дополнительно дополняем её до 8 битов. Результат работы функции – это строка, которая представляет собой бинарное представление всех символов строки. Вторая функция предназначена для того, чтобы принять массив двоичных значений (бит) и объединить все элементы массива в одну строку. Результатом работы этой функции является строка, представляющая собой последовательность битов, без промежутков между ними. Реализация функций представлена в листинге 2.2.

|  |
| --- |
| toBinary: (text) => {  return text  .split('')   .map(char => char.charCodeAt(0).toString(2).padStart(8, '0'))  .join(''); }  getBitsStr: (bits) => bits.join('') |

Листинг 2.2 – Функции перевода текста в бинарное представление и объединения

Следующим шагом будет описана функция, которая позволяет построить матрицу Хемминга. В этой функции создается матрица, состоящая из *r* строк и *k* + *r* столбцов. Сначала заполняем все строки матрицы нулями. Затем для каждого столбца, соответствующего первой части матрицы, мы определяем значения на основе битов позиции. Для каждой позиции, начиная с 1, проверяется, установлен ли *i*-й бит, и в зависимости от этого в матрице устанавливается значение 1 или 0. В последнем шаге для каждого столбца, начиная с *k*, заполняются единицы. Результатом работы функции является матрица Хэмминга размером *r* × (*k* + *r*). Функция продемонстрирована в листинге 2.3.

|  |
| --- |
| getHammingMatrix: (k, r) => {  const matrix = [];  for (let i = 0; i < r; i++) {  matrix[i] = new *Array*(k + r).fill(0);  }  for (let j = 0; j < k; j++) {  const pos = j + 1;   for (let i = 0; i < r; i++) {  matrix[i][j] = (pos & (1 << i)) ? 1 : 0;  }  }  for (let i = 0; i < r; i++) {  matrix[i][k + i] = 1;  }  return matrix; } |

Листинг 2.3 – Функция для построения матрицы Хемминга

Далее опишем функцию, которая реализует вычисление избыточных бит для переданного бинарного слова и матрицы Хэмминга. В этой функции создается массив *Xr* размером *r*, который инициализируется нулями. Затем для каждого из избыточных битов (строк матрицы Хэмминга) выполняется операция XOR с каждым битом из исходного слова, умноженным на соответствующий элемент матрицы. Для каждой строки матрицы выполняется побитовая операция XOR с битами сообщения, и результат записывается в массив *Xr*. В результате работы функции возвращается массив избыточных битов, который может быть использован для формирования контрольных битов в коде Хэмминга. Программная реализация функции показана в листинге 2.4.

|  |
| --- |
| calcRedundantBits: (hammingMatrix, binaryMessage, k, r) => {  const Xr = new *Array*(r).fill(0);  for (let i = 0; i < r; i++) {  for (let j = 0; j < k; j++) {  Xr[i] ^= parseInt(binaryMessage[j]) & hammingMatrix[i][j];  }  }  return Xr; } |

Листинг 2.4 – Функция вычисления избыточных битов

Эти фрагменты кода иллюстрируют часть алгоритма кодирования с использованием кода Хэмминга для защиты данных от ошибок при их передаче. Код Хэмминга добавляет проверочные биты в исходное сообщение, чтобы оно стало исправляемым в случае ошибок.

В результате получим построенную матрицу Хемминга с вычисленными избыточными битами. Результат показан на рисунке 2.1.

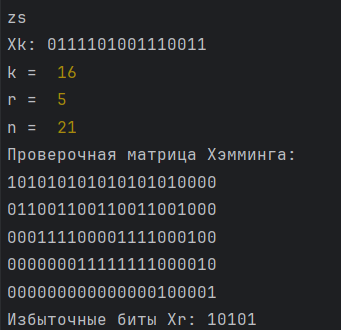


Рисунок 2.1 – Вывод результатов

Таким образом, создается структура данных, которая включает как информационные, так и проверочные биты. Этот процесс позволяет обеспечить защиту передаваемых данных и гарантировать возможность исправления ошибок в случае их возникновения в процессе передачи. Полученная матрица Хэмминга и вычисленные избыточные биты будут использованы для кодирования и проверки целостности данных, что делает код Хэмминга эффективным инструментом для защиты информации.

# **3 Информационное сообщение с различным количеством ошибок**

В данном разделе будут рассмотрены случаи, когда в передаваемом информационном сообщении могут возникать ошибки. Будут показаны три сценария: наличие 0, 1 и 2 ошибок в переданном сообщении. Для каждого из этих случаев будет проанализирован процесс обнаружения ошибок с использованием кода Хэмминга.

Для начала необходимо выполнить цикл, в котором будет моделироваться добавление ошибок в бинарное информационное слово и их исправление с использованием кода Хэмминга. В цикле переменная numOfErrors будет изменяться от 0 до 2, что позволяет обработать информационные слова с разным количеством ошибок. На каждом шаге цикла будет генерироваться бинарное информационное сообщение с определенным количеством ошибок. Далее вычисляются избыточные биты для полученного информационного слова с ошибками.

Затем рассчитывается синдром ошибки, который поможет определить, есть ли ошибки в сообщении. Если синдром не равен нулю, это значит, что в сообщении есть ошибки. В этом случае генерируется вектор ошибок, который используется для исправления ошибки в сообщении. После исправления ошибок, исправленное бинарное сообщение конвертируется обратно в текст. Если синдром равен нулю, то ошибок в сообщении не найдено. Реализация цикла представлена в листинге 3.1.

|  |
| --- |
| for (let numOfErrors = 0; numOfErrors < 3; numOfErrors++) {  const binaryWithError = *mylib*.genErr(binaryMessage, numOfErrors);  *console*.log(`Сообщение с ${numOfErrors} ошибками: ${binaryWithError}`);  const XrWithError = *mylib*.calcRedundantBits(hammingMatrix, binaryWithError, k, r);  *console*.log(`Избыточные биты Xr с ${numOfErrors} ошибками: ${*mylib*.getBitsStr(XrWithError)}`);  const S = *mylib*.calcSyndrome(Xr, XrWithError, r);  *console*.log(`Синдром: ${*mylib*.getBitsStr(S)}`);   if (S.some(bit => bit !== 0)) {  const En = *mylib*.genErrVector(S, hammingMatrix);  *console*.log(`Вектор ошибок: ${*mylib*.getBitsStr(En)}`);  const correctedBinary = *mylib*.correctErr(binaryWithError, En);  *console*.log(`Исправленное сообщение: ${correctedBinary}`);  const correctedText = *mylib*.fromBinary(correctedBinary);  *console*.log(`Исправленное сообщение в ascii: ${correctedText}`);  } else {  *console*.log('Нет ошибок для исправления');  } } |

Листинг 3.1 – Цикл для декодирования с различными ошибками

Этот цикл моделирует исправление ошибок в бинарных сообщениях и их восстановление с использованием избыточных битов и кодов Хэмминга.

Также здесь опишем функцию, которая добавляет случайные ошибки в бинарное сообщение. Сначала строка сообщения преобразуется в массив символов. Затем, в цикле, выполняется добавление ошибок: случайным образом выбирается позиция в массиве, и значение в этой позиции инвертируется. Этот процесс повторяется заданное количество раз (в зависимости от значения numOfErrors). После того как ошибки добавлены, массив снова преобразуется в строку. Функция продемонстрирована в листинге 3.2.

|  |
| --- |
| genErr: (binaryMessage, numOfErrors) => {  let binaryWithError = binaryMessage.split('');  for (let i = 0; i < numOfErrors; i++) {  const pos = *Math*.floor(*Math*.random() \* binaryWithError.length);  binaryWithError[pos] = binaryWithError[pos] === '0' ? '1' : '0';  }  return binaryWithError.join(''); } |

Листинг 3.2 – Функция для добавления случайных ошибок

## **3.1 Отсутствие ошибок**

Когда в процессе передачи данных не произошло никаких ошибок, переданное сообщение остается целым и корректным. В случае, если мы используем код Хэмминга, проверочные биты, которые были добавлены в процессе кодирования, будут использоваться для проверки корректности данных. При проверке таких данных с помощью алгоритма декодирования Хэмминга синдром будет равен нулю, что свидетельствует о том, что ошибок в сообщении нет, и, соответственно, исправлять нечего.

Следующим шагом будет описана функция, которая вычисляет синдром ошибки. Сначала создается пустой массив *S*, который будет хранить значения синдрома. Затем, в цикле от 0 до *r*, для каждого избыточного бита происходит вычисление синдрома: каждый бит из массива *Xr* сравнивается с соответствующим битом из массива XrWithError с использованием XOR. Этот процесс позволяет выявить ошибки в сообщении. Реализация функции показана в листинге 3.3.

|  |
| --- |
| calcSyndrome: (Xr, XrWithError, r) => {  const S = [];  for (let i = 0; i < r; i++) {  S[i] = Xr[i] ^ XrWithError[i];  }  return S; } |

Листинг 3.3 – Функция для нахождения синдрома ошибки

Так как кодовое слово без ошибок и с теми же избыточными битами, в результате получим нулевой синдром, что свидетельствует о безошибочной передаче по каналу. Результат показан на рисунке 3.1.

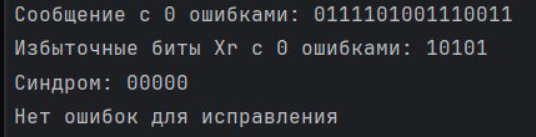


Рисунок 3.1 – Вывод результатов для безошибочного информационного слова

## **3.2 Одиночная ошибка**

Далее реализуем функцию, которая генерирует вектор ошибок на основе синдрома и матрицы Хэмминга. Сначала создается массив *En*, который будет представлять вектор ошибок, и инициализируется значениями 0. Длина этого массива равна количеству столбцов в матрице Хэмминга. Далее, синдром *S*, который является массивом битов, преобразуется в десятичное число. Это значение представляет собой позицию бита ошибки в матрице Хэмминга. Если значение синдрома больше 0 и меньше или равно длине строки в матрице Хэмминга, то в соответствующем индексе массива *En* устанавливается 1, что указывает на местоположение ошибки. Функция продемонстрирована в листинге 3.4.

|  |
| --- |
| genErrVector: (S, hammingMatrix) => {  const En = new *Array*(hammingMatrix[0].length).fill(0);  const syndromeValue = parseInt(S.join(''), 2);   if (syndromeValue > 0 && syndromeValue <= hammingMatrix[0].length) {  En[syndromeValue - 1] = 1;  }  return En; } |

Листинг 3.4 – Функция для вычисления вектора ошибок

Далее опишем функцию, которая преобразует бинарное сообщение в текстовое представление. Сначала используется регулярное выражение, которое разбивает строку на блоки по 8 бит. Это нужно для того, чтобы каждый блок представлял собой один символ в кодировке ASCII. Затем, для каждого блока, вызывается метод, который преобразует бинарную строку в десятичное число. Полученное число передается в функцию, которая возвращает символ, соответствующий этому числу в ASCII. После того как для каждого байта получен символ, результат собирается в строку, который соединяет все символы в единую строку. Программная реализация продемонстрирована в листинге 3.5.

|  |
| --- |
| fromBinary: (binary) => {  return binary  .match(/.{8}/g)  .map(byte => String.fromCharCode(parseInt(byte, 2)))  .join(''); } |

Листинг 3.5 – Функция для преобразования в текстовое представление

Следующим шагом будет описана функция, которая исправляет ошибки в бинарном сообщении на основе вектора ошибок. Сначала строка с бинарным сообщением преобразуется в массив символов. Далее, в цикле по длине вектора ошибок *En* проверяется каждый элемент этого вектора. Если на позиции *i* в векторе ошибок стоит значение 1, то происходит инверсия бита в бинарном сообщении. Этот процесс повторяется для всех позиций в векторе ошибок. После исправления всех ошибок, массив снова преобразуется в строку. Реализация функции показана в листинге 3.6.

|  |
| --- |
| correctErr: (binaryWithError, En, r) => {  const corrected = binaryWithError.split('');  for (let i = 0; i < En.length; i++) {  if (En[i] === 1 && i < corrected.length) {  corrected[i] = corrected[i] === '0' ? '1' : '0';  }  }  return corrected.join(''); } |

Листинг 3.6 – Функция для исправления ошибки

Так как кодовое слово с ошибкой в 14 бите, в результате получим избыточные биты, ненулевой синдром, который соответствует столбцу в матрице, вектор ошибок, исправленное сообщение в двоичном и символьном представлении. Результат показан на рисунке 3.2.

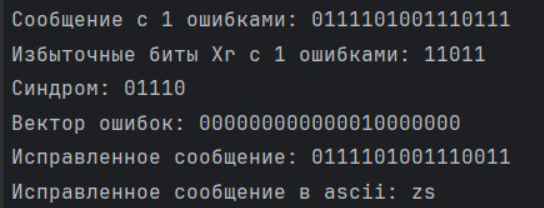


Рисунок 3.2 – Вывод результатов для информационного слова с 1 ошибкой

## **3.3 Двойная ошибка**

Когда в процессе передачи данных не произошло никаких ошибок, переданное сообщение остается целым и корректным. В случае, если мы используем код Хэмминга, проверочные биты, которые были добавлены в процессе кодирования, будут использоваться для проверки корректности данных. При проверке таких данных с помощью алгоритма декодирования Хэмминга все проверочные биты будут равны нулю, что свидетельствует о том, что ошибок в сообщении нет.

Так как кодовое слово с ошибками во 2 и 12 битах, в результате получим избыточные биты, ненулевой синдром, который соответствует столбцу в матрице, вектор ошибок, исправленное сообщение в двоичном и символьном представлении. Результат показан на рисунке 3.3.

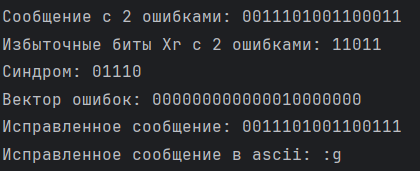


Рисунок 3.3 – Вывод результатов для информационного слова с 2 ошибками

Можно заметить, что синдром равняется сумме по модулю 2 столбцов 6 и 15. В векторе ошибки устанавливаем 1 в том бите, где столбец в матрице соответствует посчитанному синдрому. В результате получаем дополнительную ошибку, потому что система так исправила.

# **Вывод**

В результате выполнения лабораторной работы было изучено применение метода Хемминга для обеспечения корректности передачи данных в информационных системах. Текст, содержащий символы русского и английского алфавита и цифры, был преобразован в двоичный вид с использованием кодировки ASCII, что стало основой для дальнейших вычислений. Построенная проверочная матрица Хемминга обеспечивала возможность исправления ошибок в передаваемом сообщении.

В ходе работы были вычислены избыточные символы, что позволило защитить данные от ошибок, возникающих при передаче. Также было проведено моделирование случайных ошибок, что позволило протестировать алгоритм на исправление одиночных и двойных ошибок. Использование матрицы Хемминга позволило успешно исправить одиночные ошибки, однако исправление двойных ошибок оказалось невозможным.

В результате лабораторной работы были на практике продемонстрированы важные принципы кодирования с исправлением ошибок, что является основой для обеспечения надежности передачи данных в телекоммуникационных и информационных системах.