[**Введение** 2](#_Toc200459956)

[**Постановка задачи** 7](#_Toc200459957)

[**Анализ основных блоков алгоритма** 8](#_Toc200459958)

[**S2P** 8](#_Toc200459959)

[**Mapper** 9](#_Toc200459960)

[**IFFT** 13](#_Toc200459961)

[**P2S** 31](#_Toc200459962)

[**Проблема синхронизации** 33](#_Toc200459963)

[**Проверка алгоритма на микроконтроллерах** 38](#_Toc200459964)

[**Заключение** 41](#_Toc200459965)

[**Приложение 1** 42](#_Toc200459966)

[**Принцип работы приемопередатчика на основе OFDM** 42](#_Toc200459967)

[**Существующие реализации** 46](#_Toc200459968)

[**Приложение 2.** 62](#_Toc200459969)

[**Модуляция** 62](#_Toc200459970)

[**Приложение 3.** 67](#_Toc200459971)

[**Алгоритмы БПФ.** 67](#_Toc200459972)

[**Приложение 4.** 75](#_Toc200459973)

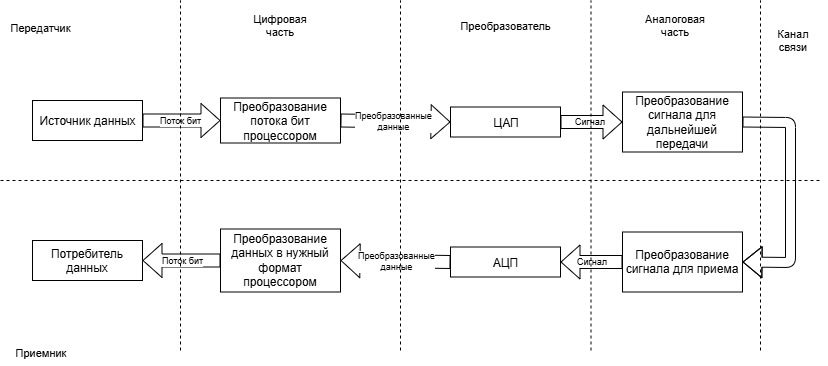
[**Шаблон программы вычисления времени работы и потребляемых ресурсов** 75](#_Toc200459974)

[**Модифицированная основная программа с моделированием шума.** 76](#_Toc200459975)

# **Введение**

Цифровой модуль приемо-передачи на основе OFDM-модуляции.

Современные информационные потоки передаются в виде двоичных данных. Если взять такую систему передачи информации как смартфон, то его можно условно разделить на аналоговую и цифровую части. В данной работе рассматривается только цифровая часть блока приема и передачи информации.

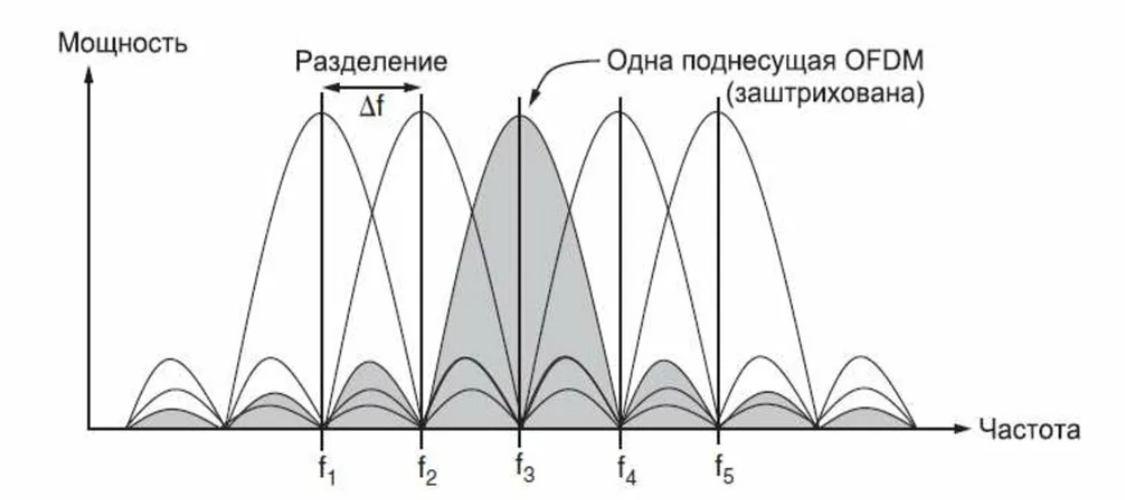


*Рис. 1. Разделение устройства передачи информации на аналоговую и цифровую часть.*

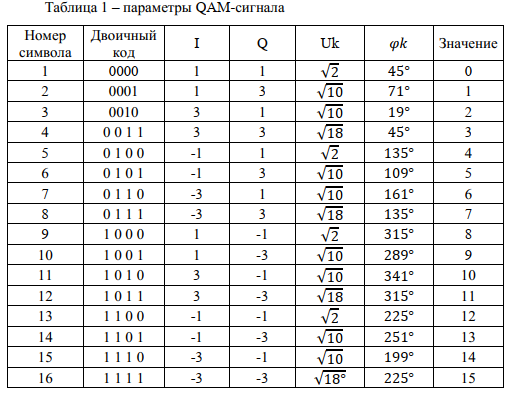
В аналоговой части осуществляется прием сигнала на сверхвысоких частотах (СВЧ, радиоволны в диапазоне частот от 300 МГц до 300 ГГц), преобразование частоты, усиление и фильтрация сигнала. На сигнал действуют шумы, происходит многолучевое распространение в результате переотражения от протяженных предметов – домов, и эффекты смещения частоты – эффект Доплера. В результате частоты, несущие полезную информацию, накладываются друг на друга; происходит потеря информации. Для решения этих проблем используется технология OFDM.

OFDM (мультиплексирование с ортогональным частотным разделением каналов) – технология цифровой связи, в которой сигнал СВЧ образуется из большого количества близко расположенных низкочастотных ортогональных поднесущих – частот, которые занимают ограниченную полосу. Спектры разных поднесущих перекрываются, но не мешают друг другу вследствии правильного выбора временного интервала, на котором все частоты кратны и ортогональны друг другу. OFDM сигнал представлен на рис.2.

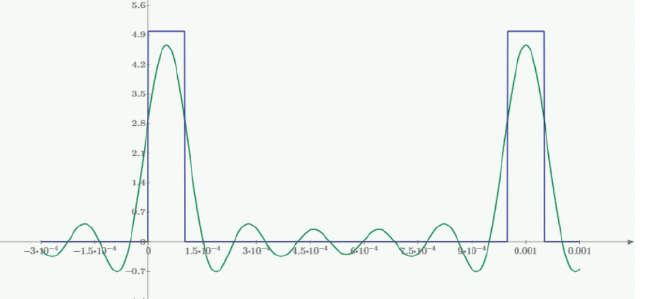
Информация о битах записывается и передается в виде амплитуды и фазы прямоугольных импульсов. Например, при манипуляции 16 QAM все 16 вариантов передачи 4-х бит сведены в таблицу 1.

****

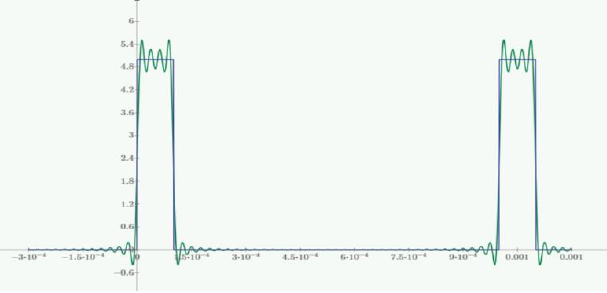
*Рис. 2. Структура OFDM сигнала*

**

Прямоугольные импульсы широкополосны (занимают широкий спектр), поэтому для передачи по узкому каналу их заменяют суммой синусоидальных волн (частот) в полосе текущей поднесущей. Сколько нужно таких волн? Ответить на данный вопрос можно, разложив прямоугольный импульс в ряд Фурье. Используя 8 частот, получится приближение, показанное на рис. 3, оно напоминает начальный сигнал, но не в полной мере его отражает. Хорошее приближение к истинному виду сигнала достигается при использовании 64 частот или более, как представлено на рис. 4.

**

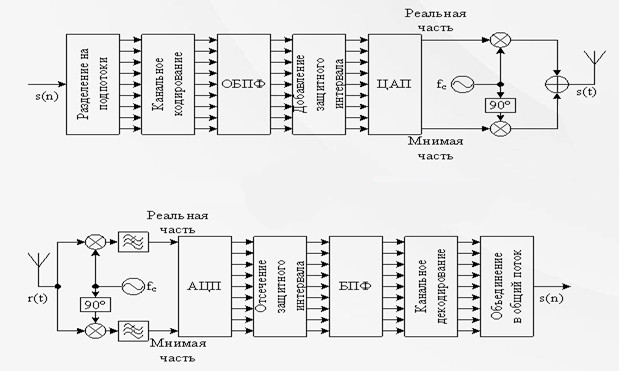
*Рис. 3. Разложение прямоугольного сигнала в ряд Фурье с использованием 8 частот*

**

*Рис. 4. Разложение прямоугольного сигнала в ряд Фурье с использованием 64 частот*

При увеличении числа частот при разложении, итоговый сигнал будет еще точнее повторять исходный, однако при увеличении количества частот увеличивается и время расчета ряда в дальнейших вычислениях. Более подробно эта проблема раскрывается далее в главе «IFFT».

Из основного свойства OFDM следует, что входной поток бит разделяется на тысячи параллельных подканалов, в которых битовые интервалы замедлены в тысячи раз, передаются в виде низкочастотных колебаний и слабо зависят изменения несущего сигнала во времени, по фазе и частоте.. В каждом подканале скорость обработки в N раз меньше, где N – число подканалов. Распараллеливание потока технически выполняет демультиплексер и он описывается в виде алгоритма на языке С++.



*Рис. 5. Схема работы приемника и передатчика по алгоритму OFDM*

В следующем блоке канального кодирования биты группируются в зависимости от отношения сигнал/шум и отображаются точками на комплексной плоскости.



*Рис. 5. Пример, соотношение 4 бит на комплексной плоскости.*

После этого комплексные числа, описывающие амплитуду и фазу каждой точки созвездия с помощью некоторого числа частот в полосе поднесущей при помощи обратного преобразования Фурье (IFFT) преобразуются в амплитудный спектр во временной области, ужимаются (мультиплексируются), подаются на ЦАП и модулируют СВЧ несущую.

В приемнике осуществляются обратные действия с учетом необходимости синхронизации по времени, частоте, амплитуде и фазе. В работе не рассмотрено введение и отсечение защитного интервала.

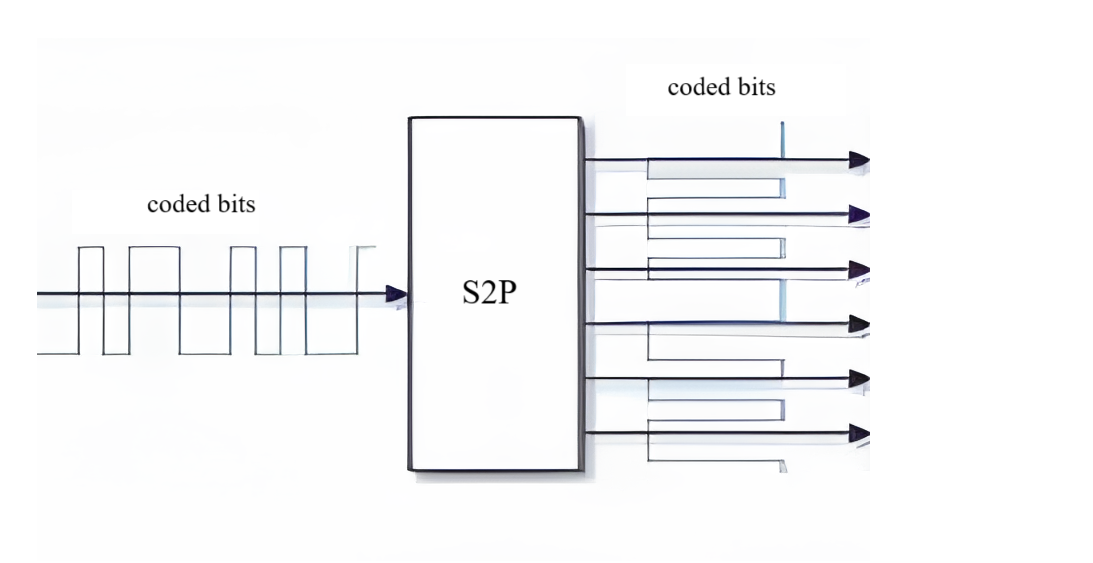
**Задача** ВКРБ заключается в проведении анализа существующих алгоритмов OFDM; реализации алгоритма в одном подканале с использованием процессора STM; реализации алгоритма расчета оптимального соотношения производительности процессоров при организации многопроцессорной версии работы алгоритма. Необходимо заменить аппаратную реализацию программной в части формирования цифрового сигнала передатчика.

# **Постановка задачи**

Проанализировать существующие алгоритмы OFDM как в аппаратной, так и в программной формах; заменить функциональные блоки OFDM-приемопередатчика их программными эквивалентами, заменить аппаратное представление всего приемопередатчика программным в рамках цифровой части обработки. Провести моделирование цифрового программного передатчика с ограниченным количеством параллельных потоков на конкретном микроконтроллере, учитывая требования к производительности, энергопотреблению и занимаемой памяти.

# **Анализ основных блоков алгоритма**

## **S2P**

****

*Рис. 6. Схема блока S2P*

Данный блок разделяет входящий поток бит на группы бит фиксированной длины. Из описания алгоритма OFDM требуется отметить, что длина выходного из блока битового слова зависит от выбранного в блоке Mapper алгоритма. Также, как было ранее описано, важно помнить, что последующая скорость обработки битовых слов будет ниже, чем обработка битового потока, так как иначе блок S2P не будет успевать обрабатывать слова, а последующие блоки будут простаивать, ожидая окончания выполнения работы блока S2P. Это уточнение будет важно при реализации данного алгоритма на микроконтроллере либо при расчете частоты работы функций в имитаторе. Кодовая реализация данного блока имеет следующий вид:

Листинг:

inline void Symbol\_produce(unsigned char\* cadrs, char\* buff)

{

    for (i = 0; i < COUNT\_CADR; ++i, ++iGl)

    {

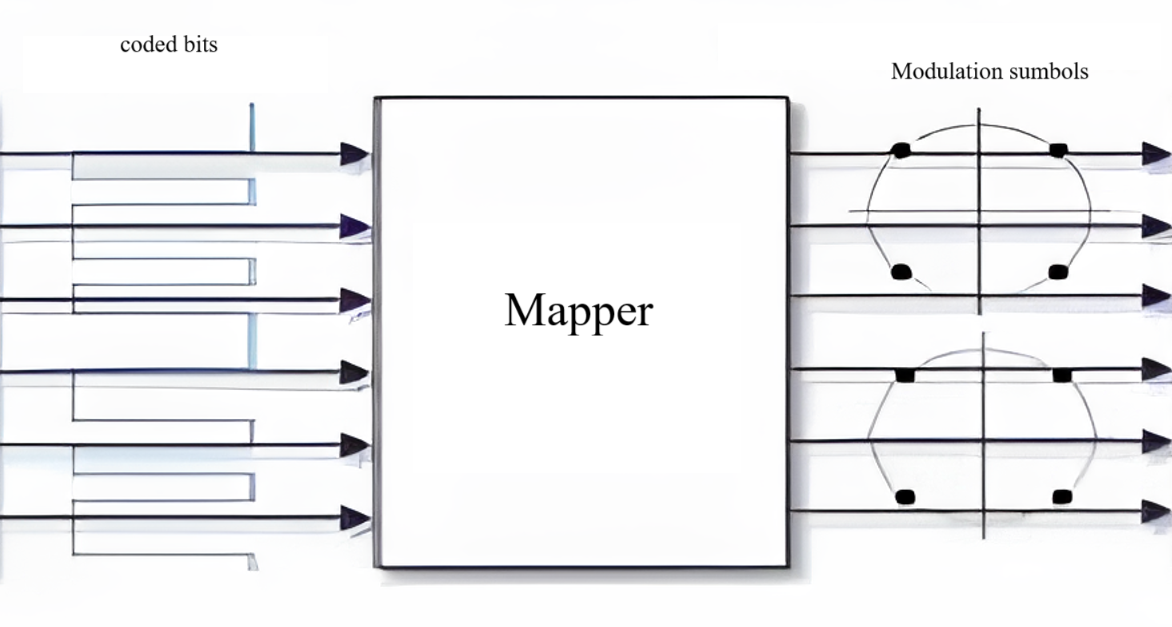
        cadrs[i] = buff[iGl] & 0b1111;

        cadrs[++i] = buff[iGl] >> 4 & 0b1111;

}

}

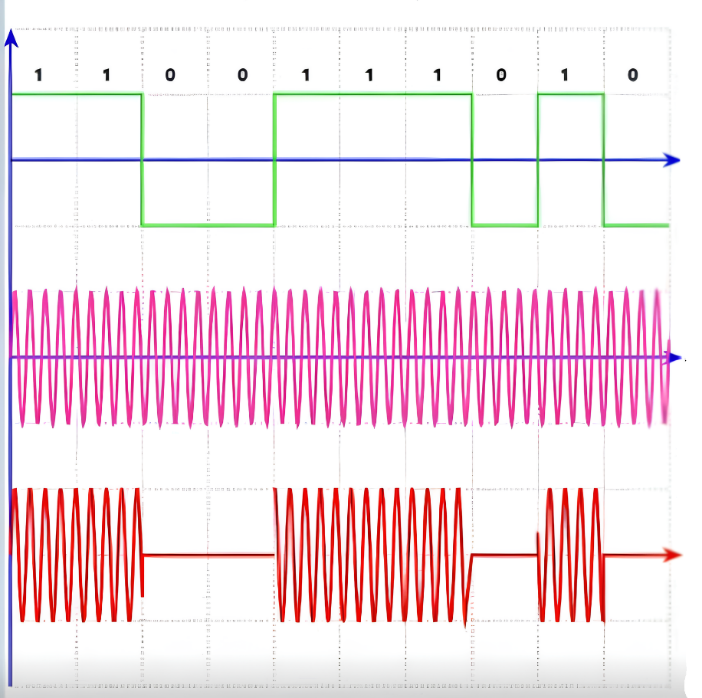
## **Mapper**

****

*Рис. 7. Схема блока Mapper*

Блок Mapper принимает на вход битовые слова и по определенному алгоритму преобразует их в соответствующее комплексное число. Данная операция нужна для реализации передачи слов в виде сигнала.

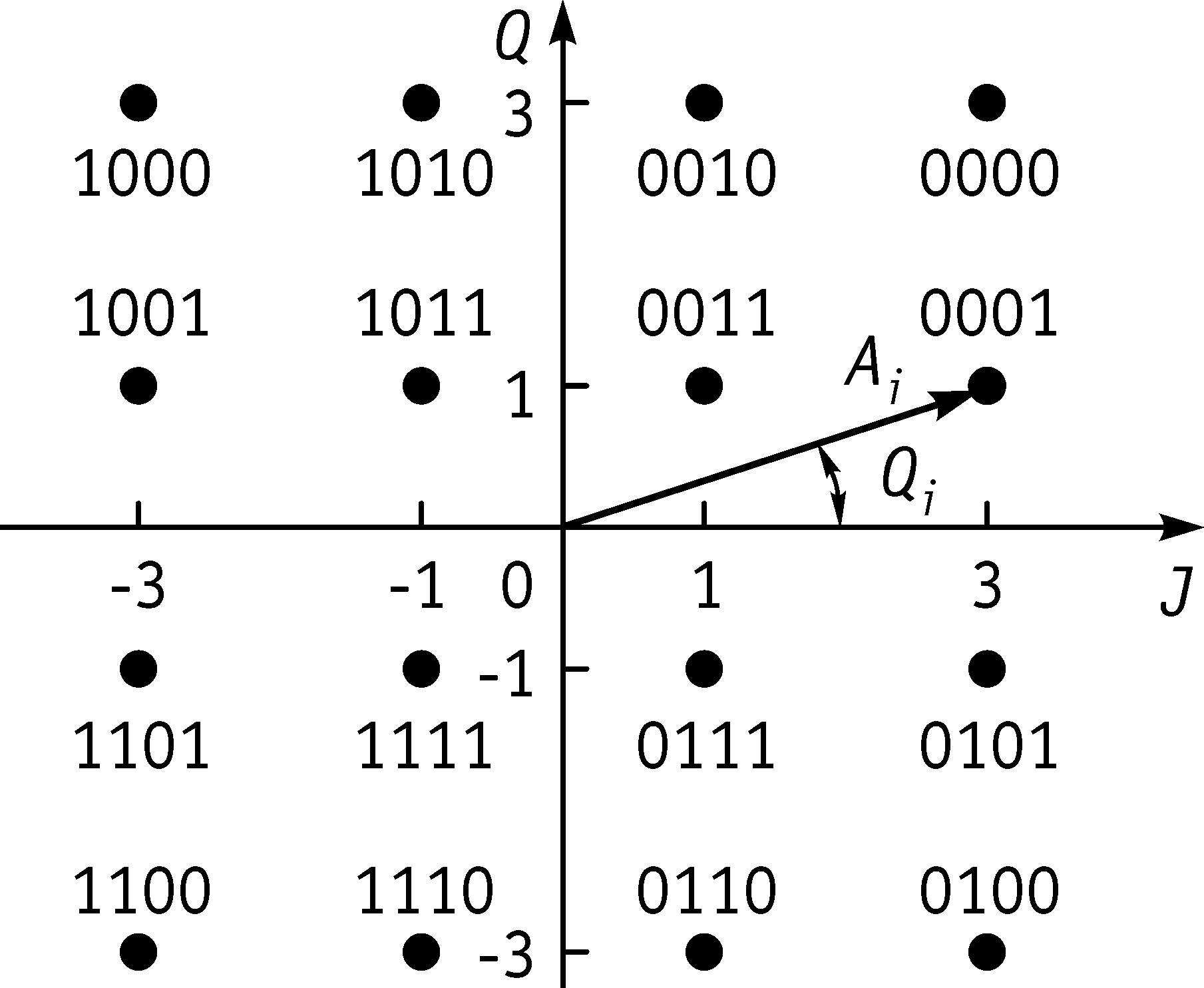
Может возникнуть вопрос, почему бы не передавать сигнал без преобразования сразу в битовом виде, как, например, на рисунке 8.

****

*Рис. 8. Представление потока бит в виде «On-Off»*

Этот способ передачи данных неэффективен, так как если передавать в виде «1 – есть сигнал, 0 – нет сигнала», то такой способ передачи будет в разы проигрывать в скорости передачи информации методам кодирования (модуляции), которые реализуются в данном блоке.

Для рассмотрения способов модуляции, рассмотрим комплексную плоскость. На ней имеется ось реальных значений и ось мнимых значений.



*Рис. 9. Представление алгоритма 16-QAM на комплексной плоскости.*

Ось реальных значений определяет амплитуду волны. Ось мнимых значений - фазу волны. На основе этого имеется 3 вида алгоритмов, в основе которых: изменения фазы при фиксированной амплитуде; изменение амплитуды при фиксированной фазе; изменение фазы и амплитуды. Описание данных видов алгоритмов представлено в Приложении 2.

Заметим, что самое эффективное кодирование является кодирование изменением и амплитуды, и фазы, так как использование обоих параметров позволит кодировать больше данных при меньших изменениях одного из параметров. В данных алгоритмах битовые слова сопоставляются определенным точкам на комплексной плоскости. Изображение сопоставления битовых слов определенным точкам на комплексной плоскости называют картой. Пример карты для алгоритма 16-QAM, который и будет реализован, представлен на рисунке 9.

Стоит отметить, что данная карта составлена так, чтобы соседние точки в ней отличались лишь на 1 бит. Данный способ кодирования позволяет уменьшить вероятность ошибки при передаче и облегчает восстановление данных при возможных ошибках при приеме и дешифровки сигнала.

Программная реализация данного блока:

inline void Modulation(unsigned char \* cadrs, fftw\_complex \*symbolCoding)

{

    // Вариант 1: swithc-case.

    for (i = 0; i < COUNT\_CADR; ++i)

    {

        switch (cadrs[i]) {

            case 0b0000: symbolCoding[i][0] = -3.0; symbolCoding[i][1] =  3.0; break; // 0

            case 0b0001: symbolCoding[i][0] = -1.0; symbolCoding[i][1] =  3.0; break; // 1

            case 0b0010: symbolCoding[i][0] = -3.0; symbolCoding[i][1] =  1.0; break; // 2

            case 0b0011: symbolCoding[i][0] = -1.0; symbolCoding[i][1] =  1.0; break; // 3

            case 0b0100: symbolCoding[i][0] = -3.0; symbolCoding[i][1] = -3.0; break; // 4

            case 0b0101: symbolCoding[i][0] = -1.0; symbolCoding[i][1] = -3.0; break; // 5

            case 0b0110: symbolCoding[i][0] = -3.0; symbolCoding[i][1] = -1.0; break; // 6

            case 0b0111: symbolCoding[i][0] = -1.0; symbolCoding[i][1] = -1.0; break; // 7

            case 0b1000: symbolCoding[i][0] =  3.0; symbolCoding[i][1] =  3.0; break; // 8

            case 0b1001: symbolCoding[i][0] =  1.0; symbolCoding[i][1] =  3.0; break; // 9

            case 0b1010: symbolCoding[i][0] =  3.0; symbolCoding[i][1] =  1.0; break; // 10

            case 0b1011: symbolCoding[i][0] =  1.0; symbolCoding[i][1] =  1.0; break; // 11

            case 0b1100: symbolCoding[i][0] =  3.0; symbolCoding[i][1] = -3.0; break; // 12

            case 0b1101: symbolCoding[i][0] =  1.0; symbolCoding[i][1] = -3.0; break; // 13

            case 0b1110: symbolCoding[i][0] =  3.0; symbolCoding[i][1] = -1.0; break; // 14

            case 0b1111: symbolCoding[i][0] =  1.0; symbolCoding[i][1] = -1.0; break; // 15

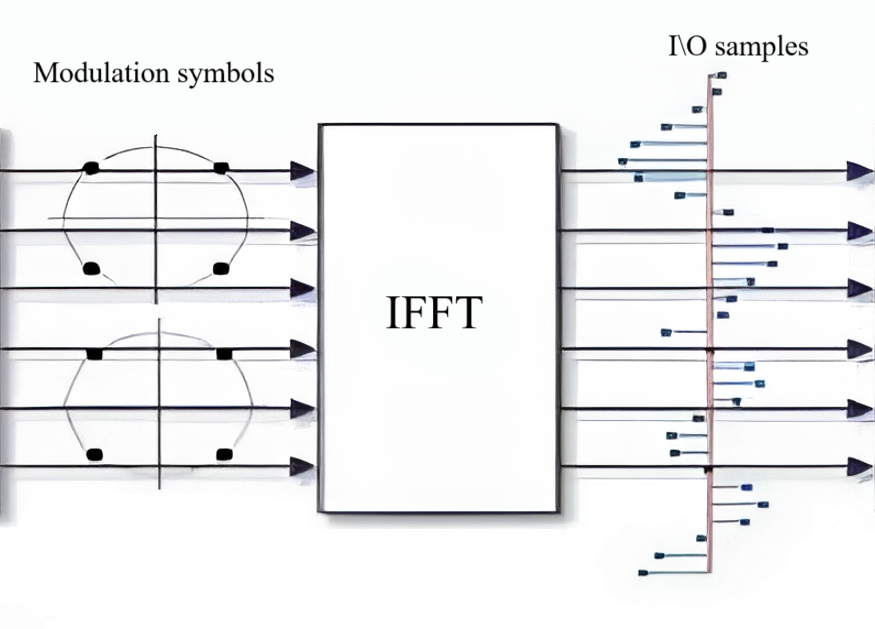
            default:   symbolCoding[i][0] = 0.0; symbolCoding[i][1] = 0.0; // Обработка неожиданной ситуации (опционально)

        }

    }

}

## **IFFT**

****

*Рис. 10. Схема блока IFFT.*

Данный блок преобразовывает комплексное число или комплексные числа, полученные на прошлом этапе, в временной сигнал. Это позволяет сделать обратное преобразование Фурье. Рассмотрим преобразование Фурье. Для дискретного сигнала  из  отсчетов (где ), ДПФ, обозначаемое , определяется как:

для , где:

- n-ый отсчет входной последовательности (сигнала во временной области);

- k-ый отсчет выходной последовательности (спектр сигнала в частотной области);

- длина последовательности;

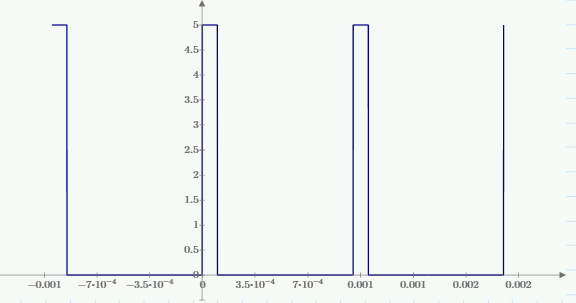
Обратное дискретное преобразование Фурье (ОДПФ) определяется как:

для , где .

Разлагая комплексную экспоненту по формуле Эйлера () получим:

Данную формулу можно разложить на реальную и мнимую составляющие. Они будут равняться соответственно:

Рассмотрим на практическом примере. Имеем следующий сигнал, представленный на рисунке 11, который необходимо передать.



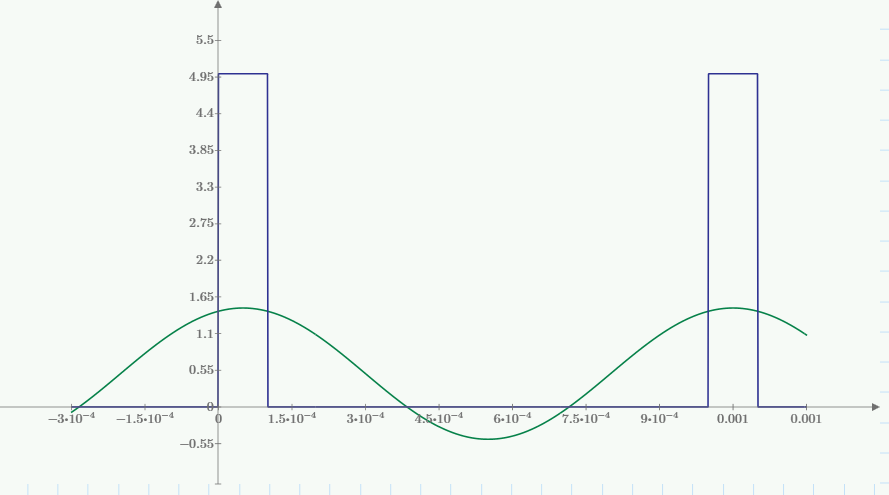
*Рис. 11. Прямоугольный импульс, являющийся входным сигналом.*

При преобразовании данного сигнала с помощью ОПФ мы получим сигнал, представленный на рисунке 12, по форме напоминающий исходный, но который возможно передать в цифро-аналоговый преобразователь для последующей передачи.



*Рис. 12. Сигнал после ОПФ при использовании 40 гармоник.*

Важным параметром в вычислениях ОПФ является число гармоник, на которые будет разложен входной сигнал. Как можно увидеть по рисунку 12, изначально прямоугольный сигнал разложился на сигнал, являющийся суммой синусоид. Эти синусоиды и есть гармоники сигнала. От их количества зависит то, насколько точно выходной сигнал повторит входной. Преобразование на рисунке 12 использует 40 гармоник. В качестве примера, рассмотрим преобразование того же входного сигнала, но используя одну гармонику. Получим следующий результат:



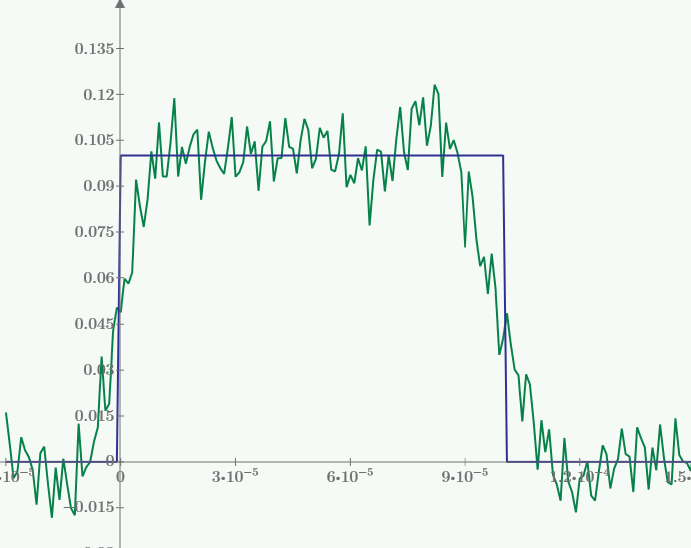
*Рис. 13. Сигнал после ОПФ при использовании 1 гармоники.*

По данному выходному сигналу очень сложно определить изначальный вид. На рисунке 14 представлен результат ОПФ при использовании 4 гармоник.



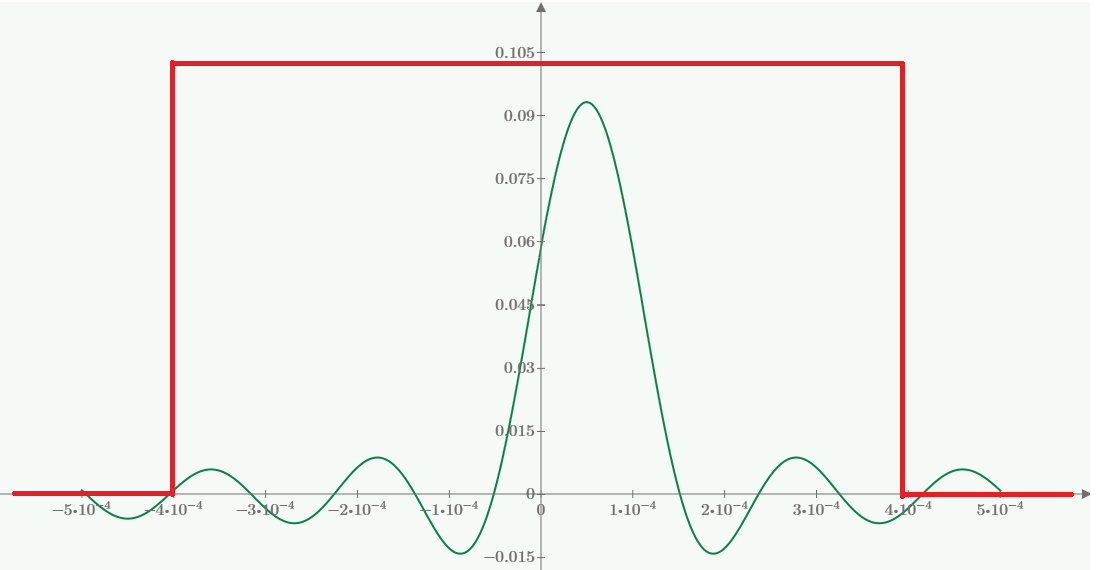
*Рис. 14. Сигнал после ОПФ при использовании 4 гармоник.*

Данный сигнал больше похож на первоначальный, но все еще не так точно его повторяет. Первоначальное разложение на 40 гармоник хорошо раскладывает входной сигнал, однако, при увеличении количества точек вплоть до бесконечности, на которые разбивается сигнал, не удастся идеально восстановить исходных сигнал. График исходного и восстановленного сигнала при разложении на 4000 точек представлен на рисунке 15.



*Рис. 15. Сигнал после ОПФ при использовании 4000 гармоник.*

При дальнейшем увеличении точек будет увеличиваться количество пульсаций, а их амплитуда останется той же. Это связанно с тем, что то, что получается после преобразования, является сверткой частотной характеристики и прямоугольного окна. То есть, когда ограничивается число гармоник, это эквивалентно умножению бесконечной частотной характеристики на прямоугольную функцию, которая равна 1 в пределах выбранного диапазона частот и 0 за его пределами. Это вырезание части спектра и есть применение прямоугольного окна. Наглядный пример на рисунке 16. Также можно использовать другие окна (Ханна, Хэмминга), это поможет побороть пульсации. Свертка – комбинирование двух сигналов.



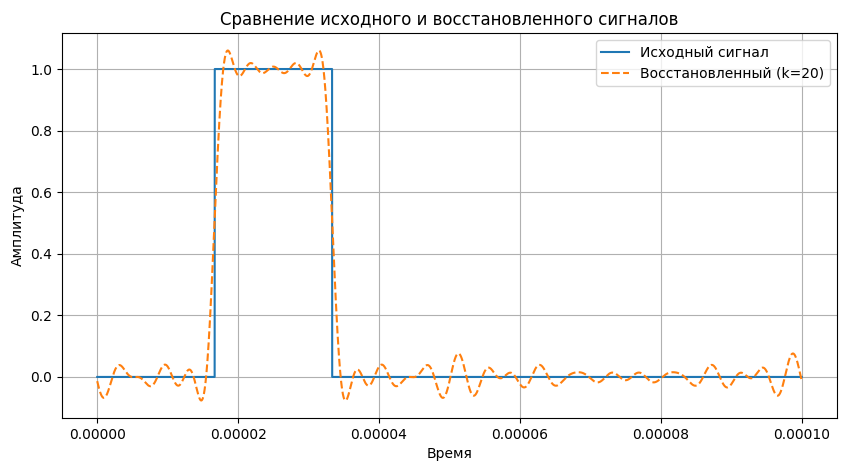
*Рис. 16. Сигнал после ОПФ с использованием прямоугольного окна.*

Учитывая ограниченность времени вычисления, квадратичную (в случае ОДПФ) или логарифмическую (в случае ОБПФ) сложность вычисления и то, что от числа гармоник зависит число вычислений, требуется использовать ограниченное число гармоник, но такое, чтобы сигнал однозначно определялся.

Возьмем такт в 20 мкс. За это время данные должны пройти через все модули и выйти из системы, преобразовавшись в единый временной сигнал. Соответственно выходной сигнал будет передавать данные с периодом в 20 мкс. Окно должно быть в 2 раза больше полезного сигнала, поэтому требуется найти число гармоник для сигнала, длительностью 10 мкс, причем такое, чтобы все 16 сигналов алгоритма 16-QAM были однозначно определимы.

Для дискретных сигналов используется теорема Котельникова: непрерывный сигнал с ограниченным спектром можно точно восстановить по его дискретным отсчётам, если они были взяты с частотой дискретизации, превышающей максимальную частоту сигнала минимум в 2 раза. То есть минимальное число гармоник для шифрования одного сигнала 8.

Попробовав разложить сигнал в ряд Фурье и задав порог дисперсии в 98%, получается, что 20 гармоник при обычном разложении хватает для того, чтобы добиться данной точности. Визуализация представлена ниже.



Из-за конкретизированных преобразований в блоке Mapper на вход данного блока могут поступать 16 конкретных комплексных чисел. Имея фиксированные данные, можно заранее рассчитать каждый коэффициент в вышеописанных формулах, что в разы увеличит скорость вычислений, хоть и потребует хранить большой объем данных.

Заранее определив число гармоник, можно рассчитать, сколько времени уйдет на проход полного цикла работы.

Если использовать стандартные алгоритмы ДПФ, сложность все равно будет квадратичной, но данные вычисления позволяют сократить число операций на каждом пункте вычислений до одного сложения.

Тесты показали, что данный алгоритм быстрее стандартных вычислений в 4-5 раз.

Итог выполнения оптимизированного ОДПФ 10000 раз (Для усреднения данные вычисления проходили 1000 раз):



Итог выполнения стандартного ОДПФ с такими же параметрами:

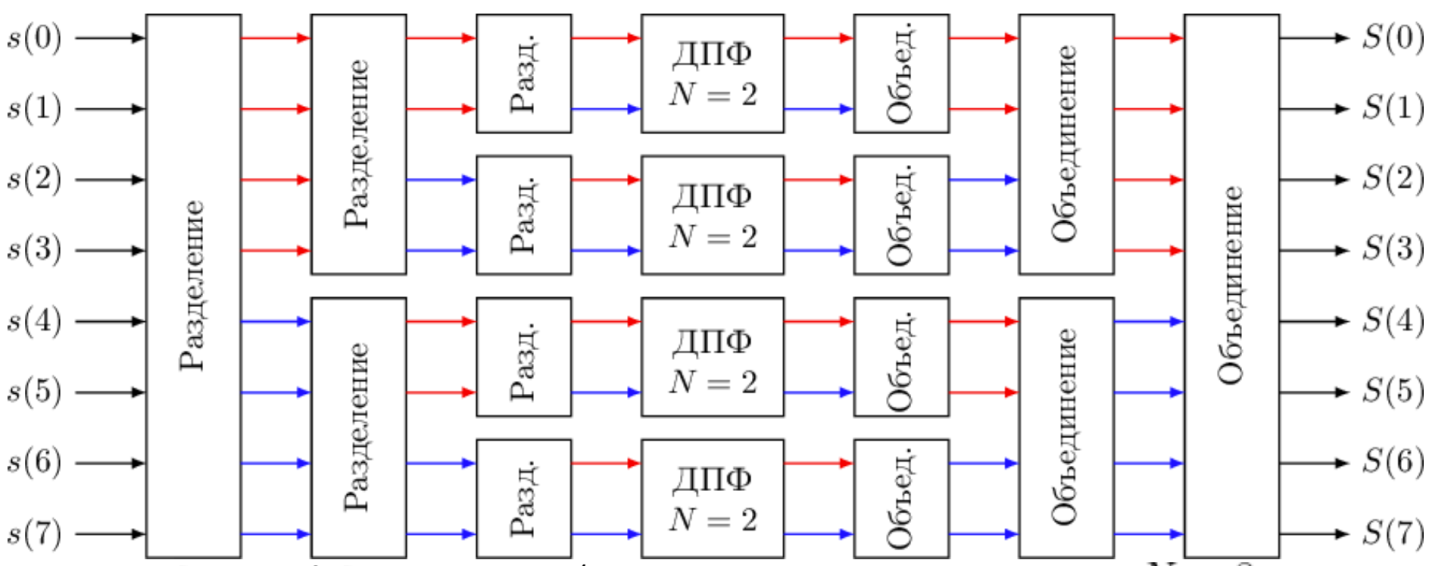


Для последующей оптимизации требуется использования алгоритмов БПФ, которые являются более сложными в реализации. Алгоритмы БПФ имеют логарифмическую сложность, в то время как стандартные ДПФ, даже оптимизированные, имеют квадратичную. Подробное описание алгоритмов БПФ описано в Приложении 3. Ключевым параметром при выборе алгоритма является число гармоник.

Наиболее распространенным алгоритмом является алгоритм Кули-Тьюки. ПФ раскладываются на сумму четных и нечетных составляющих и приводятся к такому виду, что выражения операторов суммы становятся идентичны.

Таким образом число операций уменьшается вдвое. Данное разложение можно продолжать до тех пор, пока либо число операндов в операторе суммы на текущем преобразовании остается четным, либо, в случае показателя степени 2, пока оператор суммы не разложится на четное и нечетное число.

Пример схематичного описания алгоритма:



**\*\*\*Добавить блок-схему алгоритма\*\*\***

Обычно данный алгоритм реализуют рекурсивно. Блок-схема алгоритма представлена на рисунке **\*\*\*N\*\*\***. Даже с учетом дополнительных операций разделения и объединения данный алгоритм при высоком числе гармоник работает быстрее, чем любые варианты ДПФ, так как его сложность будет . Для применения данного алгоритма в обратном преобразовании, потребуется только перед БПФ поменять знак перед мнимой частью спектра, и после БПФ сделать то же самое с результатом и поделить на число гармоник.

Из существующих реализаций алгоритмов БПФ существует библиотека FFTW. Перед вычислением БПФ, требуется определить план БПФ. План в контексте FFTW – структура данных, определяющая алгоритм вычисления преобразования Фурье и особенности в вычислениях (использование большего числа данных или входной переменной в вычислениях, прямое или обратное преобразование). Данная функция принимает на вход число гармоник, которое равно размеру входных и выходных обрабатываемых массивов, ссылки на входной и выходной массив, параметры вычислений в виде флагов.

Так как оптимальное число гармоник уже вычислено, можно практическим путем проверить, с какой скоростью и с каким потреблением ресурсов будет работать каждая вариация алгоритма под конкретные условия данной задачи. Для проверки скорости работы алгоритма реализована шаблонная программа, которая использует выбранное преобразование, и считает скорость работы программы и потребляемые ресурсы. Данная проверка нужна из-за того, что данный алгоритм будет реализован на микроконтроллере с ограниченными ресурсами, поэтому требуется конкретно знать, сколько ресурсов будет потреблять каждая часть алгоритма.

Был реализован шаблон программы на языке C++, который рассчитывает общее время работы программы, время работы самого преобразования, и максимальные потребляемые ресурсы данного процесса. Программа представлена в Приложении 4. В данный шаблон были вставлены реализации оптимальной версии алгоритма ДПФ, стандартного алгоритма БПФ Кули-Тьюки и библиотеки FFTW. Результаты работы данной программы представлены в следующем листинге.

|  |
| --- |
| *Count garmon = 20, Count tests = 100000*  *ZERO RESOURSE:*  *Время работы функции: 1e-07*  *Время работы программы: 0.0022696*  *Время одного преобразования в микросекундах: 1e-06*  *Max RAM: 1916 KiB*  *Optimise DFT...*  *Время работы функции: 0.614788*  *Время работы программы: 0.61575*  *Время одного преобразования в микросекундах: 6.14788*  *Max RAM: 3348 KiB*  *standart FFT...*  *Время работы функции: 0.914408*  *Время работы программы: 0.914514*  *Время одного преобразования в микросекундах: 9.14408*  *Max RAM: 1908 KiB*  *Lib FFT...*  *Время работы функции: 0.0137417*  *Время работы программы: 0.498802*  *Время одного преобразования в микросекундах: 0.137417*  *Max RAM: 4504 KiB* |

Из этого можно увидеть, что для конкретно заданного числа гармоник алгоритм Кули-Тьюки в отличии от оптимального ДПФ справляется на 30% медленнее, но тратит минимальное число ресурсов, однако ни первый, ни второй алгоритмы не укладываются в возможное максимальное время для преобразования. При этом использование библиотеки хоть и тратит больше памяти, чем остальные алгоритмы (без учета контрольной памяти потребляется 2588 KiB), и тратит много времени для составления плана вычислений, но при этом время расчета одного преобразования составляет 10% от минимального времени расчета.

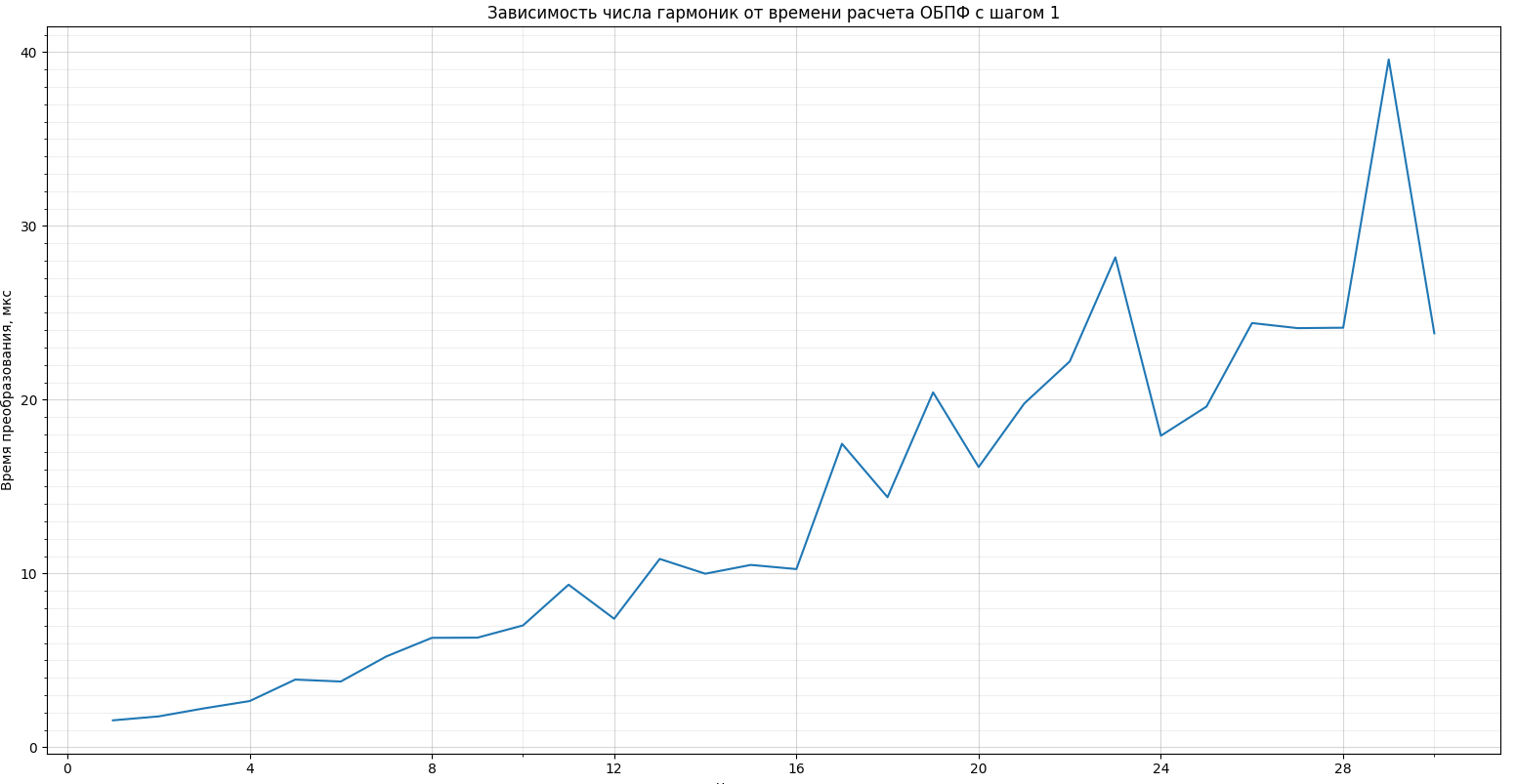
Важно уточнить, что данная программа была протестирована на ПК с быстрым процессором, но работать алгоритм будет на менее производительной STM32. Из результатов работы видно, что тестировать обычные алгоритмы на STM32 нет смысла, так как они намного менее производительные, чем работа библиотеки, либо для последующей оптимизации будет необходимо уменьшать число гармоник.

Также смоделировав Гаусовский шум и используя библиотеку FFTW была написана программа, проверяющая устойчивость преобразования. Программа для данного расчета представлена в Приложении 4.

|  |
| --- |
| *Зашифрованных чисел: 1*  *21 0*  *16 0*  *11 0*  *6 3.31518e-29*  *1 1.40934e-06*  *Зашифрованных чисел: 2*  *21 0*  *16 0*  *11 0*  *6 1.2207e-09*  *1 7.23178e-07*  *Зашифрованных чисел: 3*  *21 0*  *16 0*  *11 0*  *6 1.30725e-08*  *1 3.41995e-06*  *Зашифрованных чисел: 4*  *21 0*  *16 0*  *11 0*  *6 2.44141e-09*  *1 3.64798e-06*  *Зашифрованных чисел: 5*  *21 0*  *16 0*  *11 0*  *6 6.66971e-07*  *1 2.42564e-06*  *Зашифрованных чисел: 6*  *21 0*  *16 0*  *11 0*  *6 3.70371e-07*  *1 1.52283e-06*  *Зашифрованных чисел: 7*  *21 0*  *16 0*  *11 0*  *6 1.89738e-07*  *1 1.29153e-06*  *Зашифрованных чисел: 8*  *21 0*  *16 0*  *11 0*  *6 5.96746e-12*  *1 2.94045e-06* |

Из данного вывода видно, что при повышении числа полезных бит процент ошибки не сильно меняется, из чего можно сделать вывод, что лучшим вариантом для точности и производительности будет использование данной библиотеки, используя 8 гармоник и шифруя 8 чисел одновременно.

Попытки увеличить число гармоник не приведут к увеличению точности, но преобразования будут занимать больше времени, что показано на рис. **\*\*\*N\*\*\*.**



В итоге можно прийти к 8 гармоникам для библиотеки FFTW. Из-за этого время вычисления одного преобразования уменьшится до 0,044833 мкс на ПК.

Для подтверждения корректности работы программы в основную реализацию был добавлен вывод данных на каждом этапе с обратным процессом.

|  |
| --- |
| out data:  0, 1  2, 3  4, 5  6, 7  8, 9  10, 11  12, 13  14, 15  Now data:  1  35  69  103  After S2P:  cadr 0, digit 1  cadr 1, digit 0  cadr 2, digit 3  cadr 3, digit 2  cadr 4, digit 5  cadr 5, digit 4  cadr 6, digit 7  cadr 7, digit 6  After modulation:  cadr 0, Re: -1.000000, Im: 3.000000  cadr 1, Re: -3.000000, Im: 3.000000  cadr 2, Re: -1.000000, Im: 1.000000  cadr 3, Re: -3.000000, Im: 1.000000  cadr 4, Re: -1.000000, Im: -3.000000  cadr 5, Re: -3.000000, Im: -3.000000  cadr 6, Re: -1.000000, Im: -1.000000  cadr 7, Re: -3.000000, Im: -1.000000  After OFFT:  cadr 0, Re: -5.656854, Im: 0.000000  cadr 1, Re: -2.707107, Im: 3.121320  cadr 2, Re: 0.000000, Im: 0.000000  cadr 3, Re: -1.292893, Im: 1.121320  cadr 4, Re: 2.828427, Im: 0.000000  cadr 5, Re: 1.292893, Im: 1.121320  cadr 6, Re: 0.000000, Im: 0.000000  cadr 7, Re: 2.707107, Im: 3.121320  RECOWER...  After FFT:  cadr 0, Re: -1.000000, Im: 3.000000  cadr 1, Re: -3.000000, Im: 3.000000  cadr 2, Re: -1.000000, Im: 1.000000  cadr 3, Re: -3.000000, Im: 1.000000  cadr 4, Re: -1.000000, Im: -3.000000  cadr 5, Re: -3.000000, Im: -3.000000  cadr 6, Re: -1.000000, Im: -1.000000  cadr 7, Re: -3.000000, Im: -1.000000  After demodulation:  cadr 0, digit 1  cadr 1, digit 0  cadr 2, digit 3  cadr 3, digit 2  cadr 4, digit 5  cadr 5, digit 4  cadr 6, digit 7  cadr 7, digit 6  After P2S: #Eg  Now data:  137  171  205  239  After S2P:  cadr 0, digit 9  cadr 1, digit 8  cadr 2, digit 11  cadr 3, digit 10  cadr 4, digit 13  cadr 5, digit 12  cadr 6, digit 15  cadr 7, digit 14  After modulation:  cadr 0, Re: 1.000000, Im: 3.000000  cadr 1, Re: 3.000000, Im: 3.000000  cadr 2, Re: 1.000000, Im: 1.000000  cadr 3, Re: 3.000000, Im: 1.000000  cadr 4, Re: 1.000000, Im: -3.000000  cadr 5, Re: 3.000000, Im: -3.000000  cadr 6, Re: 1.000000, Im: -1.000000  cadr 7, Re: 3.000000, Im: -1.000000  After OFFT:  cadr 0, Re: 5.656854, Im: 0.000000  cadr 1, Re: -2.707107, Im: 3.121320  cadr 2, Re: 0.000000, Im: 0.000000  cadr 3, Re: -1.292893, Im: 1.121320  cadr 4, Re: -2.828427, Im: 0.000000  cadr 5, Re: 1.292893, Im: 1.121320  cadr 6, Re: 0.000000, Im: 0.000000  cadr 7, Re: 2.707107, Im: 3.121320  RECOWER...  After FFT:  cadr 0, Re: 1.000000, Im: 3.000000  cadr 1, Re: 3.000000, Im: 3.000000  cadr 2, Re: 1.000000, Im: 1.000000  cadr 3, Re: 3.000000, Im: 1.000000  cadr 4, Re: 1.000000, Im: -3.000000  cadr 5, Re: 3.000000, Im: -3.000000  cadr 6, Re: 1.000000, Im: -1.000000  cadr 7, Re: 3.000000, Im: -1.000000  After demodulation:  cadr 0, digit 9  cadr 1, digit 8  cadr 2, digit 11  cadr 3, digit 10  cadr 4, digit 13  cadr 5, digit 12  cadr 6, digit 15  cadr 7, digit 14 |

## **P2S**

****

*Рис. 17. Схема блока P2S.*

Данный блок получает на вход сигналы, вычисленные в предыдущем блоке, и объединяет их для последующей передачи. По основному свойству OFDM, сигналы располагаются довольно плотно, но при этом так, чтобы основные части сигнала не перекрывали друг друга. На выходе возможны 16 сигналов различной фазы и амплитуды. Взглянув на Таблицу **\*\*\*N\*\*\***, можно заметить, что минимально возможное число точек на временной шкале для однозначного описания сигнала – 12, так как в общем имеется 12 различных фаз, а амплитуда не влияет на расположение точки на временной шкале. Имеем, что требуется ужать 20 точек до 12.

Путем тестирования были определены 12 точек, которые несут полезную информацию. Блок P2S по конкретным индексам переводит входной массив из 20 комплексных чисел в массив из 12 чисел. Потребуется итерироваться по точкам 1, 3, 5, 6, 7, 9, 11, 13, 14, 15, 17, 19. Так как между индексами нет зависимости, самым оптимальным вариантом с точки зрения скорости исполнения будет прямое присваивание. Существует вариант итерирования по циклу for, но он потребует создание нового массива с индексами и дополнительные затраты на сравнение внутри for, поэтому был выбран иной вариант.

Наилучшим вариантом является функция начальной инициализации данного блока. То есть перед началом работы цикла программа перебирает все 16 возможных вариантов сигнала, преобразует их с помощью ОБПФ, анализирует сигнал, находя точки всплеска, запоминает их временной индекс, на основе этого составляет массив, в котором содержатся точки временной шкалы, которые соответствуют полезной информации для каждой точки на созвездии 16-QAM.

Программная реализация данного блока:

void p2s(fftw\_complex \*\*inBlock, comp\_t\* outBlock)

{

outBlock[0] = inBlock[1];

outBlock[1] = inBlock[3];

outBlock[2] = inBlock[5];

outBlock[3] = inBlock[6];

outBlock[4] = inBlock[7];

outBlock[5] = inBlock[9];

outBlock[6] = inBlock[11];

outBlock[7] = inBlock[13];

outBlock[8] = inBlock[14];

outBlock[9] = inBlock[15];

outBlock[10] = inBlock[17];

outBlock[11] = inBlock[19];

}

# **Проблема синхронизации**

Сложность реализации данного алгоритма на STM32 заключается в проблеме синхронизации. Необходимо обеспечить одновременное выполнение нескольких задач: прием данных из входного битового потока с заданной скоростью передачи данных, передачу обработанных данных и выполнение основной вычислительной функции с определенной частотой. Основная трудность заключается в имитации многопоточности на одноядерном микроконтроллере, где отсутствует встроенный планировщик задач, характерный для многопроцессорных систем. Хотя эмуляция многопоточности на компьютере с мощным процессором и развитыми технологиями распараллеливания процессов не представляет особых проблем, для STM32 требуются альтернативные подходы.

Существует несколько решений для организации параллельной работы на STM32.

RTOS (Операционная система реального времени): RTOS представляет собой специализированное программное обеспечение, предназначенное для управления ресурсами вычислительной системы в условиях жестких временных ограничений. Она обеспечивает детальное управление ресурсами процессора, предоставляет абстракции для упрощения реализации многопоточности, включая сложный, но эффективный встроенный планировщик задач, и обеспечивает развитые средства синхронизации потоков (мьютексы, семафоры, очереди сообщений). В основе RTOS лежит компактное ядро, которое отвечает за выполнение функций планирования (выбор задачи для выполнения в данный момент времени), управления памятью (выделение и освобождение памяти для задач) и обработки прерываний. Архитектура RTOS позволяет разработчикам создавать сложные системы, состоящие из множества параллельно выполняющихся задач, каждая из которых имеет свой приоритет и свое время, к которому задача должна быть выполнена. При выборе RTOS следует тщательно учитывать факторы, такие как размер ядра (влияет на потребление памяти), потребление ресурсов (влияет на производительность системы), доступность библиотек (облегчает разработку) и качество инструментальных средств разработки (упрощает отладку и тестирование). Однако, использование RTOS приводит к дополнительному потреблению ресурсов на переключение между задачами, которое, хоть и оптимизировано с использованием аппаратных возможностей, все равно требует процессорного времени. Примерами популярных RTOS для STM32 являются FreeRTOS (бесплатная и широко используемая), Micrium µC/OS (коммерческая, с расширенной функциональностью) и Zephyr (с открытым исходным кодом, ориентирована на безопасность и надежность). RTOS представляет собой удобный инструмент для разработки сложных многопоточных приложений. Но в данной задаче, где требуется максимальная производительность, а логика программы предсказуема и не предполагает значительных изменений, использование RTOS может оказаться избыточным, так как это может значительно снизить производительность системы, что критично для данной задачи.

В контексте данной задачи предлагается сделать собственную эмуляция многопоточности на основе системы прерываний. В рамках данного подхода используются прерывания для контроля. Прерывание – это аппаратный механизм, позволяющий микроконтроллеру временно приостановить выполнение текущей основной программы и переключиться на выполнение специальной подпрограммы, называемой обработчиком прерывания. Этот механизм позволяет микроконтроллеру реагировать на внешние события, например, поступление данных по последовательному порту, нажатие кнопки, или внутренние события, например, срабатывание таймера, завершение операции ввода-вывода, без необходимости постоянно опрашивать соответствующие флаги или регистры. Когда происходит прерывание, микроконтроллер сохраняет текущее состояние программы (значение регистров, адрес возврата) в стеке, а затем переходит к выполнению обработчика прерывания. После завершения работы обработчика прерывания микроконтроллер восстанавливает сохраненное состояние и продолжает выполнение основной программы с того места, где она была приостановлена. Важно отметить, что обработчики прерываний должны быть максимально короткими и эффективными, чтобы не задерживать выполнение основной программы. Существует несколько типов прерываний, различающихся по источнику возникновения и приоритету: аппаратные (вызываемые внешними устройствами, например, датчиками или периферийными устройствами), программные (вызываемые командами в программе, используются для переключения между задачами или вызова системных функций) и системные (вызываемые ядром микроконтроллера, используются для обработки ошибок или исключительных ситуаций). Каждый тип прерывания имеет свой уникальный вектор прерывания, который указывает на адрес соответствующего обработчика прерывания. В данном проекте используются следующие типы прерываний: Всего в системе используется четыре прерывания. Внешнее прерывание активирует цикл работы программы. Другое прерывание отвечает за отправку данных, полученных на предыдущем цикле, и его частота определяется необходимой скоростью передачи данных. Следующее прерывание отвечает за прием входящего бита и его запись в память для последующей обработки. Частота этого прерывания зависит от скорости входящего битового потока. Последнее прерывание использует системный таймер, при срабатывании которого происходит передача данных между функциями (от функции приема данных к вычислительной и от вычислительной к передающей) и устанавливается флаг, сигнализирующий о готовности к выполнению вычислительной функции. Этот флаг используется для регулирования частоты выполнения вычислительной функции. Перед выполнением вычислительной функции производится проверка флага. Если флаг установлен, функция обнуляет его, выполняет необходимые вычисления и ожидает изменения состояния флага. Такой подход позволяет добиться псевдо-параллельности, минимизируя накладные расходы, связанные с использованием полноценной RTOS.

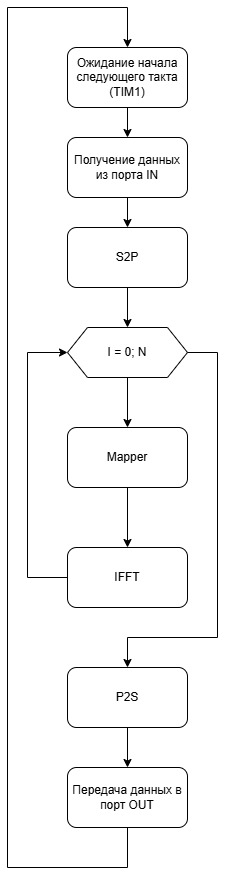


Рисунок **\*\*\*N\*\*\***. Блок-схема основной вычислительной функции

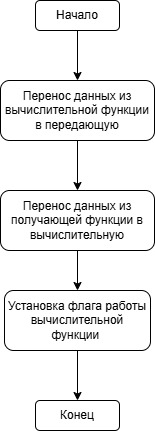


Рисунок **\*\*\*N\*\*\***. Блок-схема функции передачи данных между функциями

Несмотря на кажущуюся простоту, ранее отмечалась потенциальная проблема гонок данных. Однако, в данной реализации она решается за счет приоритетности и синхронизации. Функция передачи данных между функциями имеет наивысший приоритет прерывания, что гарантирует ее выполнение перед другими задачами. Требуется синхронизировать все функции таким образом, чтобы вычислительная функция всегда завершала вычисления до начала работы функции передачи данных, а передающая и принимающая функции успевали передать и получить все данные в отведенное им время. Эти скорости работы каждой функции рассчитываются заранее, исходя из необходимой скорости передачи данных и скорости входного битового потока, что обеспечивает предсказуемость и надежность работы системы.

# **Проверка алгоритма на микроконтроллерах**

Для тестирования на микроконтроллерах был выбран алгоритм с использованием библиотеки FFTW3 для микроконтроллеров: FFTW3-ARM, так как использование данной библиотеки показало наилучший результат по времени работы, что играет ключевую роль. Для тестирования алгоритма были выбраны 2 микроконтроллера на базе ядер Cortex-M3 и Cortex-M4: STM32F103RB и STM32F411RET.

Данные ядра являются одними из самых популярных в использовании как в учебных или личных целей, так и в различных повседневных устройствах, таких как умные часы (MI BAND 3 и MI BAND 4 построены на базе Cortex-M4, электронная книга PocketBook 626 на базе Cortex-M3). Оба ядра построены на 32-битной архитектуре, имеют набор инструкций Thumb-2 (совмещает в себе 16-битные инструкции для компактности кода и 32-битные инструкции для повышения производительности, обеспечивая баланс между размером кода и вычислительной мощностью), энергоэффективны, оптимизированы для работы с ограниченными ресурсами. Но Cortex-M4 содержит расширенный набор DSP-инструкций и блок FPU. DSP(Digital Signal Processing) инструкции – команды, предназначенные для обработки цифровых сигналов. Конкретно в Cortex-M4 были добавлены инструкции MAC (умножение с накоплением), SIMD (выполнение одной инструкции над несколькими данными), улучшенные инструкции битовых операций. Блок FPU позволяет аппаратно выполнять операции с плавающей точкой, без использования обычных инструкций для эмуляции операций с плавающей точкой.

Микроконтроллеры STM32F103RB и STM32F411RET помимо используемых ядер и их особенностей имеют следующие отличия:

* Максимальная частота работы ядра (до 72 МГц и до 100 МГц);
* Объем Flash-памяти и SRAM (128 КБ, 20 КБ и 512 КБ, 128 КБ).

Из вышеописанного можно сделать вывод, что STM32F411RET должен лучше справиться с поставленной задачей.

Проект конфигурировался с использованием STM32CubeMX. На обоих микроконтроллерах был подключен USART-порт для получения данных, модуль таймера для более точного отслеживания времени работы программы, установлено тактирование от внутренней RC-цепи, так как на данных микроконтроллерах не установлен кварцевый резонатор. Отличием в их конфигурациях является частота работы ядра: для STM32F103RB установлена частота 64 МГц (частота 72 МГц недоступна к выставлению в самой программе), а для STM32F411RET 100 МГц.

После конфигурации в режиме сборки Release была запущена одинаковая программа на обоих микроконтроллерах, которая считала время выполнения полного рабочего цикла программы на основе 10 000 000 повторов работы цикла. Были получены следующие результаты:

|  |  |  |
| --- | --- | --- |
| N | STM32F103RB | STM32F411RET |
| 8 | 12,85282723 | 6,327545711 |
| 16 | 32,48832461 | 15,99425211 |
| 20 | 40,89727749 | 19,1340443 |
| 32 | 69,2452356 | 34,08996214 |
| 40 | 94,61816754 | 46,58125171 |
| 64 | 144,9631414 | 71,36646959 |

Из результатов можно сделать вывод, что, для успешного выполнения поставленной задачи производительности микроконтроллера STM32F103RB хватает только для минимальных 8 гармоник, в то время как STM32F411RET успешно справляется с поставленной задачей в 20 гармоник.

# **Заключение**

В рамках данной работы был проведен анализ и реализована оптимизация алгоритма OFDM. Экспериментальным путем было выяснено, что микроконтроллер STM32F411RET на базе ядра Cortex-M4 подходит для выполнения поставленной задачи. Следующим шагом является использование полученных данных для оптимального подбора компонентов с целью реализации более быстрого и эффективного устройства приема и передачи данных с использованием OFDM.

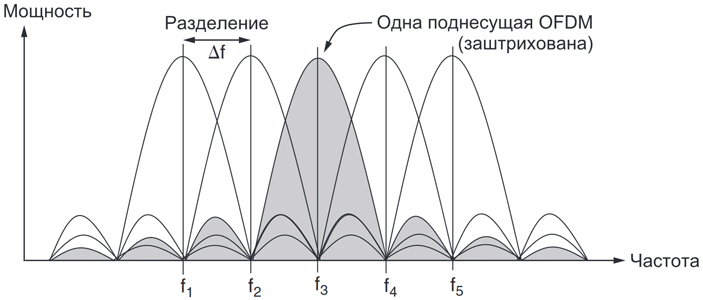
# **Приложение 1**

## **Принцип работы приемопередатчика на основе OFDM**

Прежде чем перейти к детальному рассмотрению алгоритмов формирования и обработки сигналов OFDM на микропроцессорной платформе, необходимо разобрать теоретические основы данной технологии модуляции.

Ортогональное частотное мультиплексирование (OFDM) - это метод цифровой модуляции, который делит широкополосный канал связи на множество узкополосных, ортогональных поднесущих. Вместо передачи одного высокоскоростного потока данных по всему каналу, OFDM передает несколько низкоскоростных потоков данных параллельно по этим поднесущим.

В основе OFDM лежит принцип разделения полосы пропускания на множество узкополосных поднесущих, каждая из которых модулируется независимо. Эти поднесущие выбираются таким образом, чтобы они были ортогональны друг другу. Ортогональность означает, что в точке максимальной амплитуды одной поднесущей амплитуда всех остальных поднесущих равна нулю. Это позволяет поднесущим перекрываться по частоте, не создавая взаимных помех.



Основной принцип работы OFDM заключается в следующем:

1. Высокоскоростной поток данных делится на несколько параллельных низкоскоростных потоков;
2. Каждый низкоскоростной поток данных модулирует свою поднесущую. Поднесущие выбираются таким образом, чтобы они были ортогональными друг другу;
3. Все модулированные поднесущие суммируются вместе с использованием обратного быстрого преобразования Фурье (IFFT). Это создает OFDM-сигнал во временной области;
4. К каждому OFDM-символу добавляется циклический префикс, который представляет собой копию части OFDM-символа, добавленную в начало. CP помогает бороться с межсимвольной интерференцией и межподнесущей интерференцией, вызванными многолучевым распространением.

На приемной стороне выполняются обратные операции: удаление CP, быстрое преобразование Фурье (FFT), демодуляция поднесущих и объединение низкоскоростных потоков данных в высокоскоростной поток.

Преимущества и недостатки OFDM:

Преимущества:

* OFDM эффективно борется с многолучевым распространением, так как узкополосные поднесущие менее подвержены частотно-избирательному замиранию (явление, при котором разные частотные компоненты сигнала затухают по-разному из-за многолучевого распространения);
* Так как поднесущие ортогональны друг другу, данные плотно упаковываются в частотный диапазон;
* Простота реализации с использованием БПФ/ОБПФ;
* OFDM позволяет адаптировать параметры передачи (модуляцию, кодирование) к условиям канала для каждой поднесущей.

Недостатки:

* Небольшие отклонения частоты могут привести к потере ортогональности между поднесущими и ухудшению производительности системы.
* OFDM-сигнал может иметь высокий пик-фактор, что требует использования линейных усилителей мощности.
* Фазовый шум может привести к межподнесущей интерференции (ICI).

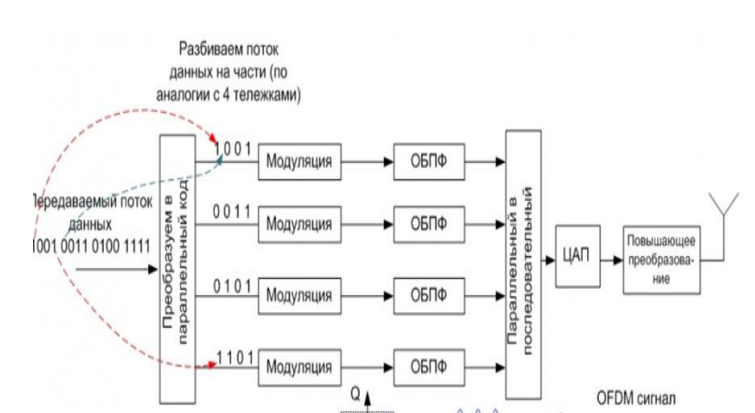
Пример:

Представим, что существует полоса частот шириной 1 МГц, которую необходимо использовать для передачи данных. Вместо того чтобы использовать один широкий канал, эта полоса разделяется, допустим, на 128 узких каналов (поднесущих) шириной примерно 7.8 кГц каждый. Каждая из этих поднесущих модулируется независимо с использованием определенного алгоритма модуляции.

Таким образом, вместо передачи одного символа с высокой скоростью передачи данных, передается 128 символов с более низкой скоростью передачи данных на каждой поднесущей. Это снижает влияние многолучевого распространения, так как узкополосные поднесущие менее подвержены частотно-избирательному замиранию (явление, при котором разные частотные компоненты сигнала затухают по-разному из-за многолучевого распространения).

В итоге, суммарная скорость передачи данных остается прежней, но система становится более устойчивой к помехам и многолучевому распространению.

Пример строения блоков:



## **Существующие реализации**

Пример 1

“Hybrid Analog and Digital Beamforming for mmWave OFDM Large-Scale Antenna Arrays” (“Гибридное аналого-цифровое формирование луча для OFDM-систем миллиметрового диапазона с крупномасштабными антенными решетками”):

Рассмотрим крупномасштабную MIMO-систему на основе OFDM, в которой передатчик, оснащенный Nt антеннами, обслуживает приемник, оснащенный Nr антеннами, отправляя Ns символов данных на каждый частотный тон. В общем случае, количество символов данных может быть различным для разных частотных тонов, однако для простоты, в этой статье внимание сосредоточено на случае с равным количеством потоков данных для всех поднесущих, поскольку для систем миллиметрового диапазона с сильно коррелированными каналами все подканалы обычно имеют низкий ранг. Более того, в практических крупномасштабных MIMO-системах количество доступных РЧ-цепей1, NRF, обычно намного меньше, чем количество антенн приемопередатчика, т.е. NRF min(Nt; Nr). Это исключает возможность реализации традиционных полностью цифровых методов формирования луча, которые требуют одной РЧ-цепи на каждый антенный элемент. В этой статье используется архитектура гибридного аналого-цифрового формирования луча, показанную на рисунке 1, для решения проблемы ограничения аппаратных ресурсов. В архитектуре гибридного формирования луча общий формирователь луча состоит из низкоразмерного цифрового (в полосе основной частоты) формирователя луча и высокоразмерного аналогового (радиочастотного) формирователя луча, реализованного с использованием простых аналоговых компонентов.

**A. Модель сигнала в гибридном формировании луча**

В архитектуре гибридного формирования луча на основе OFDM, показанной на рисунке 1, передатчик сначала предварительно кодирует Ns символов данных s[k] на каждой поднесущей k = 1, … , K, используя низкоразмерный цифровой предварительный кодировщик, VD[k] ∈ CNRF×Ns, затем преобразует сигналы во временную область, используя NRF K-точечных обратных быстрых преобразований Фурье (IFFT). После добавления циклических префиксов, передатчик использует аналоговую матрицу предварительного кодирования VRF ∈ CNt×NRF, чтобы сгенерировать финальный передаваемый сигнал. Поскольку аналоговый предварительный кодировщик является модулем, следующим за IFFT, аналоговый предварительный кодировщик идентичен для всех поднесущих. Это является ключевой проблемой при проектировании гибридных формирователей луча в OFDM системах по сравнению с однонесущими системами. Учитывая это, финальный передаваемый сигнал на поднесущей k равен:

x[k] = VRF VD[k] s[k], (1)\*

где s[k] ∈ CNs×1 — вектор передаваемых символов данных на поднесущей k с E{s[k]s[k]H} = INs.

Предполагая модель блочно-замирающего канала, полученный сигнал на поднесущей k равен:

y[k] = H[k] VRF VD[k] s[k] + z[k], (2)\*

где H[k] ∈ CNr×Nt и z[k] ∼ CN (0, σ2INr) являются, соответственно, матрицей канала и аддитивным белым гауссовским шумом для поднесущей k.

На приемнике, полученные сигналы всех поднесущих изначально обрабатываются с использованием аналогового объединителя, WRF ∈ CNr×NRF. Затем циклический префикс удаляется, и NRF K-точечные быстрые преобразования Фурье (FFT) применяются для восстановления сигналов в частотной области. Наконец, используя низкоразмерный цифровой объединитель для каждой поднесущей, WD[k] ∈ CNRF×Ns, приемник получает финальный обработанный сигнал в виде:

y˜[k] = WtH[k] H[k] Vt[k] s[k] + WtH[k] z[k], (3)\*

в котором Vt[k] = VRF VD[k] и Wt[k] = WRF WD[k] являются общим гибридным предварительным кодировщиком и объединителем для k-ой поднесущей, соответственно.

• s[k]: Вектор символов данных, которые нужно передать на k-ой поднесущей. E{s[k]s[k]H} = INs означает, что символы некоррелированы и имеют единичную мощность. INs - единичная матрица размера Ns x Ns.

• VD[k]: Матрица цифрового предварительного кодирования (digital precoder) для k-ой поднесущей. Эта матрица применяется в цифровой области после модуляции OFDM. Размерность NRF x Ns означает, что она преобразует Ns входных потоков в NRF потоков для отправки через РЧ-цепи.

• VRF: Матрица аналогового предварительного кодирования (analog precoder). Важный момент: она одинакова для всех поднесущих. Это ограничение связано с тем, что аналоговая часть формирователя луча работает после IFFT, то есть во временной области, где информация о поднесущих уже смешана. Размерность Nt x NRF означает, что она преобразует NRF потоков, выходящих из РЧ-цепей, в Nt сигналов для отправки через антенны.

• x[k]: Передаваемый сигнал на k-ой поднесущей.

• H[k]: Матрица канала для k-ой поднесущей. Описывает, как сигнал распространяется от передатчика к приемнику на этой частоте. Размерность Nr x Nt означает, что она отображает Nt передаваемых сигналов на Nr принимаемых сигналов.

• z[k]: Аддитивный белый гауссовский шум (AWGN) на k-ой поднесущей. CN (0, σ2INr) означает, что шум имеет комплексное нормальное распределение со средним значением 0 и дисперсией σ2 на каждой принимающей антенне.

• y[k]: Принимаемый сигнал на k-ой поднесущей.

• WRF: Матрица аналогового объединения (analog combiner) на приемнике. Аналогично VRF, она одинакова для всех поднесущих. Размерность Nr x NRF означает, что она преобразует Nr принимаемых сигналов в NRF потоков для обработки в РЧ-цепях.

• WD[k]: Матрица цифрового объединения (digital combiner) для k-ой поднесущей. Размерность NRF x Ns означает, что она преобразует NRF потоков, выходящих из РЧ-цепей, в Ns выходных потоков данных.

• y˜[k]: Финальный обработанный сигнал на k-ой поднесущей.

• Vt[k]: Общий гибридный предварительный кодировщик.

• Wt[k]: Общий гибридный объединитель.

• Блочно-замирающий канал: Модель канала, в которой канал остается постоянным в течение некоторого времени (блока), а затем изменяется случайным образом.



**B. Структура аналогового формирователя луча**

Аналоговая часть гибридного формирователя луча обычно реализуется с использованием простых аналоговых компонентов, таких как аналоговые сумматоры и аналоговые фазовращатели, которые могут изменять только фазу сигналов. Это приводит к некоторым ограничениям на матрицу аналогового формирования луча в зависимости от структуры аналогового формирователя луча. В этой статье мы сосредоточимся на двух широко используемых структурах аналогового формирования луча: полностью-связанной и частично-связанной структурах.

• Полностью-связанная архитектура: В этой структуре каждая РЧ-цепь соединена со всеми антенными элементами через сеть фазовращателей, как показано на рисунке 2(a). Это приводит к ограничению постоянного модуля нормы на все элементы матриц аналогового формирования луча, то есть |VRF(i, j)| = |WRF(i, j)| = 1, ∀i, j. Кроме того, как видно на рисунке 2(a), общее количество фазовращателей в этой архитектуре составляет NtNRF.

• Частично-связанная архитектура: В отличие от полностью-связанной структуры, в частично-связанной структуре каждая РЧ-цепь соединена только с одной подрешеткой с Nt/ NRF антеннами, как показано на рисунке 2(b). Следовательно, матрица аналогового формирования луча в частично-связанной архитектуре имеет блочно-диагональный формат, то есть аналоговый прекодер имеет вид:

VRF =

[ v1 0 ... 0 ]

[ 0 v2 0 ]

[ 0 0 ... ]

[ . . 0 ]

[ 0 0 ... vNRF ]

(4)

где каждый элемент вектора vi удовлетворяет ограничению постоянного модуля. Общее количество фазовращателей в этой структуре составляет Nt, что означает, что сложность аппаратной части РЧ-формирователя луча снижается в NRF раз.

• Analog Adders (Аналоговые сумматоры): Компоненты, которые складывают аналоговые сигналы.

• Analog Phase Shifters (Аналоговые фазовращатели): Компоненты, которые изменяют фазу аналогового сигнала. Они являются ключевыми компонентами для формирования луча, так как позволяют управлять направлением излучения/приема антенной решетки.

• Fully-connected Architecture (Полностью-связанная архитектура): Каждая РЧ-цепь соединена со всеми антеннами. Это обеспечивает большую гибкость в формировании луча, но требует большего количества фазовращателей.

• Partially-connected Architecture (Частично-связанная архитектура): Каждая РЧ-цепь соединена только с подмножеством антенн (подрешеткой). Это упрощает аппаратную реализацию (меньше фазовращателей), но может снижать гибкость формирования луча.

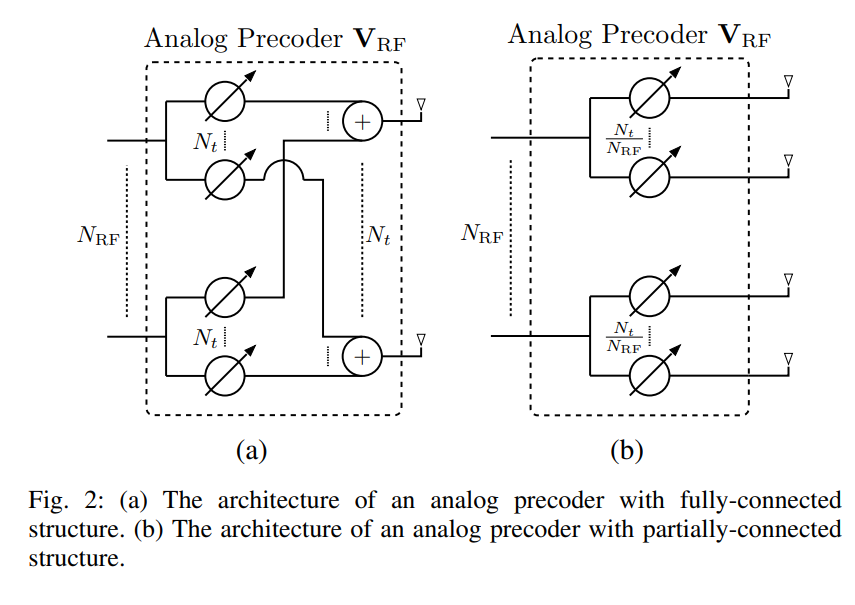
• Constant Modulus Constraint (Ограничение постоянного модуля): |VRF(i, j)| = |WRF(i, j)| = 1. Это означает, что амплитуда сигнала на выходе фазовращателя остается неизменной (равной 1), изменяется только его фаза. Это упрощает проектирование фазовращателей.

• NtNRF: Общее количество фазовращателей в полностью-связанной архитектуре.

• Block Diagonal Format (Блочно-диагональный формат): Матрица, в которой элементы за пределами диагональных блоков равны нулю.

• Nt/NRF: Количество антенн в каждой подрешетке в частично-связанной архитектуре.

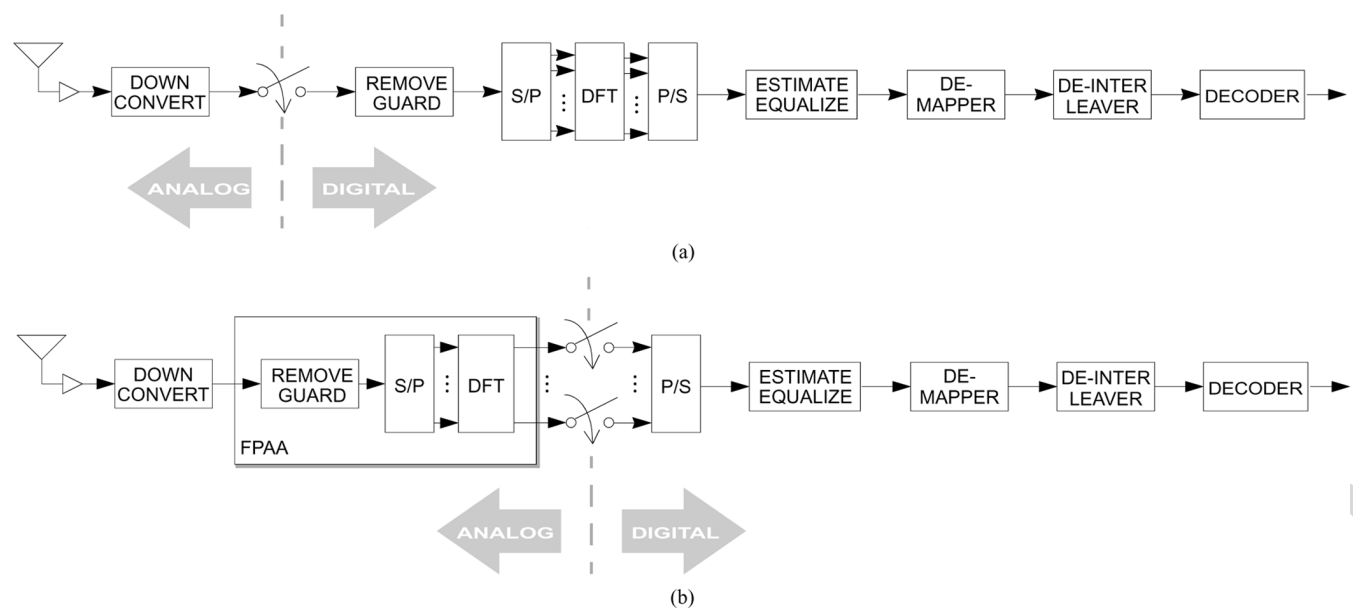
• Nt: Общее количество фазовращателей в частично-связанной архитектуре.



В 19 показано, что существует компромисс между производительностью и сложностью при выборе вышеупомянутых структур. Полностью-связанная структура может достичь полного усиления формирования луча с полным контролем фазы, в то время как частично-связанная структура имеет ограниченный контроль фазы, и, следовательно, не может достичь полного усиления формирования луча во всех случаях. С другой стороны, сложность аппаратной реализации и энергопотребление частично-связанной архитектуры значительно ниже по сравнению с таковыми у полностью-связанной структуры.

**Пример 2**

Low-Power Discrete Fourier Transform for OFDM: A Programmable Analog Approach /Малопотребляющее дискретное преобразование Фурье для OFDM: Программируемый аналоговый подход



*Рисунок 1. OFDM приемник. (a) Традиционная реализация, в которой дискретизация происходит перед цифровым ДПФ. (b) Предлагаемая реализация, в которой дискретизация происходит после аналогового ДПФ.*

На рисунке 1(a) мы показываем упрощенную блок-схему обычного OFDM-приемника, например, для системы 802.11 a/g, где принятый сигнал дискретизируется сразу после понижения частоты. Демодуляция OFDM выполняется цифровым способом с использованием ДПФ (DFT).

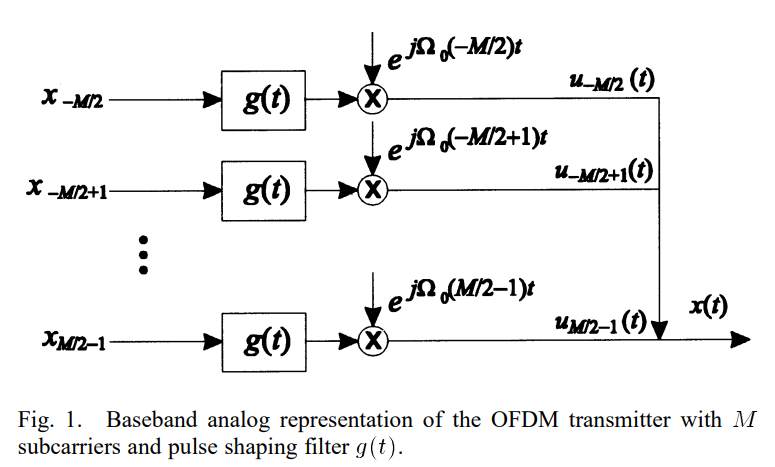
В качестве альтернативы предлагается аналоговая реализация, как показано на рисунке 1(b), где выход понижающего преобразователя подается непосредственно на FPAA (программируемую аналоговую матрицу), без дискретизации. Функциональность ДПФ реализуется аналоговым способом с использованием FPAA. Выходы FPAA — по одному на каждую поднесущую — дискретизируются отдельно.

Помимо снижения энергопотребления при аналоговой реализации ДПФ, важным преимуществом предлагаемой структуры приемника на рисунке 1(b) является значительное облегчение требований к скорости и точности аналого-цифрового преобразователя (АЦП). В частности, АЦП обычного приемника, показанного на рисунке 1(a), должен дискретизировать сигнал со скоростью, равной полной полосе пропускания сигнала, а его точность должна быть высокой (порядка 10 бит и более), чтобы соответствовать широкому динамическому диапазону и гауссовскому распределению OFDM-сигналов. В отличие от этого, предлагаемый приемник на рисунке 1(b) имеет не один АЦП, а N АЦП, по одному на каждую поднесущую, с частотой дискретизации, меньшей в N раз. Более того, каждый выход ДПФ представляет собой сигнал с конечным алфавитом, который может быть дискретизирован со значительно меньшей битовой точностью 14. По сравнению с АЦП на рисунке 1(a), каждый АЦП на рисунке 1(b) требует частоты дискретизации, которая ниже в N раз, и количества битов точности, которое меньше в три раза или больше, в зависимости от размера алфавита модуляции. Оба эффекта приводят к снижению энергопотребления АЦП, хотя точная величина зависит от типа структуры АЦП; энергопотребление АЦП находится между линейной и квадратичной функцией частоты дискретизации 19.

Таким образом, предлагаемый приемник на рисунке 1(b) выгоден не только из-за снижения энергопотребления при аналоговой реализации ДПФ, но и из-за дополнительной экономии энергии, возникающей из-за более низких требований к скорости и битовой точности последующего АЦП. Хотя здесь сосредоточены на стороне приемника, кратко отмечается, что те же преимущества справедливы и на стороне передатчика, где выходы от модулятора символов модулируются с помощью блока обратного ДПФ (IDFT), который имеет ту же структуру, что и блок ДПФ, но с другими коэффициентами. Помимо экономии энергии при аналоговой реализации IDFT, перенос блока IDFT после цифро-аналогового преобразователя (ЦАП) позволяет OFDM-передатчику заменить полноскоростной высокоточный ЦАП на N отдельных ЦАП, каждый из которых работает на частоте в 1/N раз ниже и с меньшей битовой точностью.

• FPAA (Field-Programmable Analog Array) / ПААМ (Программируемая аналоговая матрица): Интегральная схема, содержащая конфигурируемые аналоговые блоки, такие как операционные усилители, резисторы, конденсаторы и т. д., которые могут быть соединены для реализации различных аналоговых функций.

**Пример 3**

OFDM Transmitters: Analog Representation and DFT-Based Implementation” (“OFDM передатчики OFDM: Аналоговое представление и реализация на основе ДПФ”):

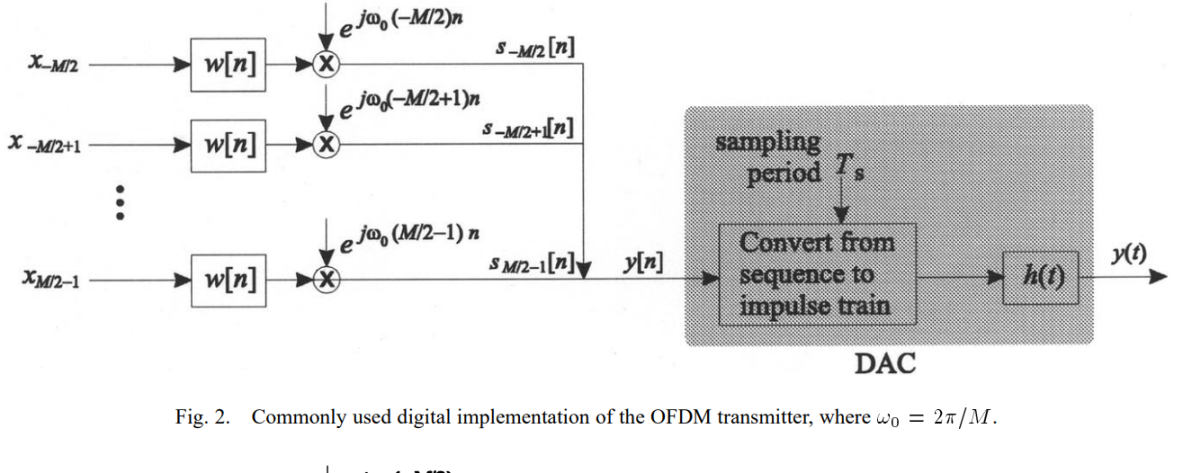
На рисунке 1 показана схема аналогового OFDM-передатчика с M поднесущими. Обозначив расстояние между поднесущими как Ω0, выход передатчика задается выражением:

x(t) = Σk=-M/2M/2-1 xk g(t) ejkΩ0t (1)

предполагая, что M четное. Фильтр формирования импульса g(t) обычно представляет собой прямоугольный импульс длительностью T0 = 2π/Ω0. Многие исследования OFDM-систем проводятся с использованием выражения (1), например, исследование спектрального спада выходных сигналов OFDM-передатчиков 3, 4, влияние смещения несущей частоты 5 и пик-факторов выходных сигналов передатчика 6. Был предложен ряд непрямоугольных форм импульсов g(t) для улучшения спектрального спада передаваемого сигнала x(t), например, в 4 и 6. Хотя аналоговое представление удобно для анализа, на практике модуляция поднесущих осуществляется в дискретном времени. Такой передатчик (см. рисунок 2) состоит из двух частей 3: цифро-аналогового преобразователя (ЦАП) и части, выполняющей цифровую модуляцию поднесущих, которая может быть эффективно реализована с использованием матрицы обратного дискретного преобразования Фурье (ОДПФ) размером M x M. Период дискретизации равен Ts = T0/ M, и дискретная последовательность w[n], показанная на рисунке 2, обычно представляет собой прямоугольное окно длины M.

Предположим, что фильтр восстановления ЦАП имеет импульсную характеристику h(t), как показано на рисунке 2. Выход ЦАП с периодом дискретизации Ts задается выражением:

y(t) = Σn=-∞∞ y[n] h(t - nTs)

где y[n] = w[n] Σk=-M/2M/2-1 xk ej(2π/M)kn: (2)

Как указано на рисунке 2, y[n] является входом ЦАП. Форма волны y(t) напоминает форму волны x(t) — выходного сигнала аналогового передатчика — особенно при больших значениях M, 1. В [3, гл. 5] упоминается, что при использовании цифровой реализации с идеальным низкочастотным фильтром восстановления h(t) в ЦАП, формирующий фильтр g(t) больше не является прямоугольным импульсом. В литературе ранее не было указано точной связи между передатчиком на основе ДПФ и аналоговым представлением.

• x(t): Аналоговое представление OFDM-сигнала во времени.

• M: Количество поднесущих.

• Ω0: Расстояние между поднесущими (в радианах в секунду).

• xk: Комплексный символ данных, модулирующий k-ую поднесущую.

• g(t): Импульсная характеристика формирующего фильтра. Определяет форму импульса каждой поднесущей во времени. Прямоугольный импульс является наиболее распространенным, но другие формы могут улучшить спектральные характеристики.

• ejkΩ0t: Комплексная экспонента, представляющая k-ую поднесущую.

• w[n]: Окно, применяемое к дискретным данным перед преобразованием в аналоговый сигнал.

• y[n]: Дискретные отсчеты, подаваемые на вход ЦАП.

• y(t): Аналоговый сигнал на выходе ЦАП.

• h(t): Импульсная характеристика фильтра восстановления ЦАП. Этот фильтр сглаживает ступенчатый выход ЦАП, чтобы приблизиться к исходному аналоговому сигналу.

• Ts: Период дискретизации.

В этой статье рассматриваются условия, при которых передатчик на основе ДПФ, показанный на рисунке 2, допускает аналоговое представление, показанное на рисунке 1. Для случая передатчика на основе ДПФ с прямоугольным окном w[n] и идеальным низкочастотным фильтром h(t) аналоговое представление не существует. Известно, что, если мы выберем прямоугольное окно g(t) в аналоговом представлении, выходной сигнал будет близок к выходному сигналу передатчика на основе ДПФ во временном окне, но выходные сигналы двух передатчиков могут иметь значительные различия в спектральном спаде. Покажем, что на самом деле, когда аналоговый передатчик имеет прямоугольный g(t), реализация на основе ДПФ не существует, независимо от выбора w[n] и h(t). Для произвольного импульса g(t) эквивалентная цифровая реализация, вообще говоря, не существует. Аналоговый и передатчик на основе ДПФ эквивалентны только в некоторых ограниченных случаях. Поэтому анализ OFDM-систем непосредственно с использованием схемы на основе ДПФ, показанной на рисунке 2, более полезен, чем использование аналоговой схемы, показанной на рисунке 1. Например, разработка w[n] и h(t) для улучшения спектрального спада более полезна, чем разработка g(t). Будет выведено необходимое и достаточное условие эквивалентности аналогового и передатчика на основе ДПФ. Будет приведен пример набора g(t), w[n] и h(t), удовлетворяющих этому условию.

Разбор и пояснения:

• Эквивалентность аналогового и цифрового представлений: Основная тема статьи - выяснить, когда аналоговый и цифровой (на основе ДПФ) OFDM-передатчики дают одинаковый результат.

• Прямоугольное окно w[n] и идеальный фильтр h(t): Это часто используемые упрощения в анализе, но в этой статье утверждается, что в этом случае не существует эквивалентного аналогового представления.

• Спектральный спад: Скорость уменьшения мощности сигнала по мере удаления от несущей частоты. Хороший спектральный спад важен для уменьшения помех соседним каналам.

• Разработка w[n] и h(t) вместо g(t): В статье утверждается, что для улучшения характеристик OFDM-передатчика лучше настраивать окно w[n] в цифровой части и фильтр восстановления h(t) в ЦАП, чем пытаться оптимизировать формирующий фильтр g(t) в аналоговом представлении.

• Необходимое и достаточное условие: Статья обещает вывести математическое условие, которое должно выполняться, чтобы аналоговый и цифровой передатчики были эквивалентны.

# **Приложение 2.**

### **Модуляция**

Модуляция поднесущих - это процесс изменения одного или нескольких параметров несущего сигнала (амплитуды, частоты, фазы) в соответствии с информацией, которую необходимо передать. В OFDM модуляция выполняется на каждой поднесущей независимо, что позволяет передавать несколько параллельных потоков данных. Выбор метода модуляции влияет на скорость передачи данных, устойчивость к шуму и сложность системы.

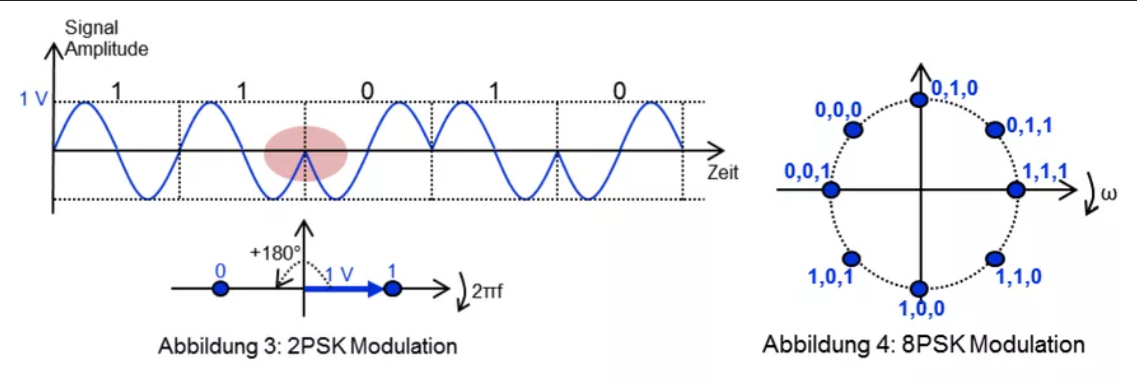
Рассмотрим наиболее распространенные методы модуляции, используемые в OFDM:

**Фазовая модуляция (Phase-Shift Keying - PSK)**

Данные кодируются путем изменения фазы несущей частоты. Амплитуда остается постоянной.

Типы PSK:

* BPSK (Binary PSK): Использует две фазы (0 и 180 градусов) для представления битов 0 и 1.
* QPSK (Quadrature PSK): Использует четыре фазы (0, 90, 180 и 270 градусов) для представления двух битов информации на символ.
* 8-PSK: Использует восемь фаз для представления трех битов на символ.
* 16-PSK: Использует шестнадцать фаз для представления четырех битов на символ.



Пример в виде алгоритма QPSK:

* Входящий поток битов делится на пары.
* Каждой паре битов присваивается определенная фаза:

00 -> 45°

01 -> 135°

10 -> 225°

11 -> 315°

* Сигнал генерируется как: , где - амплитуда, - частота поднесущей, - фаза, соответствующая дибиту.

Преимущества: Простота реализации, постоянная огибающая (что упрощает работу усилителей мощности).

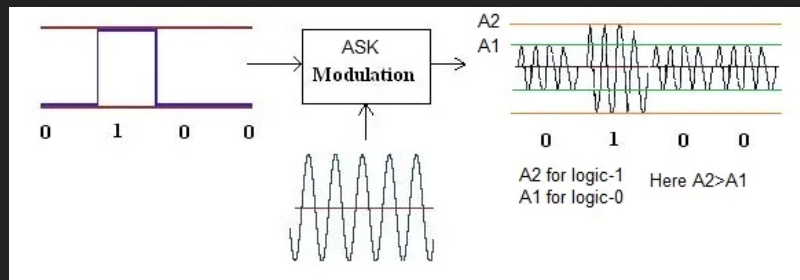
Недостатки: Ограниченная спектральная эффективность (малое количество битов на символ), чувствительность к фазовому шуму.

Вычислительные ресурсы: Низкие. Требуются простые операции для генерации синусоидальных сигналов с определенной фазой.

**Амплитудная модуляция (Amplitude-Shift Keying - ASK)**

Данные кодируются путем изменения амплитуды несущей частоты. Фаза остается постоянной.

* OOK (On-Off Keying): Простейший тип ASK, где наличие сигнала представляет бит 1, а отсутствие сигнала - бит 0.



Пример алгоритма:

Бит 1: (сигнал есть)

Бит 0: s(t) = 0 (сигнала нет)

Преимущества: Очень простая реализация.

Недостатки: Низкая спектральная эффективность, очень чувствительна к шуму и изменениям уровня сигнала. Практически не используется в OFDM.

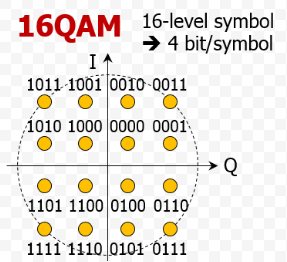
Вычислительные ресурсы: Очень низкие.

**Квадратурная амплитудная модуляция (Quadrature Amplitude Modulation - QAM)**

Данные кодируются путем изменения как амплитуды, так и фазы несущей частоты.

Типы QAM:

* 16-QAM: Использует 16 комбинаций амплитуды и фазы для представления четырех битов на символ.
* 64-QAM: Использует 64 комбинации амплитуды и фазы для представления шести битов на символ.
* 256-QAM: Использует 256 комбинаций амплитуды и фазы для представления восьми битов на символ.



Пример алгоритма 16-QAM:

1. Входящий поток битов делится на группы по 4 бита.
2. Каждой группе битов присваивается определенная амплитуда и фаза в соответствии с картой созвездия.
3. Сигнал генерируется как сумма двух квадратурных компонент: где *I* и - синфазная и квадратурная компоненты, определяемые амплитудой и фазой.

Преимущества: Высокая спектральная эффективность (большое количество битов на символ).

Недостатки: Более сложная реализация, чувствительность к шуму и искажениям, требует линейных усилителей мощности.

Вычислительные ресурсы: Средние. Требуются более сложные операции для генерации сигналов с переменной амплитудой и фазой.

|  |  |  |  |
| --- | --- | --- | --- |
| Метод | Биты | Устойчивость к шуму | Вычислительные ресурсы |
| BPSK | 1 | Высокая | Низкие |
| QPSK | 2 | Средняя | Низкие |
| 16-QAM | 4 | Средняя | Средние |
| 64-QAM | 6 | Низкая | Средние |
| 256-QAM | 8 | Очень низкая | Высокие |

Таблица 1. Сравнение методов модуляции

Выбор метода модуляции для каждой поднесущей в OFDM зависит от:

* Состояния канала: Если канал характеризуется высоким SNR, можно использовать более сложные схемы модуляции (например, 64-QAM) для увеличения скорости передачи данных. Если канал зашумлен, следует использовать более надежные схемы (например, QPSK или BPSK).
* Требований к скорости передачи данных: Если требуется высокая скорость передачи данных, используются схемы модуляции с большим количеством битов на символ.
* Требований к надежности: Если требуется высокая надежность передачи данных, используются схемы модуляции с меньшим количеством битов на символ и канальным кодированием.

# **Приложение 3.**

## **Алгоритмы БПФ.**

Преобразование Фурье является фундаментальным инструментом в цифровой обработке сигналов. Оно позволяет разложить сигнал во временной области на его составляющие частоты, предоставляя информацию о частотном спектре сигнала. Дискретное преобразование Фурье является дискретной версией преобразование Фурье и применяется к дискретным сигналам, полученным, например, в результате дискретизации аналогового сигнала.

Однако, непосредственное вычисление ДПФ требует большого количества вычислительных операций (порядка O(N2) для сигнала длиной N), что делает его неэффективным для обработки больших объемов данных. Быстрое преобразование Фурье - это класс алгоритмов, разработанных для эффективного вычисления ДПФ. БПФ значительно снижает вычислительную сложность до O(N\*log(N)), что делает его применимым для обработки сигналов в реальном времени и в различных приложениях.

Для дешифровки требуется дискретное преобразование Фурье последовательности x(n) длиной N определяется как:

для k = 0, 1, ..., N-1, где n = 0 до N-1,

- n-ый отсчет входной последовательности (сигнала во временной области);

- k-ый отсчет выходной последовательности (спектр сигнала в частотной области);

- длина последовательности;

Обратное дискретное преобразование Фурье (ОДПФ) определяется как:

для n = 0, 1, ..., N-1, где k = 0 до N-1.

Цель преобразования Фурье – это получить коэффициенты ряда Фурье. Общий вид коэффициентов ряда Фурье представлен ниже.

**Алгоритм Кули-Тьюки (Cooley-Tukey Algorithm**)

Является наиболее известным и часто используемым алгоритмом БПФ. Разделяет ДПФ на более мелкие ДПФ, которые затем рекурсивно вычисляются.

Наиболее эффективен, когда длина последовательности N является степенью двойки (radix-2 БПФ). Существуют также варианты с другими основаниями (radix-4, radix-8 и т.д.).

Имеет вычислительную сложность O(Nlog(N)).

**Алгоритм Radix-2 (основание 2)**

Алгоритм Radix-2 БПФ является частным случаем алгоритма Кули-Тьюки, который наиболее эффективно работает, когда длина входной последовательности N является степенью двойки (N = 2m, где m - целое число). Он основан на декомпозиции ДПФ на более мелкие ДПФ размером 2 (бабочки). Алгоритм рекурсивно разделяет ДПФ на две половины: четные и нечетные индексы, пока не достигнет ДПФ размером 2.

1. Переупорядочивание входной последовательности x[n] в соответствии с обращенным битовым представлением индексов.
2. Рекурсивное вычисление ДПФ размером 2 (бабочки) на каждой стадии. «Бабочка» - это базовая вычислительная единица, выполняющая следующие операции:

где: A и B - входы “бабочки” (результаты предыдущей стадии).

- вращающийся множитель.

N - размер ДПФ на текущей стадии.

1. Повторение шага 2 для каждой стадии рекурсии, пока не будут вычислены все значения ДПФ. Количество стадий равно log2(N).

Вычислительная сложность: O(N log2 N) комплексных умножений и O(N log2 N) комплексных сложений.

Отличия от Radix-4 и Radix-8:

Radix-4: Декомпозирует ДПФ на более мелкие ДПФ размером 4. Требует, чтобы длина последовательности была степенью 4 (N = 4m). Выполняет более сложные “бабочки”, но требует меньше стадий (log4 N).

Radix-8: Декомпозирует ДПФ на более мелкие ДПФ размером 8. Требует, чтобы длина последовательности была степенью 8 (N = 8m). Еще более сложные “бабочки” и еще меньше стадий (log8 N).

В общем случае, алгоритмы с большим основанием (Radix-4, Radix-8) могут быть более эффективными, чем Radix-2, за счет уменьшения количества стадий. Однако, они требуют более сложной реализации “бабочек” и более строгих ограничений на длину последовательности.

**Алгоритм БПФ Бэтчера (Bluestein’s FFT Algorithm)**

Алгоритм Бэтчера используется для вычисления ДПФ последовательностей, длина которых не является степенью двойки (или любого другого удобного основания для алгоритма Кули-Тьюки). Он основан на преобразовании ДПФ в операцию свертки, которую можно эффективно вычислить с использованием БПФ.

Алгоритм:

1. ДПФ выражается в виде свертки с использованием следующей замены:

Подставляя это в формулу ДПФ, получаем:

Это можно переписать как свертку:

где: n = 0 до N-1;

1. Вычисляются БПФ последовательностей и . Затем выполняется поэлементное умножение полученных спектров, и к результату применяется обратное БПФ.
2. Результат свертки умножается на для получения окончательного результата ДПФ.

Вычислительная сложность: O(N\*log N), так как основная часть вычислений приходится на БПФ и ОБПФ. Однако, константа перед N\*log N может быть выше, чем у алгоритма Кули-Тьюки для последовательностей, длина которых является степенью двойки.

Применяется, когда длина входной последовательности не является степенью двойки или когда требуется использовать БПФ фиксированного размера (например, из-за ограничений оборудования).

**Алгоритм Винограда(Радера) (Rader’s Algorithm)**

Алгоритм Радера используется для вычисления ДПФ последовательностей, длина которых является простым числом (N - простое число). Он также преобразует ДПФ в свертку, которую можно эффективно вычислить с использованием БПФ.

Алгоритм:

1. Поскольку N - простое число, можно найти образующий элемент *g* мультипликативной группы *ZN* \* (множество целых чисел от 1 до N-1, взаимно простых с N, с операцией умножения по модулю N). Это означает, что все числа от 1 до N-1 можно представить в виде степеней *g* по модулю N.

Выполняется замена индексов:

где n’, k’ = 1, 2, …, N-1.

1. После переиндексации формула ДПФ может быть преобразована в циклическую свертку:

Пусть , , тогда

где n' = 1 до N-1.

Последняя сумма является циклической сверткой y и h.

1. Вычисление свертки с использованием БПФ: Вычисляются БПФ последовательностей и . Затем выполняется поэлементное умножение полученных спектров, и к результату применяется обратное БПФ.
2. К результату свертки добавляется для получения окончательного результата ДПФ.

Вычислительная сложность: O(N\*log N) из-за использования БПФ для вычисления свертки. Константа перед N\*log N может быть выше, чем у алгоритма Кули-Тьюки.

Применение: Когда длина входной последовательности является простым числом.

**Применение БПФ для реализации OFDM**

БПФ играет ключевую роль в реализации OFDM. Он используется как для модуляции, так и для демодуляции сигнала.

Модуляция (Передатчик):

* Последовательность комплексных символов X[k], представляющих данные для каждой поднесущей, подается на вход обратного быстрого преобразования Фурье.
* ОБПФ преобразует частотное представление сигнала X[k] во временное представление x[n], создавая OFDM-символ.
* Выход ОБПФ представляет собой дискретный OFDM-сигнал во временной области.
* Добавление циклического префикса (CP) к OFDM-символу.

Демодуляция (Приемник):

* Принятый OFDM-сигнал (после удаления циклического префикса) подается на вход быстрого преобразования Фурье (БПФ или FFT).
* БПФ преобразует временное представление сигнала обратно в частотное представление, восстанавливая символы X[k] на каждой поднесущей.
* Символы демодулируются для получения исходных данных.

Преимущества использования БПФ в OFDM:

Эффективность: БПФ значительно снижает вычислительную сложность модуляции и демодуляции OFDM-сигналов, делая возможной реализацию систем OFDM с большим количеством поднесущих.

Простота реализации: БПФ является хорошо изученным и реализованным алгоритмом, для которого существуют оптимизированные библиотеки и аппаратные реализации.

# **Приложение 4.**

## **Шаблон программы вычисления времени работы и потребляемых ресурсов**

#include <iostream>

#include <chrono>

#include <sys/resource.h>

int N;                 // Число гармоник

double normalizeCoef;       // Нормализующий коэффициент

int COUNT\_TEST;    // Число тестов

int main(int argc, char \*\* argv)

{

    std::chrono::system\_clock::time\_point startFunc, endFunc, startProgram;

    std::chrono::duration<double> srTime;

    struct rusage dataStatus;

    startProgram = std::chrono::system\_clock::now();

    N = atoi(argv[1]);

    normalizeCoef = 1/sqrt(N);

    COUNT\_TEST = atoi(argv[2]);

    // Стартовые настройки

    // {

    // }

    startFunc = std::chrono::system\_clock::now();

    // функция обработки

    // {

    // }

    endFunc = std::chrono::system\_clock::now();

    std::cout << "Время работы функции: " << std::chrono::duration<double>(endFunc - startFunc).count()

    << "\nВремя работы программы: " << std::chrono::duration<double>(std::chrono::system\_clock::now() - startProgram).count()

    << "\nВремя одного преобразования в микросекундах: " << std::chrono::duration<double>(endFunc - startFunc).count() \* 1000000 / COUNT\_TEST << std::endl;

    if (getrusage(RUSAGE\_SELF, &dataStatus) == -1)

    {

        std::cout << "Статусные данные не получены" << std::endl;

    }

    else

    {

        std::cout << "Max RAM: " << dataStatus.ru\_maxrss << " KiB\n"

        << "Разделяемая память: " << dataStatus.ru\_ixrss << "\n"

        << "Неразделяемая память: " << dataStatus.ru\_idrss << "\n"

        << "Неразделяемый стек: " << dataStatus.ru\_isrss << std::endl;

    }

    return 0;

}

## **Модифицированная основная программа с моделированием шума.**

/\*

Обязательные флаги при компиляции: -lfftw3 -lm

\*/

#include<stdio.h>

#include<stdlib.h>

#include<ctime>

#include<cstring>

#include<iostream>

#include<math.h>

#include <bitset>

#include<fftw3.h>

#include <random>

constexpr double PI = 3.1415;

constexpr int DATA\_LEN = 14191;

int COUNT\_CADR = 8;     // Количество кадров

constexpr int COUNT\_TESTS = 1;

double EbN0\_dB = 5;

double Eb = 1;

#define RECOWER // Раскомментировать если нужно провести обратный процесс

unsigned long timeStart;

int i;

int iGl = 0;

double normalizeCoef;   // Нормализующий коэффициент

int N = 16; // Число комплексных чисел, которые одновременно обрабатывает одна операция ОБПФ

                     // В будущем сюда добавятся комплексные числа для поддержания точности.

int maxLen;

void Data\_produce(char\*, int\*);

void Symbol\_produce(unsigned char\*, char\*);

void Modulation(unsigned char\*, fftw\_complex\*);

void Modulation\_16PSK(unsigned char \*, double\*\*);

void Demodulation(unsigned char \* , double\*\*);

void Decoding\_produse(unsigned char\*, char\*);

void OFFTW\_func(fftw\_plan, fftw\_complex\*);

void Gaussnoise(double, fftw\_complex\*);

double Erdproduce(unsigned char \* cadrs, unsigned char \* cadrs\_out);

constexpr int COUNT\_EB = 100000;

int main()

{

    double all\_errors;

    int countCurcle;

    std::srand(std::time(NULL));

            // std::cout << "Число гармоник: " << countGarm << std::endl;

        // Выделение памяти для входных и выходных данных

        N = 8;

        normalizeCoef = 1/sqrt(N);

        fftw\_complex \*offtDataIn = (fftw\_complex\*) fftw\_malloc(sizeof(fftw\_complex) \* N);

        fftw\_complex \*offtDataOut = (fftw\_complex\*) fftw\_malloc(sizeof(fftw\_complex) \* N);

        memset(offtDataIn, 0, sizeof(fftw\_complex) \* N);

        memset(offtDataOut, 0, sizeof(fftw\_complex) \* N);

        /\*

            Тип данных fftw\_complex представляет из себя массив двух чисел.

            Нулевой элемент массива содержит действительную часть комплексного числа;

            Первый элемент массива содержит мнимую часть комплексного числа.

        \*/

        #ifdef RECOWER

        fftw\_complex \*fftDataIn = (fftw\_complex\*) fftw\_malloc(sizeof(fftw\_complex) \* N);

        fftw\_complex \*fftDataOut = (fftw\_complex\*) fftw\_malloc(sizeof(fftw\_complex) \* N);

        memset(fftDataIn, 0, sizeof(fftw\_complex) \* N);

        memset(fftDataOut, 0, sizeof(fftw\_complex) \* N);

        #endif

        // std::cout << "Подготовка планов..." << std::endl;

        // Для передатчика

        fftw\_plan planBack;

        // Заранее создаем план для ОБПФ.

        planBack = fftw\_plan\_dft\_1d(N, offtDataIn, offtDataOut, FFTW\_BACKWARD, FFTW\_EXHAUSTIVE | FFTW\_DESTROY\_INPUT);

        #ifdef RECOWER

        // Для приемника и проверки

        fftw\_plan planForward;

        planForward = fftw\_plan\_dft\_1d(N, fftDataIn, fftDataOut, FFTW\_FORWARD, FFTW\_EXHAUSTIVE);

        #endif

        // Создать план БПФ без явного указания адреса входных и выходных данных на данной версии библиотеки нельзя.

        // Но можно создать фиктивные входные массивы, засунуть их в создание плана и при выполнении (О)БПФ явно указывать адреса нужных массивов

        // НОx2 типы массивов и размер массивов должны быть одинаковыми!!!!!!!!!!!!!!

        /\*

        Справка по флагам:

        fftw\_plan\_dft\_1d(N, in, out, flag1, flag2)

        flag1:

            FFTW\_FORWARD: прямое преобразование Фурье (БПФ).

            FFTW\_BACKWARD: обратное преобразование Фурье (ОБПФ).

        flag2:

            Построение плана:

                FFTW\_ESTIMATE: Быстрое создание плана, не сильно оптимальное выполнение (О)БПФ

                FFTW\_MEASURE: Ищет более оптимальный алгоритм.

                Создание плана займет время, выполнение (О)БПФ будет быстрее

                FFTW\_PATIENT: Ищет еще более оптимальный алгоритм.

                Создание будет еще дольше, но будет еще более высокая производительность (О)БПФ.

                FFTW\_EXHAUSTIVE: Перебор всех имеющихся алгоритмов и выбор наилучшего.

                Самый затратный по времени создания плана и самый эффективный по выполнению (О)БПФ.

            Управления памятью:

                FFTW\_IN\_PLACE: Входные и выходные данные находятся в одном и том же массиве.

                Экономит память, но изменяет исходные данные.

                FFTW\_PRESERVE\_INPUT: Гарантирует, что входные данные не будут изменены в процессе преобразования.

                Может немного замедлить выполнение.

                FFTW\_DESTROY\_INPUT: Разрешает FFTW изменять входные данные для оптимизации преобразования.

                Может ускорить выполнение, но входные данные будут потеряны.

            Доп флаги:

                FFTW\_UNALIGNED: Входные и/или выходные данные могут быть не выровнены в памяти.

                Использовать, если используется malloc() вместо fftw\_malloc() для выделения памяти.

                FFTW\_CONSERVE\_MEMORY: Пытается использовать меньше памяти, но может немного замедлить выполнение.

                FFTW\_TRANSPOSED\_ORDER: Указывает, что выходные данные должны быть транспонированы.

        \*/

        // std::cout << "Подготовка плана завершена." << std::endl;

    for (COUNT\_CADR = 1; COUNT\_CADR < 9; ++COUNT\_CADR)

    {

        std::cout << "Зашифрованных чисел: " << COUNT\_CADR << std::endl;

        unsigned char cadrs[COUNT\_CADR];

        char buff[DATA\_LEN];

        buff[0]=0b00000001;

        buff[1]=0b00100011;

        buff[2]=0b01000101;

        buff[3]=0b01100111;

        buff[4]=0b10001001;

        buff[5]=0b10101011;

        buff[6]=0b11001101;

        buff[7]=0b11101111;

        buff[8]='\0';

        maxLen = strlen(buff);

        // memset(buff, 0, sizeof(char)\*DATA\_LEN);

        #ifdef RECOWER

        char buff\_out[COUNT\_CADR];

        unsigned char cadrs\_out[COUNT\_CADR];

        double\*\* symbolCodingOut = static\_cast<double\*\*>(malloc(2 \* sizeof(double\*)));

        for (int i = 0; i < 2; ++i) {

            symbolCodingOut[i] = static\_cast<double\*>(malloc(COUNT\_CADR \* sizeof(double)));

        }

        #endif

        // printf("Введите строку\n");

        // Data\_produce(buff, &maxLen);

        // printf("Введенная строка: %s\nЧисло символов в строке: %d\n", buff, maxLen);

        // timeStart = time(NULL);

        int lss = 1;

        // for (lss = 0; lss < COUNT\_TESTS; ++lss)

        for (EbN0\_dB = 21; EbN0\_dB > 0; EbN0\_dB -= 5)

        {

            all\_errors = 0;

        for (int count\_tt = 0; count\_tt < COUNT\_EB; ++count\_tt)

        {

            // std::cout << "Отношение сигнал-шум на бит в децибелах: " << EbN0\_dB << std::endl;

            iGl = 0;

        {

            // printf("Тест %d\n", lss);

            // timeStart = time(NULL);

            // Выравнивание времени:

            // while (timeStart == time(NULL));

            // timeStart = time(NULL);

            // while(timeStart == time(NULL))

            // while(getchar() != '\n');

            // for(unsigned long long kk = 0; kk < 20000; ++kk, iGl = 0)

            countCurcle = 0;

            while (iGl < maxLen)

            {

                ++countCurcle;

                // printf("\nКадр: %ld\n", iGl/6+1);

                // printf("Строка обработки:\n");

                // for (i = (iGl/6) \* 6; i < (iGl/6 + 1) \* 6; i++)

                //  printf("%c", buff[i]);

                // printf("\n");

                // printf("Данные, обработанные на этом этапе:\n");

                // for (i = (iGl/6) \* 6; i < (iGl/6 + 1) \* 6; i++)

                //  printf("%8b\n", buff[i]);

                Symbol\_produce(cadrs, buff);

                // for (i = 0; i < COUNT\_CADR; i++)

                //  std::cout << i << " "<< std::bitset<4>(cadrs[i]) << std::endl;

                Modulation(cadrs, offtDataIn);

                // Здесь идет работа ОБПФ и дальнейшая передача.

                // std::cout << "Start data:\n";

                // for (i = 0; i < N; ++i)

                //  std::cout << "Real = " << offtDataIn[i][0] << " Mnim = " << offtDataIn[i][1] << std::endl;

                OFFTW\_func(planBack, offtDataOut);

                // std::cout << "Rezult OFFT: \n ";

                // for (i = 0; i < N; ++i)

                //  std::cout << "Real = " << offtDataOut[i][0] << " Mnim = " << offtDataOut[i][1] << std::endl;

                // Сюда вставить моделирование шума

                Gaussnoise(EbN0\_dB, offtDataOut);

                #ifdef RECOWER

                // Обратный процесс:

                for (i = 0; i < N; ++i)

                {

                    fftDataIn[i][0] = offtDataOut[i][0];

                    fftDataIn[i][1] = offtDataOut[i][1];

                }

                // БПФ

                fftw\_execute(planForward);

                // Нормализация

                for (i = 0; i < N; ++i)

                {

                    fftDataOut[i][0] = fftDataOut[i][0] \* normalizeCoef;

                    fftDataOut[i][1] = fftDataOut[i][1] \* normalizeCoef;

                }

                for (i = 0; i < COUNT\_CADR; ++i)

                {

                    symbolCodingOut[0][i] = fftDataOut[i][0];

                    symbolCodingOut[1][i] = fftDataOut[i][1];

                }

                // std::cout << "\n\nFFT:" << std::endl;

                // for (i = 0; i < N; ++i)

                //  std::cout << "Real = " << fftDataOut[i][0] << " Mnim = " << fftDataOut[i][1] << std::endl;

                Demodulation(cadrs\_out, symbolCodingOut);

                // printf("Демодулированные данные:\n");

                // for (i = 0; i < COUNT\_CADR; ++i)

                //  std::cout << i << " "<< std::bitset<4>(cadrs\_out[i]) << std::endl;

                // printf("\n\nДекодирование:\n");

                Decoding\_produse(cadrs\_out, buff\_out);

                // printf("Декодированные данные:\n");

                // for (int i = 0; i < COUNT\_CADR/2; i++)

                //  printf("%c\n", buff\_out[i]);

                all\_errors += Erdproduce(cadrs, cadrs\_out);

                // if (Erdproduce(cadrs, cadrs\_out) > 0.99)

                //  std::cout << Erdproduce(cadrs, cadrs\_out) << std::endl;

                #endif

            }

            all\_errors/=countCurcle;

        }

            }

        std::cout << EbN0\_dB << ' ' << all\_errors/(COUNT\_EB) << std::endl;

        }

        for (int i = 0; i < 2; ++i) {

            free (symbolCodingOut[i]);

        }

        free(symbolCodingOut);

    }

        /\*

        Gaussnoise(EbN0\_dB);

        Demodulation();

        Erd = Erdproduce();

        Mrd = Mrdproduce();

        printf("Eb/N0 %d dB\t Symbol\_Error: %f \t Bit\_Error:%f \n", i, Mrd, Erd);

        \*/

        // Очистка данных для FFT

        fftw\_destroy\_plan(planBack);

        fftw\_free(offtDataIn);

        fftw\_free(offtDataOut);

        #ifdef RECOWER

        // for (int i = 0; i < 2; ++i) {

        //  free (symbolCodingOut[i]);

        // }

        // free(symbolCodingOut);

        fftw\_destroy\_plan(planForward);

        fftw\_free(fftDataIn);

        fftw\_free(fftDataOut);

        #endif

        fftw\_cleanup();

    // }

    return 0;

}

inline void OFFTW\_func(fftw\_plan planBack, fftw\_complex\* offtDataOut)

{

    // ОБПФ

    fftw\_execute(planBack);

    // Нормализация. В будущем заменить в зависимости от параметров приемника-передатчика

    for (i = 0; i < N; ++i)

    {

        offtDataOut[i][0] \*= normalizeCoef;

        offtDataOut[i][1] \*= normalizeCoef;

    }

}

// Тут просто будет ввод текста и ожидание подтверждения отправки.

void Data\_produce(char\* buff, int\* maxLen)

{

    char c;

    int i = -1;

    while((buff[++i] = getchar()) !='\n');

    buff[i] = 0;

    \*maxLen = i;

}

// Преобразование символов в кадры по 4 бита. (Проверено)

inline void Symbol\_produce(unsigned char\* cadrs, char\* buff)

{

    for (i = 0; i < COUNT\_CADR; ++i, ++iGl)

    {

        cadrs[i] = buff[iGl] & 0b1111;

        cadrs[++i] = buff[iGl] >> 4 & 0b1111;

    }

}

// Преобразование кадра в символы. (Проверено)

void Decoding\_produse(unsigned char\* cadrs\_out, char\* buff\_out)

{

    int i, ss;

    for (i = 0, ss = 0; i < COUNT\_CADR; i+=2, ++ss)

    {

        buff\_out[ss] = cadrs\_out[i] | (cadrs\_out[i+1] << 4);

    }

}

// Модуляция. Реализованный 16-QAM алгоритм.

// Возможно требуется нормализация I и Q

// Попробовать преобразовать if-else во что-то более стабильное, либо в switch, либо в выражение

/\* Таблица соответствия

Биты  | I  | Q  | Комплексное число (I + jQ)

------|----|----|---------------------------

0000  | -3 |  3 | -3 + 3j

0001  | -1 |  3 | -1 + 3j

0010  | -3 |  1 | -3 + 1j

0011  | -1 |  1 | -1 + 1j

0100  | -3 | -3 | -3 - 3j

0101  | -1 | -3 | -1 - 3j

0110  | -3 | -1 | -3 - 1j

0111  | -1 | -1 | -1 - 1j

1000  |  3 |  3 |  3 + 3j

1001  |  1 |  3 |  1 + 3j

1010  |  3 |  1 |  3 + 1j

1011  |  1 |  1 |  1 + 1j

1100  |  3 | -3 |  3 - 3j

1101  |  1 | -3 |  1 - 3j

1110  |  3 | -1 |  3 - 1j

1111  |  1 | -1 |  1 - 1j

 \*/

inline void Modulation(unsigned char \* cadrs, fftw\_complex \*symbolCoding)

{

    // Вариант 1: swithc-case.

    for (i = 0; i < COUNT\_CADR; ++i)

    {

        switch (cadrs[i]) {

            case 0b0000: symbolCoding[i][0] = -3.0; symbolCoding[i][1] =  3.0; break; // 0

            case 0b0001: symbolCoding[i][0] = -1.0; symbolCoding[i][1] =  3.0; break; // 1

            case 0b0010: symbolCoding[i][0] = -3.0; symbolCoding[i][1] =  1.0; break; // 2

            case 0b0011: symbolCoding[i][0] = -1.0; symbolCoding[i][1] =  1.0; break; // 3

            case 0b0100: symbolCoding[i][0] = -3.0; symbolCoding[i][1] = -3.0; break; // 4

            case 0b0101: symbolCoding[i][0] = -1.0; symbolCoding[i][1] = -3.0; break; // 5

            case 0b0110: symbolCoding[i][0] = -3.0; symbolCoding[i][1] = -1.0; break; // 6

            case 0b0111: symbolCoding[i][0] = -1.0; symbolCoding[i][1] = -1.0; break; // 7

            case 0b1000: symbolCoding[i][0] =  3.0; symbolCoding[i][1] =  3.0; break; // 8

            case 0b1001: symbolCoding[i][0] =  1.0; symbolCoding[i][1] =  3.0; break; // 9

            case 0b1010: symbolCoding[i][0] =  3.0; symbolCoding[i][1] =  1.0; break; // 10

            case 0b1011: symbolCoding[i][0] =  1.0; symbolCoding[i][1] =  1.0; break; // 11

            case 0b1100: symbolCoding[i][0] =  3.0; symbolCoding[i][1] = -3.0; break; // 12

            case 0b1101: symbolCoding[i][0] =  1.0; symbolCoding[i][1] = -3.0; break; // 13

            case 0b1110: symbolCoding[i][0] =  3.0; symbolCoding[i][1] = -1.0; break; // 14

            case 0b1111: symbolCoding[i][0] =  1.0; symbolCoding[i][1] = -1.0; break; // 15

            default:   symbolCoding[i][0] = 0.0; symbolCoding[i][1] = 0.0; // Обработка неожиданной ситуации (опционально)

        }

    }

    /\*Вариант 2: Использование выражения\*/

    /\*

    for (i = 0; i < COUNT\_CADR; i++)

    {

        symbolCoding[0][i] = (cadrs[i] >> 3) \* 2 - 1;

        symbolCoding[1][i] = ((cadrs[i] & 0b0010) >> 1) \* 2 - 1;

        if ((cadrs[i] & 0b0100)) symbolCoding[0][i] = symbolCoding[0][i] \* 3;

        if ((cadrs[i] & 0b0001)) symbolCoding[1][i] = symbolCoding[1][i] \* 3;

    }

    \*/

}

/\* Таблица соответствия для 16-PSK

Биты  | I  | Q

------|----|----

0000  | -3 |  3

0001  | -3 |  2.25

0010  | -3 |  1.5

0011  | -3 |  0.75

0100  | -3 | -3

0101  | -3 | -2.25

0110  | -3 | -1.5

0111  | -3 | -0.75

1000  |  3 |  3

1001  |  3 |  2.25

1010  |  3 |  1.5

1011  |  3 |  0.75

1100  |  3 | -3

1101  |  3 | -2.25

1110  |  3 | -1.5

1111  |  3 | -0.75

\*/

void Modulation\_16PSK(unsigned char \* cadrs, double\*\* symbolCoding)

{

    // Вариант 1: swithc-case.

    for (i = 0; i < COUNT\_CADR; ++i)

    {

        switch (cadrs[i]) {

            case 0b0000: symbolCoding[0][i] = -3.0; symbolCoding[1][i] =  3.0; break; // 0

            case 0b0001: symbolCoding[0][i] = -3.0; symbolCoding[1][i] =  2.25; break; // 1

            case 0b0010: symbolCoding[0][i] = -3.0; symbolCoding[1][i] =  1.5; break; // 2

            case 0b0011: symbolCoding[0][i] = -3.0; symbolCoding[1][i] =  0.75; break; // 3

            case 0b0100: symbolCoding[0][i] = -3.0; symbolCoding[1][i] = -3.0; break; // 4

            case 0b0101: symbolCoding[0][i] = -3.0; symbolCoding[1][i] = -2.25; break; // 5

            case 0b0110: symbolCoding[0][i] = -3.0; symbolCoding[1][i] = -1.5; break; // 6

            case 0b0111: symbolCoding[0][i] = -3.0; symbolCoding[1][i] = -0.75; break; // 7

            case 0b1000: symbolCoding[0][i] =  3.0; symbolCoding[1][i] =  3.0; break; // 8

            case 0b1001: symbolCoding[0][i] =  3.0; symbolCoding[1][i] =  2.25; break; // 9

            case 0b1010: symbolCoding[0][i] =  3.0; symbolCoding[1][i] =  1.5; break; // 10

            case 0b1011: symbolCoding[0][i] =  3.0; symbolCoding[1][i] =  0.75; break; // 11

            case 0b1100: symbolCoding[0][i] =  3.0; symbolCoding[1][i] = -3.0; break; // 12

            case 0b1101: symbolCoding[0][i] =  3.0; symbolCoding[1][i] = -2.25; break; // 13

            case 0b1110: symbolCoding[0][i] =  3.0; symbolCoding[1][i] = -1.5; break; // 14

            case 0b1111: symbolCoding[0][i] =  3.0; symbolCoding[1][i] = -0.75; break; // 15

            default:   symbolCoding[0][i] = 0.0; symbolCoding[1][i] = 0.0; // Обработка неожиданной ситуации (опционально)

        }

    }

}

void Demodulation\_16PSK(unsigned char \* cadrs\_out, double\*\* symbolCoding)

{

    for (int i = 0; i < COUNT\_CADR; ++i)

    {

        if (symbolCoding[0][i] < 0)

        {

            cadrs\_out[i] = 0b0000;

        }

        else

        {

            cadrs\_out[i] = 0b1000;

        }

        if (symbolCoding[1][i] < -2.625)

        {

            cadrs\_out[i] |= 0b0100;

        }

        else if (symbolCoding[1][i] > 2.625)

        {

            cadrs\_out[i] |= 0b0000;

        }

        else if (symbolCoding[1][i] > 1.875)

        {

            cadrs\_out[i] |= 0b0001;

        }

        else if (symbolCoding[1][i] > 1.125)

        {

            cadrs\_out[i] |= 0b0010;

        }

        else if (symbolCoding[1][i] > 0)

        {

            cadrs\_out[i] |= 0b0010;

        }

        else if (symbolCoding[1][i] < -1.875)

        {

            cadrs\_out[i] |= 0b0101;

        }

        else if (symbolCoding[1][i] < -1.125)

        {

            cadrs\_out[i] |= 0b0110;

        }

        else

        {

            cadrs\_out[i] |= 0b0111;

        }

    }

}

void Demodulation(unsigned char \* cadrs\_out, double\*\* symbolCoding)

{

    for (int i = 0; i < COUNT\_CADR; ++i)

    {

        if (symbolCoding[0][i] < -2.0)

        {

            cadrs\_out[i] = 0b0000;

        }

        else if (symbolCoding[0][i] > 2.0)

        {

            cadrs\_out[i] = 0b1000;

        }

        else if (symbolCoding[0][i] < 2.0 && symbolCoding[0][i] > 0.0)

        {

            cadrs\_out[i] = 0b1001;

        }

        else

        {

            cadrs\_out[i] = 0b0001;

        }

        if (symbolCoding[1][i] < -2.0)

        {

            cadrs\_out[i] |= 0b0100;

        }

        else if (symbolCoding[1][i] > 2.0)

        {

            cadrs\_out[i] |= 0b0000;

        }

        else if (symbolCoding[1][i] < 2.0 && symbolCoding[1][i] > 0.0)

        {

            cadrs\_out[i] |= 0b0010;

        }

        else

        {

            cadrs\_out[i] |= 0b0110;

        }

    }

}

double Gauss()

{

    double U, V, Z;

    U = (double)(std::rand() + 1.0) / (double)(RAND\_MAX + 1.0);

    V = (double)(std::rand() + 1.0) / (double)(RAND\_MAX + 1.0);

    Z = sqrt(-2 \* log(U)) \* sin(2 \* PI \* V);

    return Z;

}

void Gaussnoise(double EbN0\_dB, fftw\_complex\* symbolCoding)

{

    double EbN0 = pow(10, EbN0\_dB / 10);

    double N0 = Eb / EbN0;

    double sigma = sqrt(N0 / 2);

    for (int i = 0; i < N; i++)

    {

        symbolCoding[i][0] = symbolCoding[i][0] + Gauss() \* sigma;

        symbolCoding[i][1] = symbolCoding[i][1] + Gauss() \* sigma;

    }

}

double Erdproduce(unsigned char \* cadrs, unsigned char \* cadrs\_out)

{

    double Erd;

    int Nrd = 0;

    for (i = 0; i < COUNT\_CADR; ++i)

    {

        if (cadrs[i] != cadrs\_out[i])

            Nrd++;

    }

    Erd = (double)Nrd / (double)COUNT\_CADR;

    return Erd;

}

// double Mrdproduce()

// {

//  double Mrd;

//  int Srd = 0;

//  int p[M];

//  int pt;

//  for (int i = 0; i < symbol\_len; i++)

//  {

//      int pt = 0;

//      for (int j = 0; j < M; j++)

//      {

//          if (symbol\_decoding[j][i] == symbol\_bfcoding[j][i])

//              p[j] = 0;

//          else

//              p[j] = 1;

//      }

//      for (int j = 0; j < M; j++)

//      {

//          pt = pt + p[j];

//      }

//      if (pt > 0)

//          Srd++;

//  }

//  Mrd = (double)Srd / (double)(symbol\_len);

//  return Mrd;

// }