Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Пермский национальный исследовательский политехнический университет»
Электротехнический факультет
Кафедра «Информационные технологии и автоматизированные системы»

Дисциплина: «Защита информации»

Профиль: «Автоматизированные системы обработки информации и управления»

Семестр 5

ОТЧЕТ

по лабораторной работе №5

Тема: «Шифры обнаружения и коррекции ошибок»

Выполнил: студент группы РИС-19-1б
Миннахметов Э.Ю. _____
Проверил: доцент кафедры ИТАС
Шереметьев В. Г. _____
Дата _____

Пермь, 2021

ЦЕЛЬ РАБОТЫ

Получить практические навыки по применению шифров обнаружения и коррекции ошибок.

ЗАДАНИЕ

Вариант №14. Реализовать коррекцию ошибки в двоичной кодовой последовательности, используя метод циклических кодов с максимальной степенью полинома 3.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Обнаружение ошибок.

Каналы передачи данных ненадежны, да и само оборудование обработки информации работает со сбоями. По этой причине важную роль приобретают механизмы детектирования ошибок. Ведь если ошибка обнаружена, можно осуществить повторную передачу данных и решить проблему. Если исходный код по своей длине равен полученному коду, обнаружить ошибку передачи не предоставляется возможным.

Простейшим способом обнаружения ошибок является контроль по четности. Обычно контролируется передача блока данных (M бит). Этому блоку ставится в соответствие кодовое слово длиной N бит, причем N>M. Избыточность кода характеризуется величиной 1-M/N. Вероятность обнаружения ошибки определяется отношением M/N (чем меньше это отношение, тем выше вероятность обнаружения ошибки, но и выше избыточность).

При передаче информации она кодируется таким образом, чтобы с одной стороны характеризовать ее минимальным числом символов, а с другой – минимизировать вероятность ошибки при декодировании получателем. Для выбора типа кодирования важную роль играет так называемое расстояние Хэмминга.

Пусть А и Б две двоичные кодовые последовательности равной длины. Расстояние Хэмминга между двумя этими кодовыми последовательностями равно числу символов, которыми они отличаются. Например, расстояние Хэмминга между кодами 00111 и 10101 равно 2.

Можно показать, что для детектирования ошибок в n битах, схема кодирования требует применения кодовых слов с расстоянием Хэмминга не менее N+1. Можно также показать, что для исправления ошибок в N битах необходима схема кодирования с расстоянием Хэмминга между кодами не менее 2N+1. Таким образом, конструируя код,

мы пытаемся обеспечить расстояние Хэмминга между возможными кодовыми последовательностями больше, чем оно может возникнуть из-за ошибок.

Широко распространены коды с одиночным битом четности. В этих кодах к каждым M бит добавляется 1 бит, значение которого определяется четностью (или нечетностью) суммы этих M бит. Так, например, для двухбитовых кодов 00, 01, 10, 11 кодами с контролем четности будут 000, 011, 101 и 110. Если в процессе передачи один бит будет передан неверно, четность кода из M+1 бита изменится.

Предположим, что частота ошибок (BER) равна $p=10^{-4}$. В этом случае вероятность передачи 8 бит с ошибкой составит $1-(1-p)^8=7{,}9\times10^{-4}$. Добавление бита четности позволяет детектировать любую ошибку в одном из переданных битах. Здесь вероятность ошибки в одном из 9 бит равна $9p(1-p)^8$. Вероятность же реализации необнаруженной ошибки составит $1-(1-p)^9 - 9p(1-p)^8 = 3{,}6\times10^{-7}$. Таким образом, добавление бита четности уменьшает вероятность необнаруженной ошибки почти в 1000 раз. Использование одного бита четности типично для асинхронного метода передачи. В синхронных каналах чаще используется вычисление и передача битов четности как для строк, так и для столбцов передаваемого массива данных. Такая схема позволяет не только регистрировать но и исправлять ошибки в одном из битов переданного блока.

Контроль по четности достаточно эффективен для выявления одиночных и множественных ошибок в условиях, когда они являются независимыми. При возникновении ошибок в кластерах бит метод контроля четности неэффективен и тогда предпочтительнее метод вычисления циклических сумм (CRC). В этом методе передаваемый кадр делится на специально подобранный образующий полином. Дополнение остатка от деления и является контрольной суммой.

CRC

В Ethernet Вычисление crc производится аппаратно. На рис. 1. показан пример реализации аппаратного расчета CRC для образующего полинома $B(x)= 1 + x^2 + x^3 + x^5 + x^7$. В этой схеме входной код приходит слева.

Эффективность CRC для обнаружения ошибок на многие порядки выше простого контроля четности. В настоящее время стандартизовано несколько типов образующих полиномов. Для оценочных целей можно считать, что вероятность невыявления ошибки в случае использования CRC, если ошибка на самом деле имеет место, равна $(1/2)^r$, где r - степень образующего полинома.

CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$

CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$

Основная идея алгоритма CRC состоит в представлении сообщения виде огромного двоичного числа, делении его на другое фиксированное двоичное число и использовании остатка этого деления в качестве контрольной суммы. Получив сообщение, приемник может выполнить аналогичное действие и сравнить полученный остаток с «контрольной суммой» (переданным остатком). Приведем пример:

Предположим, что сообщение состоит из 2 байт. Их можно рассматривать, как двоичное число 0000 0110 0001 0111. Предположим, что ширина регистра контрольной суммы составляет 1 байт, в качестве делителя используется 1001, тогда сама контрольная сумма будет равна остатку от деления 0000 0110 0001 0111 на 1001.Хотя в данной ситуации деление может быть выполнено с использованием стандартных 32 битных регистров, в общем случае это неверно. Поэтому воспользуемся делением "в столбик" в двоичной системе счисления:

1001=9d делитель

0000011000010111 =0617h =1559d делимое

0000011000010111/1001=2d остаток

Хотя влияние каждого бита исходного сообщения на частное не столь существенно, однако 4 битный остаток во время вычислений может радикально измениться, и чем больше байтов имеется в исходном сообщении (в делимом), тем сильнее меняется каждый раз величина остатка. Вот почему деление оказывается применимым там, где обычное сложение работать отказывается.

В нашем случае передача сообщения вместе с 4 битной контрольной суммой выглядела бы (в шестнадцатеричном виде) следующим образом:06172,где 0617 – это само сообщение, а 2 – контрольная сумма. Приемник, получив сообщение, мог бы выполнить аналогичное деление и проверить, равен ли остаток переданному значению (2).

Полиномиальная арифметика

Все CRC алгоритмы основаны на полиномиальных вычислениях, и для любого алгоритма CRC можно указать, какой полином он использует. Что это значит? Вместо представления делителя, делимого (сообщения), частного и остатка в виде положительных целых чисел, можно представить их в виде полиномов с двоичными коэффициентами или в виде строки бит, каждый из которых является коэффициентом полинома. Например, десятичное число 23 в шестнадцатеричной системе счисления имеет вид 17,в двоичном –

10111, что совпадает с полиномом: $1*x^4 +0*x^3 +1*x^2 +1*x^1 +1*x^0$ или, упрощенно: $x^4 +x^2 +x^1 +x^0$

И сообщение, и делитель могут быть представлены в виде полиномов, с которыми, как и раньше можно выполнять любые арифметические действия; только теперь надо не забывать о «иксах». Предположим, что мы хотим перемножить, например,1101 и 1011.Это можно выполнить, как умножение полиномов:

$(x^3 +x^2 +x^0)(x^3 +x^1 +x^0)$

$=(x^6 +x^4 +x^3+x^5 +x^3 +x^2+x^3 +x^1 +x^0)=$

$=x^6 +x^5 +x^4 +3*x^3 +x^2 +x^1 +x^0$

Теперь для получения правильного ответа нам необходимо указать, что x равен 2,и выполнить перенос бита от члена $3*x^3$.В результате получим: $x^7 +x^3 +x^2 +x^1 +x^0$

А то, что если мы считаем, «X» нам не известным, то мы не можем выполнить перенос. Нам не известно, что $3*x^3$ – это то же самое, что и $x^4 +x^3$, так как мы не знаем, что x =2. В полиномиальной арифметике связи между коэффициентами не установлены и, поэтому, коэффициенты при каждом члене полинома становятся строго типизированными — коэффициент при x^2 имеет иной тип, чем при x^3 . Если коэффициенты каждого члена полинома совершенно изолированы друг от друга, то можно работать с любыми видами полиномиальной арифметики, просто меняя правила, по которым коэффициенты работают. Одна из таких схем для нам чрезвычайно интересна, а именно, когда коэффициенты складываются по модулю 2 без переноса – то есть коэффициенты могут иметь значения лишь 0 или 1,перенос не учитывается. Это называется «полиномиальная арифметика по модулю 2». Возвращаясь к предыдущему примеру, получим результат: $=x^6 +x^5 +x^4 +x^3 +x^2 +x^1 +x^0$.

Двоичная арифметика без учета переносов

Сфокусируем наше внимание на арифметике, применяемой во время вычисления CRC. Фактически как операция сложения, так и операция вычитания в CRC арифметике идентичны операции «Исключающее ИЛИ» (eXclusive OR –XOR),что позволяет заменить 2 операции первого уровня (сложение и вычитание) одним действием, которое, одновременно, оказывается инверсным самому себе. Определив действие сложения, перейдем к умножению и делению. Умножение, как и в обычной арифметике, считается суммой значений первого сомножителя, сдвинутых в соответствии со значением второго сомножителя. При суммировании используется CRC-сложение. Деление несколько сложнее, хотя вполне интуитивно понятно как его выполнять. Действия аналогичны простому делению в столбик с применением CRC-сложения. Если число A получено умножением числа B, то в CRC арифметике это означает, что существует возможность

сконструировать число A из нуля, применяя операцию XOR к число B, сдвинутому на различное количество позиций. Например, если A равно 0111010110,а B – 11,то мы может сконструировать A из B следующим способом:

0111010110

=.....11.

+....11....

+...11.....

+.11.......

Однако, если бы A было бы равно 0111010111,то нам бы не удалось составить его с помощью различных сдвигов числа 11. Поэтому что, как говорят в CRC арифметике, оно не делится на B.

Полностью рабочий пример

Чтобы выполнить вычисление CRC, нам необходимо выбрать делитель. Говоря математическим языком, делитель называется генераторным полиномом(generator polinomial), или просто полиномом, и это ключевое слово любого CRC алгоритма. Степень полинома W (Width – ширина, позиция самого старшего единичного бита) чрезвычайно важна, так как от нее зависят все остальные расчеты. Обычно выбирается степень 16 ил 32,т.к. как это облегчает реализацию алгоритма на современных компьютерах. Степень полинома – это действительная позиция старшего бита, например, степень полинома 10011 равна 4 а, не 5. Выбрав полином приступим к расчетам. Это будет простое деление (в терминах CRC арифметики) сообщения на наш полином.

Единственное, что надо будет сделать до начала работы, так это дополнить сообщение W нулевыми битами. Итак, начнем:

Исходное сообщение: 1101011011

Полином: 10011

Сообщение, дополненное W битами:11010110110000

Теперь просто поделим сообщение на полином, используя правила CRC-арифметики.

11010110110000 / 10011 = 1100001010 (частное, оно никого не интересует) +

+1110 = остаток = контрольная сумма!

Как правило, контрольная сумма добавляется к сообщению и вместе с ним передается по каналам связи. В нашем случае будет передано следующее сообщение:11010110111110. На другом конце канала приемник может сделать одно из равноценных действий:

1)Выделить текст собственно сообщения, вычислить для него контрольную сумму (не

забыв при этом дополнить сообщение W битами), и сравнить ее с переданной.
2)Вычислить контрольную сумму для всего переданного сообщения (без добавления нулей), и посмотреть, получится ли в результате нулевой остаток. Оба эти варианта совершенно равноправны. Однако отныне мы будем работать со вторым вариантом, которое является математически более правильным. Таким образом, при вычислении CRC необходимо выполнить следующие действия:

- Выбрать степень полинома W и полином G (степени W).
- Добавить к сообщению W нулевых битов. Назовем полученную строку M'.
- Поделим M' на G с использованием правил CRC арифметики. Полученный остаток и будет контрольной суммой.

Выбор полинома

Во-первых, надо отметить, что переданное сообщение T является произведением полинома. Чтобы понять это, обратите внимание, что 1)последние W бит сообщения – это остаток от деления дополненного нулями исходного сообщения на выбранный полином, и 2)сложение равносильно вычитанию, поэтому прибавление остатка дополняет значение сообщения до следующего полного произведения. Теперь смотрите, если сообщение при передаче было повреждено, то мы получим сообщение T + E, где E – это вектор ошибки, '+ '– это CRC сложение (или операция XOR). Получив сообщение, приемник делит T + E на G. Так как T mod G =0,(T+E) mod G =E mod G.

Следовательно, качество полинома, который мы выбираем для перехвата некоторых определенных видов ошибок, будет определяться набором произведений G, так как в случае, когда E также является произведением G, такая ошибка выявлена не будет. Следовательно, наша задача состоит в том, чтобы найти такие классы G, произведения которых будут как можно меньше похожи на шумы в канале передачи (которые и вызывают повреждение сообщения).Давайте рассмотрим, какие типы шумов в канале передачи мы можем ожидать. Однобитовые ошибки. Ошибки такого рода означают, что E=1000...0000.Мы можем гарантировать, что ошибки этого класса всегда будет распознаны при условии, что в G по крайней мере 2 бита установлены в "1".Любое произведение G может быть сконструировано операциями сдвига и сложения, и, в тоже время, невозможно получить значение с 1 единичным битом сдвигая и складывая величину, имеющую более 1 единичного бит, так как в результате всегда будет присутствовать по крайней мере 2 бита. Двухбитовые ошибки. Для обнаружения любых ошибок вида 100...000100...000 (то есть когда E содержит по крайней мере 2 единичных бита) необходимо выбрать такое G, которые бы не имело множителей 11,101,1001,10001,и так далее. В качестве примера приведем полином с единичными битами в позициях 15,14

и 1, который не может быть делителем ни одно числа меньше 1...1, где "..."32767 нулей. Ошибки с нечетным количеством бит .Мы может перехватить любые повреждения, когда E имеет нечетное число бит, выбрав полином G таким, чтобы он имел четное количество бит. Чтобы понять это, обратите внимание на то, что

1)CRC умножение является простой операцией XOR постоянного регистрового значения с различными смещениями;

2)XOR – это всего-навсего операция переключения битов; и

3)если Вы применяете в регистре операцию XOR к величине с четным числом битов, четность количеств единичные битов в регистре останется неизменной.

Например, начнем с E=111 и попытаемся сбросить все 3 бита в "0" последовательным выполнением операции XOR с величиной 11 и одним из 2 вариантов сдвигов (то есть, "E=E XOR 011"и "E=E XOR 110").Это аналогично задаче о перевертывании стаканов, когда за одно действие можно перевернуть одновременно любые два стакана. Большинство популярных CRC полиномов содержат четное количество единичных битов. Пакетные ошибки .Пакетная ошибка выглядит как E=000...000111...11110000...00, то есть E состоит из нулей за исключением группы единиц где-то в середине. Эту величину можно преобразовать в E=(00000...00)(1111111...111),где имеется z нулей в левой части и n единиц в правой. Для выявления этих ошибок нам необходимо установить младший бит G в 1.При этом необходимо, чтобы левая часть не была множителем G. При этом всегда, пока G шире правой части, ошибка всегда будет распознана. Несколько популярных полиномов:

16 битные:(16,12,5,0)[стандарт "X25"]

(16,15,2,0)["CRC 16"]

32 битные:(32,26,23,22,16,12,11,10,8,7,5,4,2,1,0) [Ethernet]

Коррекция ошибок.

Исправлять ошибки труднее, чем их детектировать или предотвращать. Процедура коррекции ошибок предполагает два совмещенные процесса: обнаружение ошибки и определение места (идентификация сообщения и позиции в сообщении). После решения этих двух задач, исправление тривиально – надо инвертировать значение ошибочного бита. В наземных каналах связи, где вероятность ошибки невелика, обычно используется метод детектирования ошибок и повторной пересылки фрагмента, содержащего дефект. Для спутниковых каналов с типичными для них большими задержками системы коррекции ошибок становятся привлекательными. Здесь используют коды Хэмминга или коды свертки.

Код Хэмминга представляет собой блочный код, который позволяет выявить и исправить ошибочно переданный бит в пределах переданного блока. Обычно код Хэмминга характеризуется двумя целыми числами, например, (11,7) используемый при передаче 7-битных ASCII-кодов. Такая запись говорит, что при передаче 7-битного кода используется 4 контрольных бита (7+4=11). При этом предполагается, что имела место ошибка в одном бите и что ошибка в двух или более битах существенно менее вероятна. С учетом этого исправление ошибки осуществляется с определенной вероятностью. Например, пусть возможны следующие правильные коды (все они, кроме первого и последнего, отстоят друг от друга на расстояние 4):

00000000

11110000

00001111

11111111

При получении кода 00000111 не трудно предположить, что правильное значение полученного кода равно 00001111. Другие коды отстоят от полученного на большее расстояние Хэмминга.

Рассмотрим пример передачи кода буквы s = 0x073 = 1110011 с использованием кода Хэмминга (11,7).

Позиция бита:	11	10	9	8	7	6	5	4	3	2	1
Значение бита:	1	1	1	*	0	0	1	*	1	*	*

Символами * помечены четыре позиции, где должны размещаться контрольные биты. Эти позиции определяются целой степенью 2 (1, 2, 4, 8 и т.д.). Контрольная сумма формируется путем выполнения операции XOR (исключающее ИЛИ) над кодами позиций ненулевых битов. В данном случае это 11, 10, 9, 5 и 3. Вычислим контрольную сумму:

11 =	1011
10 =	1010
09 =	1001
05 =	0101

03 =	0011
□□□	1110

Таким образом, приемник получит код:

Позиция бита:	11	10	9	8	7	6	5	4	3	2	1
Значение бита:	1	1	1	1	0	0	1	1	1	1	0

Просуммируем снова коды позиций ненулевых битов и получим нуль.

11 =	1011
10 =	1010
09 =	1001
08 =	1000
05 =	0101
04 =	0100
03 =	0011
02 =	0010
□□□	0000

Ну а теперь рассмотрим два случая ошибок в одном из битов посылки, например, в бите 7 (1 вместо 0) и в бите 5 (0 вместо 1). Просуммируем коды позиций ненулевых бит еще раз.

11 =	1011
10 =	1010
09 =	1001
08 =	1000
07 =	0111
05 =	0101
04 =	0100
03 =	0011
02 =	0010
□□□	0111

11 =	1011
10 =	1010
09 =	1001
08 =	1000
04 =	0100
03 =	0011
02 =	0010
□ =	0001

В обоих случаях контрольная сумма равна позиции бита, переданного с ошибкой. Теперь для исправления ошибки достаточно инвертировать бит, номер которого указан в контрольной сумме. Понятно, что если ошибка произойдет при передаче более чем одного бита, код Хэмминга при данной избыточности окажется бесполезен.

В общем случае код имеет N=M+C бит и предполагается, что не более чем один бит в коде может иметь ошибку. Тогда возможно N+1 состояние кода (правильное состояние и n ошибочных). Пусть M=4, а N=7, тогда слово-сообщение будет иметь вид: M4, M3, M2, C3, M1, C2, C1. Теперь попытаемся вычислить значения C1, C2, C3. Для этого используются уравнения, где все операции представляют собой сложение по модулю 2:

C1 = M1 + M2 + M4

C2 = M1 + M3 + M4

C3 = M2 + M3 + M4

Для определения того, доставлено ли сообщение без ошибок, вычисляем следующие выражения (сложение по модулю 2):

C11 = C1 + M4 + M2 + M1

C12 = C2 + M4 + M3 + M1

C13 = C3 + M4 + M3 + M2

Результат вычисления интерпретируется следующим образом.

C11	C12	C13	Значение
1	2	4	Позиция бит
0	0	0	Ошибок нет
0	0	1	Бит C3 не верен
0	1	0	Бит C2 не верен
0	1	1	Бит M3 не верен
1	0	0	Бит C1 не верен
1	0	1	Бит M2 не верен
1	1	0	Бит M1 не верен
1	1	1	Бит M4 не верен

Описанная схема легко переносится на любое число n и M.

Число возможных кодовых комбинаций **M** помехоустойчивого кода делится на **n** классов, где **N** – число разрешенных кодов. Разделение на классы осуществляется так, чтобы в каждый класс вошел один разрешенный код и ближайшие к нему (по расстоянию Хэмминга) запрещенные коды. В процессе приема данных определяется, к какому классу принадлежит пришедший код. Если код принят с ошибкой, он заменяется ближайшим разрешенным кодом. При этом предполагается, что кратность ошибки не более q_m.
Можно доказать, что для исправления ошибок с кратностью не более $q_{m</sub}$ кодовое расстояние должно превышать $2q_m$ (как правило, оно выбирается равным $D = 2q_m + 1$). В теории кодирования существуют следующие оценки максимального числа N n-разрядных кодов с расстоянием D.

d=1	$n=2^n$
d=2	$n=2^{n-1}$
d=3	$n\ 2^n/(1+n)$
d=2q+1	$N \le 2^n (1 + \sum_{i=1}^{d} C_n^i)^{-1}$ (для кода Хэмминга это неравенство превращается в

	равенство)

В случае кода Хэмминга первые k разрядов используются в качестве информационных, причем

$k = n - log(n+1)$,

откуда следует (логарифм по основанию 2), что k может принимать значения 0, 1, 4, 11, 26, 57 и т.д., это и определяет соответствующие коды Хэмминга (3,1); (7,4); (15,11); (31,26); (63,57) и т.д.

Циклические коды

Обобщением кодов Хэмминга являются циклические коды BCH (Bose-Chadhuri-Hocquenghem). Это коды с широким выбором длины и возможностей исправления ошибок. Циклические коды характеризуются полиномом g(x) степени n-k, $g(x) = 1 + g_1x + g_2x^2 + \ldots + x^{n-k}$. g(x) называется порождающим многочленом циклического кода. Если многочлен g(x) n-k и является делителем многочлена $x^n + 1$, то код C(g(x)) является линейным циклическим (n,k)-кодом. Число циклических n-разрядных кодов равно числу делителей многочлена $x^n + 1$.

При кодировании слова все кодовые слова кратны g(x). g(x) определяется на основе сомножителей полинома $x^n + 1$ как:

$x^n + 1 = g(x)h(x)$

Например, если n=7 (x^7+1), его сомножители $(1 + x + x^3)(1 + x + x^2 + x^4)$, а $g(x) = 1+x + x^3$.

Чтобы представить сообщение h(x) в виде циклического кода, в котором можно указать постоянные места проверочных и информационных символов, нужно разделить многочлен $x^{n-k}h(x)$ на g(x) и прибавить остаток от деления к многочлену $x^{n-k}h(x)$. Привлекательность циклических кодов заключается в простоте аппаратной реализации с использованием сдвиговых регистров.

Пусть общее число бит в блоке равно N, из них полезную информацию несут в себе K бит, тогда в случае ошибки, имеется возможность исправить m бит. Таблица 1 содержит зависимость m от N и K для кодов BCH.

Общее число бит N	Число полезных бит M	Число исправляемых бит m
31	26	1
	21	2
	16	3

63	57	1
	51	2
	45	3
127	120	1
	113	2
	106	3

Увеличивая разность N-M, можно не только нарастить число исправляемых бит m, но открыть возможность обнаружить множественные ошибки. В таблице 2 приведен процент обнаруживаемых множественных ошибок в зависимости от M и N-M.

Число полезных бит M	Число избыточных бит (n-m)		
	6	7	8
32	48%	74%	89%
40	36%	68%	84%
48	23%	62%	81%

Другой блочный метод предполагает «продольное и поперечное» контрольное суммирование предаваемого блока. Блок при этом представляется в виде N строк и M столбцов. Вычисляется биты четности для всех строк и всех столбцов, в результате получается два кода, соответственно длиной N и M бит. На принимающей стороне биты четности для строк и столбцов вычисляются повторно и сравниваются с присланными. При выявлении отличия в бите i кода битов четности строк и бите j – кода столбцов, позиция неверного бита оказывается определенной (i,j). Понятно, что если выявится два и более неверных битов в контрольных кодах строк и столбцов, задача коррекции становится неразрешимой. Уязвим этот метод и для двойных ошибок, когда сбой был, а контрольные коды остались корректными.

Применение кодов свертки позволяют уменьшить вероятность ошибок при обмене, даже если число ошибок при передаче блока данных больше 1.

Представление кодовой комбинации в виде многочлена

Описание циклических кодов и их построение удобно проводить с помощью многочленов (или полиномов). Запись комбинации в виде полинома понадобилась для того, чтобы отобразить формализованным способом операцию циклического сдвига

исходной кодовой комбинации. Так, n-элементную кодовую комбинацию можно описать полиномом (n − 1) степени, в виде:

$An-1(x) = a_{n-1}\, x^{n-1} + a_{n-2}\, x^{n-2} + \ldots + a_1\, x + a_0$,

где ai = {0, 1}, причем ai = 0 соответствуют нулевым элементам комбинации, ai = 1 — ненулевым.

Запишем полиномы для конкретных 4-элементных комбинаций:

1101 => $A1(x) = x^3 + x^2 + 1$

1010 => $A2(x) = x^3 + x$

Операции сложения и вычитания выполняются по модулю 2. Они являются эквивалентными и ассоциативными:

G1(x) + G2(x) => G3(x)

G1(x) − G2(x) => G3(x)

G2(x) + G1(x) => G3(x)

Примеры

$G1(x) = x^5 + x^3 + x$

$G2(x) = x^4 + x^3 + 1$

$G3(x) = G1(x) => G2(x) = x^5 + x^4 + x + 1$

Операция деления является обычным делением многочленов, только вместо вычитания используется сложение по модулю 2:

$G1(x) = x^6 + x^4 + x^3$

$G2(x) = x^3 + x^2 + 1$

$x^6 + x^4 + x^3$ | $x^3 + x^2 + 1$

+----------

$x^6 + x^5 + x^3$ | $x^3 + x^2$

$x^5 + x^4$

$x^5 + x^4 + x^2$

---

x^2

Циклический сдвиг кодовой комбинации

Циклический сдвиг кодовой комбинации — перемещение ее элементов справа налево без нарушения порядка их следования, так что крайний левый элемент занимает место крайнего правого.

Основные свойства и само название циклических кодов связаны с тем, что все разрешенные комбинации бит в передаваемом сообщении (кодовые слова) могут быть получены путем операции циклического сдвига некоторого исходного кодового слова.

Допустим, задана исходная кодовая комбинация и соответствующий ей полином:

$a(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_2 x + a_1$

Умножим a(x) на x:

$a(x) \cdot x = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x$

Так как максимальная степень x в кодовой комбинации длиной n не превышает (n − 1), то из правой части полученного выражения для получения исходного полинома необходимо вычесть an(xn − 1). Вычитание an(xn − 1) называется взятием остатка по модулю (xn − 1).

Сдвиг исходной комбинации на i тактов можно представить следующим образом: a(x) · xi − an(xn − 1), то есть умножением a(x) на xi и взятием остатка по модулю (xn − 1). Взятие остатка необходимо при получении многочлена степени, большей или равной n.

Идея построения циклических кодов базируется на использовании неприводимых многочленов. Неприводимым называется многочлен, который не может быть представлен в виде произведения многочленов низших степеней, т. е. такой многочлен делиться только на самого себя или на единицу и не делиться ни на какой другой многочлен. На такой многочлен делиться без остатка двучлен xn + 1. Неприводимые многочлены в теории циклических кодов играют роль образующих полиномов.

Возвращаясь к определению циклического кода и учитывая запись операций циклического сдвига кодовых комбинаций, можно записать порождающую матрицу циклического кода в следующем виде:

$$V = \begin{matrix} p(x) \\ p(x) \cdot x - C2\,(x^n - 1) \\ p(x) \cdot x2 - C3\,(x^n - 1) \\ \ldots \\ p(x) \cdot xm{-}1 - Cm\,(x^n - 1) \end{matrix},$$

где p(x) — исходная кодовая комбинация, на базе которой получены все остальные (m − 1) базовые комбинации, Ci = 0 или Ci = 1 («0», если результирующая степень полинома p(x) · xi не превосходит (n − 1), «1», если превосходит).

Комбинация p(x) называется порождающей (порождающей, генераторной) комбинацией. Для построения циклического кода достаточно верно выбрать p(x). Затем все остальные кодовые комбинации получаются такими же, как и в групповом коде.

Порождающий полином должен удовлетворять следующим требованиям:

p(x) должен быть ненулевым;

вес p(x) не должен быть меньше минимального кодового расстояния: v(p(x)) ≥ dmin;

p(x) должен иметь максимальную степень k (k — число избыточных элементов в коде);

p(x) должен быть делителем полинома (xn − 1).

Выполнение условия 4 приводит к тому, что все рабочие кодовые комбинации циклического кода приобретают свойство делимости на p(x) без остатка. Учитывая это, можно дать другое определение циклического кода. Циклический код — код, все рабочие комбинации которого делятся на порождающий без остатка.

Для определения степени порождающего полинома можно воспользоваться выражением r ≥ log2(n+1), где n — размер передаваемого пакета за раз, т. е. длинна строящегося циклического кода.

Примеры порождающих полиномов, можно найти в таблице:

r, степень полинома	P(x), порождающий полином
2	111
3	1011
4	10011
5	100101, 111101, 110111
6	1000011, 1100111
7	10001001, 10001111, 10011101
8	111100111, 100011101, 101100011

Алгоритм получения разрешенной кодовой комбинации циклического кода из комбинации простого кода

Пусть задан полином $P(x) = a_{r-1}\, x^r + a_{r-2}\, x^{r-1} + \ldots + 1$, определяющий корректирующую способность кода и число проверочных разрядов r, а также исходная комбинация простого k-элементного кода в виде многочлена Ak−1(x).

Требуется определить разрешенную кодовую комбинацию циклического кода (n, k).

Умножаем многочлен исходной кодовой комбинации на xr:

$Ak{-}1(x) \cdot xr$

Определяем проверочные разряды, дополняющие исходную информационную комбинацию до разрешенной, как остаток от деления полученного в предыдущем пункте произведения на порождающий полином:

$Ak{-}1(x) \cdot x^r / Pr(x) =>\ (x)$

Окончательно разрешенная кодовая комбинация циклического кода определится так:

$An{-}1(x) = Ak{-}1(x) \cdot x^r + R(x)$

Для обнаружения ошибок в принятой кодовой комбинации достаточно поделить ее на производящий полином. Если принятая комбинация — разрешенная, то остаток от деления будет нулевым. Ненулевой остаток свидетельствует о том, что принятая комбинация содержит ошибки. По виду остатка (синдрома) можно в некоторых случаях также сделать вывод о характере ошибки, ее местоположении и исправить ошибку.

Пример. Пусть требуется закодировать комбинацию вида 1101, что соответствует $A(x) = x^3 + x^2 + 1$.

Определяем число контрольных символов r = 3. Из таблицы возьмем многочлен $P(x) = x^3 + x + 1$, т. е. 1011.

Умножим A(x) на xr:

$A(x) \cdot x^r = (x^3 + x^2 + 1) \cdot x^3 = x^6 + x^5 + x^3 => 11010000$

Разделим полученное произведение на образующий полином g(x):

$A(x) \cdot x^r / P(x) = (x^6 + x^5 + x^3) / (x^3 + x + 1) = x^3 + x^2 + x + 1 + 1 / (x^3 + x + 1) => 1111 + 001 / 1011$

При делении необходимо учитывать, что вычитание производится по модулю 2. Остаток суммируем с $h(x) \cdot xr$. В результате получим закодированное сообщение:

$F(x) = (x^3 + x^2 + 1) \cdot (x^3 + x + 1) = (x^3 + x^2 + 1) \cdot x^3 + 1 \Rightarrow 1101001$

В полученной кодовой комбинации циклического кода информационные символы A(x) = 1101, а контрольные R(x) = 001. Закодированное сообщение делится на образующий полином без остатка.

Алгоритм определения ошибки

Пусть имеем n-элементные комбинации (n = k + r) тогда:

Получаем остаток от деления E(x) соответствующего ошибке в старшем разряде [1000000000], на порождающий полином Pr(x):

$E1(x) / Pr(x) = R0(x)$

Делим полученный полином H(x) на Pr(x) и получаем текущий остаток R(x).

Сравниваем R0(x) и R(x).

Если они равны, то ошибка произошла в старшем разряде.

Если нет, то увеличиваем степень принятого полинома на x и снова проводим деления:

$H(x) \cdot x / Pr(x) = R(x)$

Опять сравниваем полученный остаток с R0(x).

Если они равны, то ошибки во втором разряде.

Если нет, то умножаем $H(x) \cdot x^2$ и повторяем эти операции до тех пор, пока R(x) не будет равен R0(x).

Ошибка будет в разряде, соответствующем числу, на которое повышена степень H(x), плюс один.

Например: $H(x) \cdot x^3 / Pr(x) = R0(x)$

Линейные блочные коды

Блочный код определяется, как набор возможных кодов, который получается из последовательности бит, составляющих сообщение. Например, если мы имеем K бит, то имеется 2^K возможных сообщений и такое же число кодов, которые могут быть получены из этих сообщений. Набор этих кодов представляет собой блочный код. Линейные коды получаются в результате перемножения сообщения M на порождающую матрицу G[IA]. Каждой порождающей матрице ставится в соответствие матрица проверки четности (n-k)*n. Эта матрица позволяет исправлять ошибки в полученных сообщениях путем вычисления синдрома. Матрица проверки четности находится из матрицы идентичности i и транспонированной матрицы A. G[IA] ==> $H[A^TI]$.

IA_{A^T}</td

Если $\mathbf{G[IA]} = \begin{bmatrix} 100\ 110 \\ 010\ 011 \\ 001\ 101 \end{bmatrix}$, то $H[A^TI] = \begin{bmatrix} 101\ 100 \\ 110\ 010 \\ 011\ 001 \end{bmatrix}$

Синдром полученного сообщения равен

S = [полученное сообщение]· [матрица проверки четности].

Если синдром содержит нули, ошибок нет, в противном случае сообщение доставлено с ошибкой. Если сообщение M соответствует $M=2^k$, а k =3 высота матрицы, то можно записать восемь кодов:

Сообщения	Кодовые	Вычисленные как

	вектора	
M1 = 000	V1 = 000000	M1·G
M2 = 001	V2 = 001101	M2·G
M3 = 010	V3 = 010011	M3· G
M4 = 100	V4 = 100110	M4· G
M5 = 011	V5 = 011110	M5·G
M6 = 101	V6 = 101011	M6 ·G
M7 = 110	V7 = 110101	M7 ·G
M8 = 111	V8 = 111000	M8 ·G

Кодовые векторы для этих сообщений приведены во второй колонке. На основе этой информации генерируется таблица 3, которая называется стандартным массивом. Стандартный массив использует кодовые слова и добавляет к ним биты ошибок, чтобы получить неверные кодовые слова.

Стандартный массив для кодов (6,3) представлен ниже.

000000	001101	010011	100110	011110	101011	110101	111000
000001	001100	010010	100111	011111	101010	110100	111001
000010	001111	010001	100100	011100	101001	110111	111010
000100	001001	010111	100010	011010	101111	110001	111100
001000	000101	011011	101110	010110	100011	111101	110000
010000	011101	000011	110110	001110	111011	100101	101000
100000	101101	110011	000110	111110	001011	010101	011000
001001	000100	011010	101111	010111	100010	111100	011001

Предположим, что верхняя строка таблицы содержит истинные значения переданных кодов. Из таблицы 3 видно, что, если ошибки случаются в позициях,

соответствующих битам кодов из левой колонки, можно определить истинное значение полученного кода. Для этого достаточно полученный код сложить с кодом в левой колонке посредством операции XOR.

Синдром равен произведению левой колонки (CL "coset leader") стандартного массива на транспонированную матрицу контроля четности H^T.

Синдром = $CL \cdot H^T$	Левая колонка стандартного массива
000	000000
001	000001
010	000010
100	000100
110	001000
101	010000
011	100000
111	001001

Чтобы преобразовать полученный код в правильный, нужно умножить полученный код на транспонированную матрицу проверки четности, с тем чтобы получить синдром. Полученное значение левой колонки стандартного массива добавляется (XOR!) к полученному коду, чтобы получить его истинное значение. Например, если мы получили 001100, умножаем этот код на H^T:

$$S = \begin{bmatrix} 110 \\ 011 \\ 101 \\ 100 \\ 010 \\ 001 \end{bmatrix} 0011 = 001$$

этот результат указывает на место ошибки, истинное значение кода получается в результате операции XOR:

$$\begin{array}{r} 001100 \\ 000001 \\ \hline 001101 \end{array}$$

под горизонтальной чертой записано истинное значение кода.

ХОД РАБОТЫ

На рисунке 1 представлена главная форма программы.

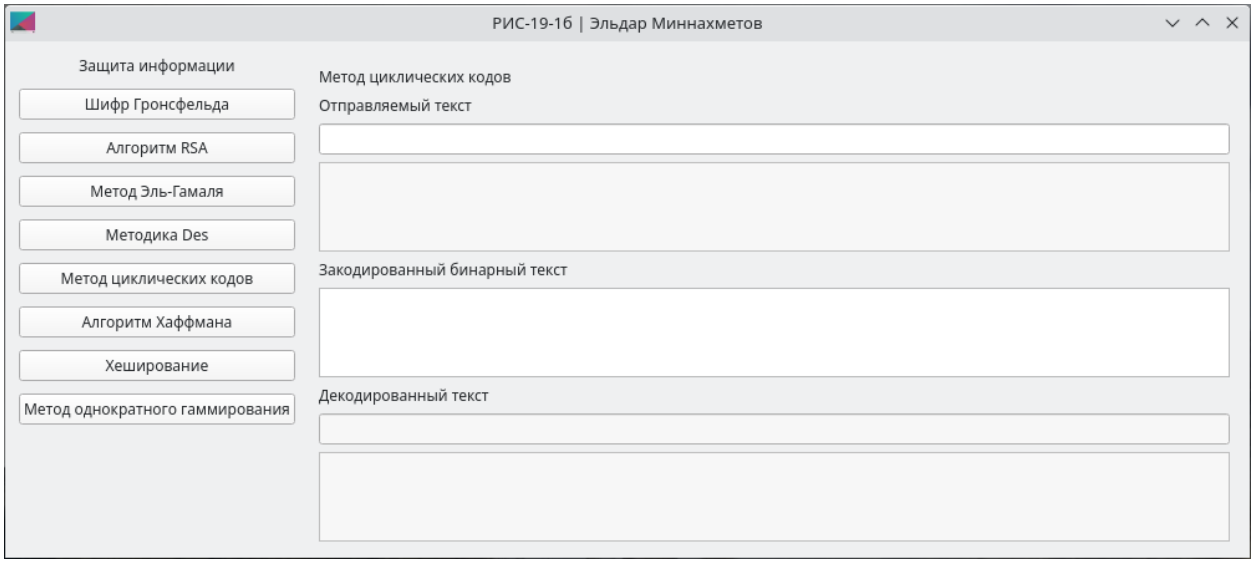

Рисунок 1 – Главная форма программы.

При вводе текста он сразу же преобразуется в двоичную последовательность, а та, в свою очередь, кодируется (сохраняется в соответствующем поле) и отправляется на декодирование. Пример работы программы представлен на рисунке 2.

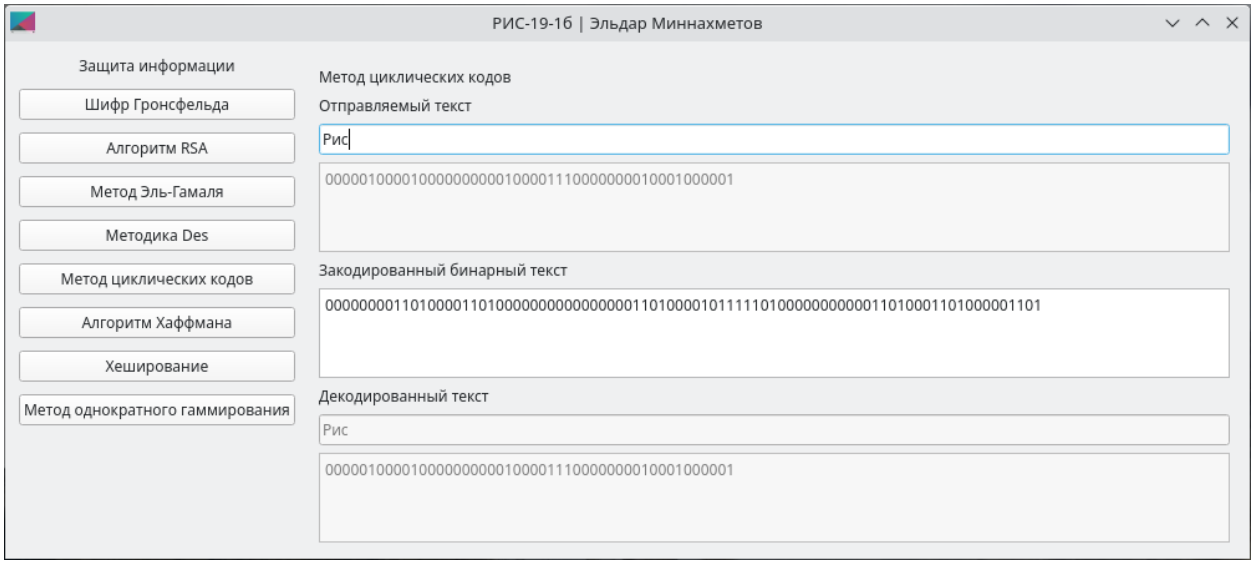

Рисунок 2 – Пример работы программы.

При кодировании битовая последовательность разбивается на блоки по 4 бита, которые можно представить полиномом 3-ей степени, и каждый блок преобразуется в 7-битовый блок. Каждый 7-битовый блок получает защиту от изменения одного из семи битов. Изменим в полученном сообщении произвольное число битов на противоположные

значения, но не более одной замены на 7-битовый блок. Результат замены представлен на рисунке 3.

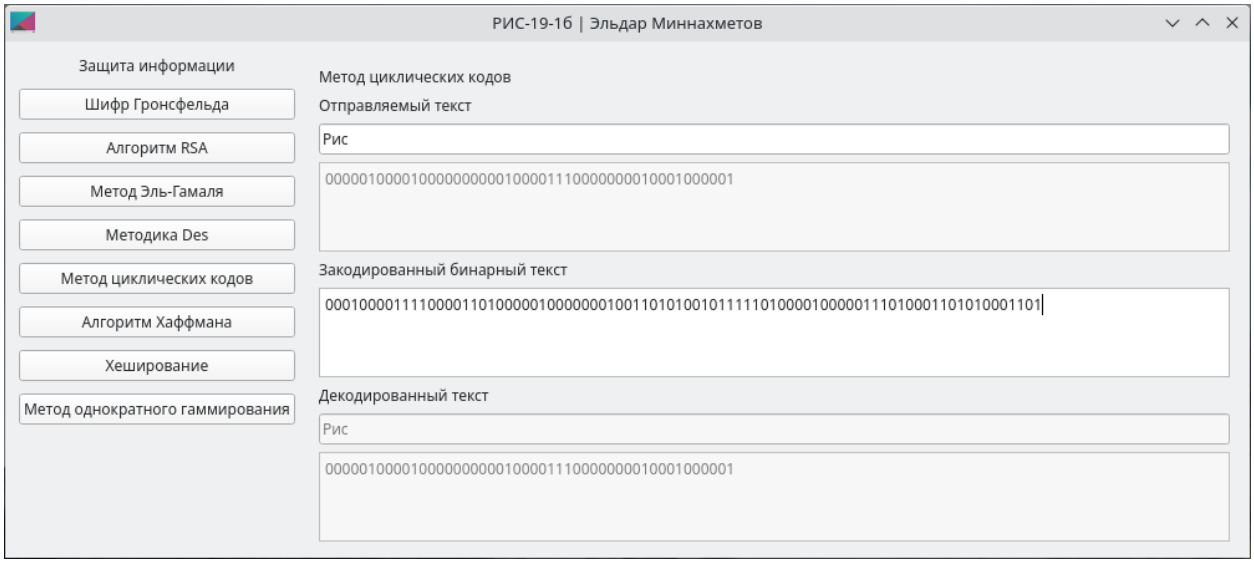

Рисунок 3 – Пример работы программы с ошибкой.

Как видно, декодирование дало тот же результат, что и до замены. Теперь же выполним еще по одной замене в каждый блок – итого по 2 замены во всех 7-битовых блоках. Результат представлен на рисунке 4.

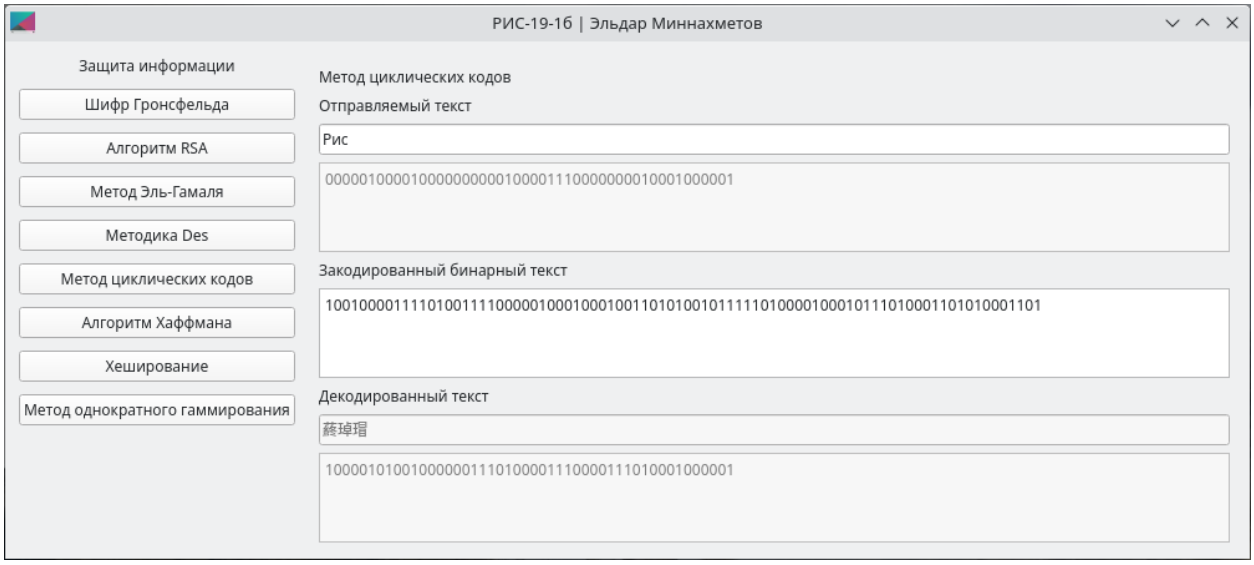

Рисунок 4 – Пример работы программы с ошибкой.

Из результата декодирования можно сделать вывод, что с повреждением 2 и более битов в 7-битовом блоке результат декодирования уже не соответствует отправляемому сообщению, что и не удивительно, ведь циклический код в данном варианте гарантирует защиту от повреждения 1 бита в каждом 7-битовом блоке, с чем он и справляется, что показано на рисунке 3. Вывод: задача выполнена.

ПРИЛОЖЕНИЕ А

Листинг файла CicleTask.h

```
#pragma once

#include <QObject>
#include "Task.h"

class QLabel;
class QLineEdit;
class QTextEdit;
class QVBoxLayout;

class CicleTask: public QObject, public Task {
Q_OBJECT

private:
   QVBoxLayout *lytMain;
   QLabel *lblName;
   QLabel *lblSource;
   QLineEdit *leSource;
   QTextEdit *teSource;
   QLabel *lblEncoded;
   QTextEdit *teEncoded;
   QLabel *lblDecoded;
   QTextEdit *teDecoded;
   QLineEdit *leDecoded;

public:
   CicleTask(): Task("Метод циклических кодов") {}

   void initWidget(QWidget *wgt) override;
   void run() const override { }

private slots:
   void sourceChanged(const QString &text);
   void encodedChanged();

private:
   static QBitArray stringToBits(const QString &text);
   static QString bitsToString(const QBitArray &bits);
   static QString bitsAsString(const QBitArray &bits);
   static QBitArray stringAsBits(const QString &text);

   static QBitArray encode(const QBitArray &bits);
   static QBitArray decode(const QBitArray &bits);

   static QVector<QBitArray> divide(const QBitArray &bits, int denominator);
   static QBitArray sum(const QVector<QBitArray> &bits, int denominator);
   static QBitArray bit4to7(const QBitArray &bits);
   static QBitArray bit7to4(const QBitArray &bits);

   static QBitArray plus(const QBitArray &left, const QBitArray &right);
   static QBitArray code(bool r0, bool r1, bool r2);
   static QBitArray c(std::initializer_list<int> vector);
   static int e(std::initializer_list<int> vector);
};
```

ПРИЛОЖЕНИЕ Б

Листинг файла CicleTask.cpp

```
#include "CicleTask.h"

#include <QLabel>
#include <QBitArray>
#include <QLineEdit>
#include <QTextEdit>
#include <QVBoxLayout>

void CicleTask::initWidget(QWidget *wgt) {
    lytMain = new QVBoxLayout;
    lblName = new QLabel("Метод циклических кодов");
    lblSource = new QLabel("Отправляемый текст");
    leSource = new QLineEdit;
    teSource = new QTextEdit;
    lblEncoded = new QLabel("Закодированный бинарный текст");
    teEncoded = new QTextEdit;
    lblDecoded = new QLabel("Декодированный текст");
    teDecoded = new QTextEdit;
    leDecoded = new QLineEdit;

    teSource->setDisabled(true);
    teDecoded->setDisabled(true);
    leDecoded->setDisabled(true);

    wgt->setLayout(lytMain);
    wgt->setFixedHeight(400);
    lytMain->setAlignment(Qt::AlignTop);
    lytMain->addWidget(lblName);
    lytMain->addWidget(lblSource);
    lytMain->addWidget(leSource);
    lytMain->addWidget(teSource);
    lytMain->addWidget(lblEncoded);
    lytMain->addWidget(teEncoded);
    lytMain->addWidget(lblDecoded);
    lytMain->addWidget(leDecoded);
    lytMain->addWidget(teDecoded);

    connect(leSource, SIGNAL(textChanged(QString)), SLOT(sourceChanged(QString)));
    connect(teEncoded, SIGNAL(textChanged()), SLOT(encodedChanged()));
}

void CicleTask::sourceChanged(const QString &text) {
    QBitArray bits = stringToBits(text);
    QBitArray encoded = encode(bits);
    teSource->setText(bitsAsString(bits));
    teEncoded->setText(bitsAsString(encoded));
}

void CicleTask::encodedChanged() {
    QBitArray bits = stringAsBits(teEncoded->toPlainText());
    QBitArray decoded = decode(bits);
    teDecoded->setText(bitsAsString(decoded));
    leDecoded->setText(bitsToString(decoded));
}

QBitArray CicleTask::stringToBits(const QString &text) {
```

```
    int size = text.size();
    QBitArray result(size * 16);
    for(int i = 0; i < size; ++i) {
        ushort symbol = text[i].unicode();
        for(int j = 0; j < 16; ++j) {
            result[i * 16 + 15 - j] = symbol % 2;
            symbol /= 2;
        }
    }
    return result;
}

QString CicleTask::bitsToString(const QBitArray &bits) {
    QString result;
    int size = bits.size() / 16;
    for(int i = 0; i < size; ++i) {
        ushort symbol = 0;
        for(int j = 0; j < 16; ++j) {
            symbol *= 2;
            symbol += bits[i * 16 + j];
        }
        result += QChar(symbol);
    }
    return result;
}

QString CicleTask::bitsAsString(const QBitArray &bits) {
    QString result;
    int size = bits.size();
    for(int i = 0; i < size; ++i) {
        result += bits[i] ? "1" : "0";
    }
    return result;
}

QBitArray CicleTask::stringAsBits(const QString &text) {
    int size = text.size();
    QBitArray result(size);
    for(int i = 0; i < size; ++i) {
        result[i] = text[i] != '0';
    }
    return result;
}

QBitArray CicleTask::encode(const QBitArray &bits) {
    QVector<QBitArray> devs = divide(bits, 4);
    for(int i = 0, n = devs.size(); i < n; ++i) {
        devs[i] = bit4to7(devs[i]);
    }
    return sum(devs, 7);
}

QBitArray CicleTask::decode(const QBitArray &bits) {
    QVector<QBitArray> devs = divide(bits, 7);
    for(int i = 0, n = devs.size(); i < n; ++i) {
        devs[i] = bit7to4(devs[i]);
    }
    return sum(devs, 4);
}

QVector<QBitArray> CicleTask::divide(const QBitArray &bits, int denominator) {
```

```
    int size = bits.size() / denominator;
    QVector<QBitArray> result(size, QBitArray(denominator));
    for(int i = 0; i < size; ++i) {
        for(int j = 0; j < denominator; ++j) {
            result[i][j] = bits[i * denominator + j];
        }
    }
    return result;
}

QBitArray CicleTask::sum(const QVector<QBitArray> &bits, int denominator) {
    int size = bits.size();
    QBitArray result(size * denominator);
    for(int i = 0; i < size; ++i) {
        for(int j = 0; j < denominator; ++j) {
            result[i * denominator + j] = bits[i][j];
        }
    }
    return result;
}

QBitArray CicleTask::bit4to7(const QBitArray &bits) {
    QBitArray s(7, 0);
    auto sdvig = [](const QBitArray& in) -> QBitArray {
        int size = in.size();
        QBitArray result(size, 0);
        for(int i = 1; i < size; ++i) {
            result[i] = in[i - 1];
        }
        return result;
    };
    auto summod = [](const QBitArray& a, const QBitArray& b, const QBitArray& d, int i) -> int {
        return (a[i] + b[i] + d[i]) % 2;
    };
    auto bitArray = [](const QBitArray& a, int size) -> QBitArray {
        QBitArray result(size, 0);
        for(int i = 0, n = a.size(); i < n; ++i) {
            result[i] = a[i];
        }
        return result;
    };
    auto a = bitArray(bits, 7);
    auto b = sdvig(a);
    auto c = sdvig(b);
    auto d = sdvig(c);
    for(int i = 0; i < 7; ++i) {
        s[i] = summod(a, b, d, i);
    }
    return s;
}

QBitArray CicleTask::bit7to4(const QBitArray &s) {
    QBitArray result(4, 0);
    QBitArray r0(7, 0), r1(7, 0), r2(7, 0), a(7, 0);
    r0[0] = s[6];
    for(int i = 1; i < 7; ++i) {
        r2[i] = r1[i - 1];
        a[i] = r2[i - 1];
        r0[i] = (a[i] + s[6 - i]) % 2;
        r1[i] = (a[i] + r0[i - 1]) % 2;
        if(i >= 3) {
```

```
            result[6 - i] = a[i];
        }
    }
    return plus(result, code(r0[6], r1[6], r2[6]));
}

QBitArray CicleTask::plus(const QBitArray &left, const QBitArray &right) {
    int n = left.size();
    QBitArray result(left);
    for(int i = 0; i < n; ++i) {
        result[i] = (result[i] + right[i]) % 2;
    }
    return result;
}

QBitArray CicleTask::code(bool r0, bool r1, bool r2) {
    int p = e({r0, r1, r2});
    switch(p) {
        case 6: // 4
            return c({1, 0, 0, 0});
        case 3: // 5
            return c({0, 1, 0, 0});
        case 7: // 6
            return c({1, 0, 1, 0});
        case 5: // 7
            return c({1, 1, 0, 1});
        default: // c 1 no 3
            return c({0, 0, 0, 0});
    }
}

QBitArray CicleTask::c(std::initializer_list<int> vector) {
    int size = vector.size();
    QBitArray result(size);
    for(int i = 0; i < size; ++i) {
        result[i] = *(vector.begin() + i);
    }
    return result;
}

int CicleTask::e(std::initializer_list<int> vector) {
    int result = 0;
    for(int i = 0, n = vector.size(); i < n; ++i) {
        result *= 2;
        result += *(vector.begin() + i);
    }
    return result;
}
```