Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Дисциплина «Защита информации и надежность информационных систем»

**Отчёт по лабораторной работе №6**

Избыточное кодирование данных в информационных системах.

Циклические коды

Студент: Жук С.С.

ФИТ 3 курс 2 группа

Преподаватель: Савельева М.Г.

Минск 2025

**Содержание**

[1 Теоретические сведения 3](#_Toc194396934)

[2 Порождающая и проверочная матрицы канонической формы 5](#_Toc194396935)

[3 Информационное сообщение с различным количеством ошибок 10](#_Toc194396936)

[3.1 Отсутствие ошибок 11](#_Toc194396937)

[3.2 Одиночная ошибка 12](#_Toc194396938)

[3.3 Двойная ошибка 13](#_Toc194396939)

[Вывод 14](#_Toc194396940)

# **1 Теоретические сведения**

Циклические коды – это семейство помехоустойчивых кодов, одной из разновидностей которых являются коды Хемминга.

Основные свойства ЦК:

* относятся к классу линейных, систематических;
* сумма по модулю 2 двух разрешенных кодовых комбинаций дает также разрешенную кодовую комбинацию;
* каждый вектор (кодовое слово), получаемый из исходного кодового вектора путем циклической перестановки его символов, также является разрешенным кодовым вектором; к примеру, если кодовое слово имеет следующий вид: 1101100, то разрешенной кодовой комбинацией будет и такая: 0110110;
* при простейшей циклической перестановке символы кодового слова перемещаются слева направо на одну позицию, как в приведенном примере;
* поскольку к числу разрешенных кодовых комбинаций ЦК относится нулевая комбинация 000...00, то минимальное кодовое расстояние *d*min для ЦК определяется минимальным весом разрешенной кодовой комбинации;
* циклический код не обнаруживает только такие искаженные помехами кодовые комбинации, которые приводят к появлению на стороне приема других разрешенных комбинаций этого кода;
* в основе описания и использования ЦК лежит полином или многочлен некоторой переменной (обычно *Х*).

Операции кодирования и декодирования циклических кодов сводятся к известным процедурам умножения и деления двоичных чисел, либо соответствующих этим числам полиномов. Действия с кодовыми словами производятся по правилам арифметики по модулю 2. Вычитание равносильно сложению.

ЦК составляют множество многочленов {*Вj*(*X*)} степени *r* (*r* − число проверочных символов в кодовом слове), кратных порождающему (образующему) полиному *G*(*Х*) степени *r*, который должен быть делителем бинома *Xn* + 1, т. е. остаток после деления бинома на *G*(*X*) должен быть нулевым.

Формирование разрешенных кодовых комбинаций ЦК *Вj*(*X*) основано на предварительном выборе порождающего (генераторного или образующего) полинома *G*(*Х*), который обладает важным отличительным признаком: все комбинации *Вj*(*X*) делятся на порождающий полином *G*(*Х*) без остатка:

(1.1)

здесь *Вj*(*X*) = *Xn* – кодовое слово; *Aj*(*X*) = *Xk* – информационное слово.

Степень порождающего полинома определяет число проверочных символов: *r* = *n* – *k*. Из этого свойства следует простой способ формирования разрешeнных кодовых слов ЦК − умножение информационного слова *A*(*X*) на порождающий полином *G*(*X*):

(1.2)

Порождающими могут быть только такие полиномы, которые являются делителями двучлена (бинома) *X* z+ 1:

(1.3)

при нулевом остатке: *R*(*X*) = 0.

Для *k* = 4 существуют только два порождающих полинома: *х*3 + *х* + 1 (1011), и *х*3 + *х*2 + 1 (1101). Эти полиномы относятся к двойственным: обратная запись (справа налево) одного в бинарной форме дает другой.

Таким образом, в основе построения ЦК лежит операция деления передаваемой кодовой комбинации на порождающий неприводимый полином степени *r*. Остаток *R*(*X*) от деления используется при формировании проверочных разрядов. При декодировании принятой *n*-разрядной кодовой комбинации (*Yn*) опять производится ее деление на порождающий (производящий, образующий) полином.

Синдромом ошибки в этих кодах является наличие остатка от деления принятой кодовой комбинации на порождающий полином. Если синдром равен нулю, то считается, что ошибок нет. В противном случае с помощью полученного синдрома можно определить номер разряда принятой кодовой комбинации, в котором произошла ошибка, и исправить ее примерно по той же схеме, которую мы использовали для кода Хемминга.

При этом следует обратить внимание на важную деталь: умножение полинома на х приводит к сдвигу членов полинома на один разряд влево, а при умножении на *хr* – на *r* разрядов влево с заменой r младших разрядов полинома на нули. Деление же полинома на *х* приводит к соответствующему сдвигу членов полинома вправо с уменьшением показателей членов на 1. Такой сдвиг требует дописать справа *r* проверочных символов к исходной кодовой комбинации *Аi*(*Х*) после умножения ее на *хr*.

Кодирование информационного слова. Деление полиномов позволяет представить кодовые слова в виде блочного кода, т. е. информационных *Хk* (*Аi*(*Х*)) и проверочных *Хr* (*Ri*(X)) символов. Поскольку число последних равно r, то для компактной их записи в младшие разряды кодового слова надо предварительно к кодируемому (информационному) слову *Аi*(*Х*) справа дописать r нулевых символов.

Декодирование принятого сообщения по синдрому. Основная операция: принятое кодовое слово (*Yn*) нужно поделить на порождающий полином, который использовался при кодировании. Если *Yn* принадлежит коду, т. е. слово не искажено помехами, то остаток от деления (синдром) будет нулевым.

Всякому ненулевому синдрому соответствует определенное расположение ошибок: синдром для ЦК имеет те же свойства, что и для кода Хемминга.

Декодирование синдрома и исправление ошибки в принятом сообщении. Декодирование ненулевого синдрома имеет целью определение ошибочного бита в принятом сообщении или, иначе говоря, определение вектора *Еn*. Поиск ошибочного бита будем производить через поиск соответствия между синдромом и проверочной матрицей кода.

Ненулевой синдром всегда равен сумме по модулю 2 тех вектор-столбцов матрицы *Н*, номера которых соответствуют номерам ошибочных битов в слове *Yn*.

# **2 Порождающая и проверочная матрицы канонической формы**

Циклические коды – это особый класс линейных блоковых кодов, где сдвиг всех битов вправо с переносом последнего в начало даёт допустимый код. Одним из эффективных способов обнаружения и исправления ошибок при передаче данных является использование циклических кодов с заданным порождающим полиномом.

Для построения генераторной матрицы циклического кода используется порождающий полином, представленный в виде строки битов. На основе этого полинома формируется порождающая матрица размером *k* × *n*, где первая строка содержит сам полином, выровненный по левому краю. Каждая последующая строка получается циклическим сдвигом предыдущей влево. Такой способ построения соответствует свойству цикличности кода. Кодовые слова можно безопасно передавать по каналу связи с возможностью обнаружения и исправления одиночных ошибок, а также некоторых множественных.

Для начала опишем переменные: *k* и *r* хранят количество информационных и избыточных битов соответственно, *n* – общая длина кодового слова, *g* – количество битов полинома. Код представлен в листинге 2.1.

|  |
| --- |
| const g = '101111'; const message = '001011'; const k = message.length; const r = g.length - 1; const n = k + r;  const genMatrix = makeGenMatrix(g, k, n); |

Листинг 2.1 – Вывод в главной функции main

Следующим шагом будет описана функция, которая позволяет построить порождающую матрицу для циклического кода. Вначале вся матрица заполняется нулями. Затем первая строка заполняется битами порождающего полинома, который передаётся в виде строки *gStr*. Этот полином определяет структуру кода и определяет допустимые кодовые слова. Остальные строки матрицы формируются путём циклического сдвига предыдущей строки. Каждый новый ряд создается сдвигом всех битов предыдущей строки вправо, при этом последний бит переносится в начало. Функция продемонстрирована в листинге 2.3.

|  |
| --- |
| const makeGenMatrix = (gStr, k, n) => {  const g = gStr.split('').map(Number);  const matrix = *Array*.from({ length: k }, () => *Array*(n).fill(0));  g.forEach((bit, i) => matrix[0][i] = bit);  for (let i = 1; i < k; i++) {  matrix[i] = [matrix[i - 1][n - 1], ...matrix[i - 1].slice(0, n - 1)];  }  return matrix; }; |

Листинг 2.2 – Функция для построения порождающей матрицы

Далее опишем функцию, которая реализует деление бинарных многочленов по модулю 2. Функция принимает два массива битов: делимое *a* и делитель *b*. Результатом выполнения является частное *q* и остаток *r*. Внутри функции создается копия делимого в виде текущего остатка *r*, а массив *q* используется для хранения битов частного. Пока длина остатка не меньше длины делителя, берётся первый бит текущего остатка. Если он равен единице, выполняется операция XOR между началом *r* и делителем *b*, имитируя побитовую операцию вычитания. После каждой итерации первый бит остатка удаляется, что моделирует сдвиг влево. Программная реализация функции показана в листинге 2.3.

|  |
| --- |
| const polyDiv = (a, b) => {  let r = [...a];  const q = [];  while (r.length >= b.length) {  q.push(r[0]);  if (r[0] === 1) {  for (let i = 0; i < b.length; i++) {  r[i] ^= b[i];  }  }  r.shift();  }  return [q, r]; }; |

Листинг 2.3 – Функция деления многочленов по модулю 2

Следующая функция формирует каноническую порождающую матрицу *G*. Изначально создается единичная матрица *G*, где каждая строка начинается с одного ненулевого бита. Затем для каждой строки матрицы выполняется деление этой строки на полином *g*. Полученный остаток отражает необходимость добавления избыточных битов для корректного кодирования. Остаток записывается в правую часть строки, внося необходимые изменения в матрицу. Программная реализация функции представлена в листинге 2.4.

|  |
| --- |
| const makeCanonicalG = (k, n, gStr) => {  const g = gStr.split('').map(Number);  const G = *Array*.from({ length: k }, (\_, i) =>  *Array*.from({ length: n }, (\_, j) => (i === j ? 1 : 0))  );  for (let i = 0; i < k; i++) {  const [, remainder] = polyDiv(G[i], g);  const remStart = n - remainder.length;  for (let j = 0; j < remainder.length; j++) {  G[i][remStart + j] ^= remainder[j];  }  }  return G; }; |

Листинг 2.4 – Функция для построения канонической порождающей матрицы

Далее опишем функцию, которая формирует проверочную матрицу на основе канонической порождающей матрицы циклического кода. На основе параметров вычисляется количество избыточных битов, которое определяет количество строк в проверочной матрице. Каждая строка проверочной матрицы *H* создаётся следующим образом: сначала берутся соответствующие значения из правой части генераторной матрицы G, то есть элементы, стоящие в столбцах от *k* до *n* - 1. Эти значения представляют собой транспонированную матрицу. Затем к каждому ряду добавляется единичная матрица размером *r* × *r*, где в каждой строке ровно одна единица на диагонали. Таким образом, каждая строка матрицы H объединяет паритетную часть из *G* и единичную матрицу. В результате формируется проверочная матрица *H* размером *r* × *n*, которая используется для контроля и обнаружения ошибок в кодовых словах. Программная реализация функции представлена в листинге 2.5.

|  |
| --- |
| const makeCheckMatrix = (G, k, n) => {  const r = n - k;  return *Array*.from({ length: r }, (\_, i) => [  ...*Array*.from({ length: k }, (\_, j) => G[j][k + i]),  ...*Array*.from({ length: r }, (\_, j) => (i === j ? 1 : 0))  ]); }; |

Листинг 2.5 – Функция для построения проверочной матрицы

Следующая функция реализует процесс кодирования сообщения с использованием циклического кода. В начале работы функция формирует расширенную версию сообщения путём дописывания *n* - *k* нулей в конец исходной строки. Это соответствует умножению *Xk* на *Xr* и подготавливает его к делению на порождающий многочлен. Затем строка *padded* и полином *g* преобразуются в десятичные числа и передаются в функцию, которая выполняет XOR-деление. Возвращается остаток от деления в виде числа *remainder*, который затем преобразуется обратно в двоичную строку, добавляя ведущие нули при необходимости. Эта строка представляет собой избыточные биты, которые позволяют исправлять ошибки при передаче. Наконец, к исходному сообщению добавляются избыточные биты, и возвращается полное кодовое слово, готовое к передаче. Код показан в листинге 2.6.

|  |
| --- |
| const encode = (message, g, k, n) => {  const padded = message + '0'.repeat(n - k);  *console*.log('\nx\_k\*x^r =', padded);   const [, remainder] = xorDivide(parseInt(padded, 2), parseInt(g, 2));  const remBin = remainder.toString(2).padStart(n - k, '0');  *console*.log('x\_k\*x^r XOR g(x) =', remBin);   return message + remBin; }; |

Листинг 2.6 – Функция кодирования сообщения

Далее опишем функцию, которая реализует побитовое деление двух целых чисел по модулю 2. Функция принимает два аргумента – делимое и делитель, оба в виде десятичных чисел. Оба значения сначала преобразуются в бинарное представление, а затем они превращаются в массивы битов. Далее выполняется имитация бинарного деления столбиком. Пока длина текущего массива не меньше длины делителя, проверяется, равен ли первый бит единице. Если да, то происходит побитовая операция XOR, имитируя вычитание по модулю 2. После каждого шага первый бит массива удаляется, что соответствует переходу к следующему разряду в делении. Результатом работы функции является массив из двух элементов: первый элемент равен 0, а второй – остаток от деления, преобразованный обратно в десятичное число. Программная реализация функции представлена в листинге 2.7.

|  |
| --- |
| const xorDivide = (dividend, divisor) => {  let a = dividend.toString(2).split('').map(Number);  let b = divisor.toString(2).split('').map(Number);   while (a.length >= b.length) {  if (a[0] === 1) {  for (let i = 0; i < b.length; i++) {  a[i] ^= b[i];  }  }  a.shift();  }   return [0, parseInt(a.join('') || '0', 2)]; }; |

Листинг 2.7 – Функция деления двух целых чисел

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

В результате получим вычисленные параметры (*n*, *k*)-кода, построенную порождающую и проверочную матрицу канонической формы с вычисленными избыточными битами и сформированным кодовым словом. Результат показан на рисунках 2.1 и 2.2.

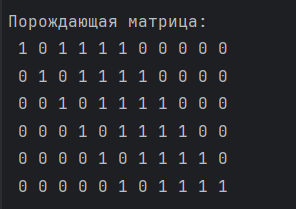
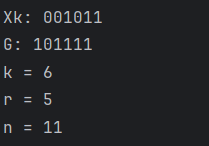
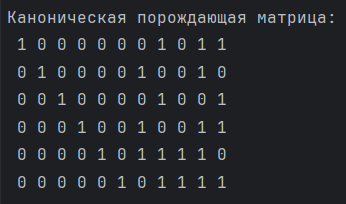
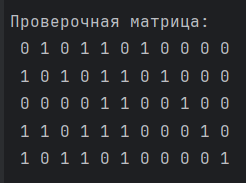


Рисунок 2.1 – Вывод результатов

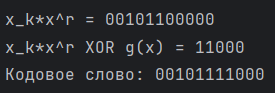


Рисунок 2.2 – Вывод результатов

В данном разделе была рассмотрена реализация циклических кодов, включая построение порождающей и проверочной матриц в канонической форме. Основой для кодирования служит порождающий полином, определяющий допустимые кодовые слова. С его помощью формируется порождающая матрица, каждая строка которой создаётся посредством циклического сдвига предыдущей строки.

Также была реализована функция побитового деления многочленов по модулю 2, что позволило построить каноническую порождающую матрицу *G*. Она учитывает необходимость добавления избыточных битов, определяемых остатком от деления. На её основе строится проверочная матрица *H*, используемая для контроля целостности переданных данных и обнаружения ошибок.

Завершающим этапом стала функция кодирования исходного сообщения, где к информационной части добавляются проверочные биты, сформированные в результате деления с помощью порождающего полинома. Итогом является кодовое слово, готовое к передаче по каналу связи с повышенной надёжностью.

# **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 циклического кода.

Для начала выполняется цикл, моделирующий процесс добавления ошибок в закодированное бинарное сообщение и их последующего обнаружения и исправления с использованием исправляющего кода. На каждой итерации цикла создаётся принятый вариант сообщения с заданным количеством случайных ошибок. Сначала рассматривается случай без ошибок, чтобы убедиться в корректной передаче. Далее для принятого сообщения вычисляется синдром ошибки. Синдром позволяет определить наличие ошибок: если он равен нулю, сообщение не содержит ошибок; если нет – ошибка присутствует.

При наличии ошибок по синдрому определяется вектор ошибок, на основе проверочной матрицы. Затем выполняется исправление принятого сообщения путём XOR между полученным сообщением и вектором ошибки, что позволяет восстановить исходное закодированное сообщение. Реализация цикла представлена в листинге 3.1.

|  |
| --- |
| let received = addRandomErrors(encoded, 0); *console*.log('\nПринятое без ошибок:', received); let syndrome = getSyndrome(received, g); *console*.log('Синдром:', syndrome);  for (let mistakes = 1; mistakes <= 2; mistakes++) {  received = addRandomErrors(encoded, mistakes);  *console*.log(`\nПринятое сообщение (${mistakes} ошибк${mistakes > 1 ? 'и' : 'а'}):`, received);  syndrome = getSyndrome(received, g);  *console*.log('Синдром:', syndrome);  const errVec = findErrorPosition(syndrome, checkMatrix, n);  *console*.log('Вектор ошибки:', errVec);  *console*.log('Исправленное сообщение:', xorStr(received, errVec)); } |

Листинг 3.1 – Цикл для декодирования с различными ошибками

Этот цикл моделирует исправление ошибок в сообщениях и их восстановление с использованием циклического кода.

Также здесь опишем функцию, которая добавляет случайные ошибки в сообщение. Сначала входная бинарная строка преобразуется в массив чисел, где. Далее начинается цикл, количество итераций которого соответствует числу ошибок, указанных в параметре errors (по умолчанию – 1). На каждой итерации случайным образом выбирается позиция в массиве, и бит в этой позиции инвертируется. Это достигается с помощью операции XOR. После добавления всех ошибок массив *bits* преобразуется обратно в строку и возвращается как результат. Функция продемонстрирована в листинге 3.2.

|  |
| --- |
| const addRandomErrors = (str, errors = 1) => {  const bits = str.split('').map(Number);  while (errors--) {  const pos = *Math*.floor(*Math*.random() \* bits.length);  bits[pos] ^= 1;  }  return bits.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 циклический код, проверочные биты, которые были добавлены в процессе кодирования, будут использоваться для проверки корректности данных. При проверке таких данных с помощью алгоритма декодирования циклического кода 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. Синдром позволяет определить, были ли внесены ошибки в переданное сообщение и, в некоторых случаях, локализовать их. Функция принимает два аргумента: *received* – строка, представляющая принятое бинарное сообщение, и *g* – порождающий полином, также заданный в бинарной форме. На первом этапе строковые значения переводятся в числа в двоичной системе. Затем выполняется деление принятого сообщения на порождающий полином. Эта операция возвращает остаток от деления, который и является синдромом. Остаток преобразуется обратно в бинарную строку, и затем дополняется ведущими нулями до длины 5 бит. Это позволяет сохранить корректную длину синдрома для дальнейшей обработки. Реализация функции показана в листинге 3.3.

|  |
| --- |
| const getSyndrome = (received, g) => {  const [, rem] = xorDivide(parseInt(received, 2), parseInt(g, 2));  return rem.toString(2).padStart(5, '0'); }; |

Листинг 3.3 – Функция для нахождения синдрома ошибки

Так как кодовое слово без ошибок, 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 реализуется функция, которая на основе синдрома и проверочной матрицы *H* определяет позицию ошибки и формирует соответствующий вектор ошибок. Функция принимает три параметра: *syndrome* – строка, представляющая собой двоичный синдром ошибки, *H* – проверочная матрица, и *n* – длина переданного бинарного сообщения. В теле функции осуществляется перебор всех столбцов матрицы *H*. Для каждого столбца с индексом *i* формируется строка *col*, содержащая все элементы этого столбца. После этого полученный столбец сравнивается с синдромом. Если они совпадают, значит ошибка произошла в позиции *i*. В этом случае формируется вектор ошибок длиной *n*, в котором все элементы равны 0, кроме одного – на позиции *i* устанавливается 1. Этот вектор указывает на положение бита, который был искажен в принятом сообщении. Если ни один из столбцов не совпал с синдромом, функция возвращает нулевой вектор длиной n, что означает отсутствие исправляемых ошибок. Функция продемонстрирована в листинге 3.4.

|  |
| --- |
| const findErrorPosition = (syndrome, H, n) => {  for (let i = 0; i < n; i++) {  const col = H.map(row => row[i]).join('');  if (col === syndrome) {  return *Array*.from({ length: n }, (\_, j) => (j === i ? 1 : 0)).join('');  }  }  return '0'.repeat(n); }; |

Листинг 3.4 – Функция для вычисления вектора ошибок

Следующим шагом представлена функция, которая выполняет исправление ошибок в бинарном сообщении на основе вектора ошибок. На первом этапе определяется максимальная длина между двумя входными строками, после чего обе строки дополняются ведущими нулями до этой длины. Это необходимо для корректного выполнения побитовой операции XOR. Затем каждая пара соответствующих битов сравнивается и обрабатывается с использованием XOR. Результатом является новая бинарная строка, в которой ошибки, указанные в векторе, устранены – соответствующие биты инвертированы. После обработки всех битов результат собирается в одну строку и возвращается как исправленное сообщение. Реализация функции показана в листинге 3.5.

|  |
| --- |
| const xorStr = (a, b) => {  const maxLen = *Math*.max(a.length, b.length);  const aBits = a.padStart(maxLen, '0');  const bBits = b.padStart(maxLen, '0');  return [...aBits].map((bit, i) => bit ^ bBits[i]).join(''); }; |

Листинг 3.5 – Функция для исправления ошибки

Так как кодовое слово с ошибкой в 3 бите, в результате получим избыточные биты, ненулевой синдром, который соответствует столбцу в проверочной матрице, вектор ошибок, исправленное сообщение в двоичном представлении. Результат показан на рисунке 3.2.

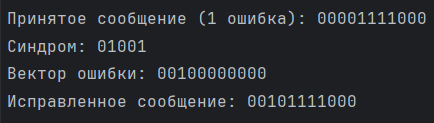
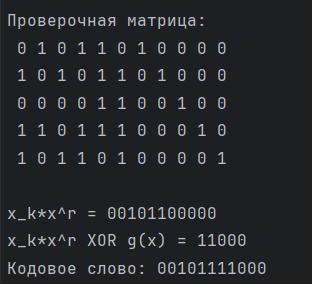


Рисунок 3.2 – Вывод результатов для информационного слова с 1 ошибкой

## **3.3 Двойная ошибка**

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

Так как кодовое слово с ошибками во 3 и 10 битах, в результате получим избыточные биты, ненулевой синдром, который соответствует столбцу в проверочной матрице, вектор ошибок, исправленное сообщение в двоичном представлении. Результат показан на рисунке 3.3.

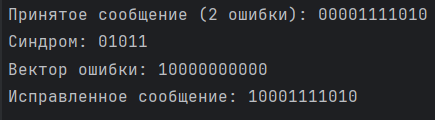


Рисунок 3.3 – Вывод результатов для информационного слова с 2 ошибками

Можно заметить, что синдром равняется сумме по модулю 2 столбцов 3 и 10. 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 были изучены основные принципы построения и применения циклических кодов для обнаружения и исправления ошибок при передаче данных.

Задание выполнялось в соответствии с индивидуальным вариантом, в котором был задан порождающий полином и соответствующее значение *r*. На основе параметров были рассчитаны характеристики (*n*, *k*)-кода, обеспечивающего возможность обнаружения и исправления одиночных ошибок.

В рамках работы была составлена порождающая матрица кода, затем она была приведена к канонической форме, после чего была сформирована проверочная матрица, необходимая для анализа ошибок при приёме сообщений. На следующем этапе было реализовано вычисление избыточных символов, что позволило сформировать полное кодовое слово​, включающее как информационную, так и избыточную часть.

Далее было проведено моделирование передачи кодового слова с внесением 0, 1 и 2 случайных ошибок. Для каждого случая была выполнена проверка на наличие ошибок путём вычисления синдрома. В случае обнаружения одиночной ошибки был построен вектор ошибки, с помощью которого сообщение было успешно исправлено. При наличии двух ошибок возможность их корректного исправления отсутствовала, что соответствует теоретическим ограничениям используемого кода.

Таким образом, в процессе лабораторной работы были реализованы кодирование и декодирование на основе циклического кода, продемонстрирована эффективность данного метода для исправления одиночных ошибок, а также проанализированы его ограничения. При наличии более одной ошибки, система может не справиться с их обнаружением или исправлением.