|  |  |
| --- | --- |
| ФАКУЛЬТЕТ | «Робототехника и комплексная автоматизация» |
| КАФЕДРА | «Системы автоматизированного проектирования (РК-6)» |

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ***

***НА ТЕМУ:***

|  |
| --- |
| «Цифровой модуль приемо-передачи на основе OFDM- модуляции» |
|  |
|  |
|  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент | РК6-84Б |  |  |  | Д.С. Николайчук |
|  | (группа) |  | (подпись) |  | (инициалы, фамилия) |
| Руководитель ВКР | | |  |  | Д.И. Оглоблин |
|  | | | (подпись) |  | (инициалы, фамилия) |
| Нормоконтролер | | |  |  | С.В. Грошев |
|  | | | (подпись) |  | (инициалы, фамилия) |
|  | | |  |  |  |
|  | | |  |  |  |
|  | | |  |  |  |
|  | | |  |  |  |

2025 год

РЕФЕРАТ

Данная работа посвящена разбору алгоритма OFDM, его оптимизации и реализации на микроконтроллере с полным циклом работы системы передачи данных.

В ходе проделанной работы было выполнено:

* Разбор основных блоков алгоритма OFDM.
* Анализ готовых реализаций алгоритмов.
* Оптимизация алгоритма на языке Си.
* Реализация алгоритма для микроконтроллера STM32 с последующим тестированием времени работы алгоритма.
* Разработка и реализация полного цикла обработки данных для последующей передачи для микроконтроллера STM32.

В результате проделанной работы была разработана и реализована многоскоростная многопоточная система для корректной обработки и синхронной передачи данных.

Работа содержит 52 страниц, 4 раздела, 30 иллюстраций, 1 таблицу, 3 листинга. Выполнена с использованием 11 источников.

СОДЕРЖАНИЕ

[РЕФЕРАТ 2](#_Toc201070623)

[СОДЕРЖАНИЕ 3](#_Toc201070624)

[ОПРЕДЕЛЕНИЯ 5](#_Toc201070625)

[ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 6](#_Toc201070626)

[ВВЕДЕНИЕ 7](#_Toc201070627)

[ПОСТАНОВКА ЗАДАЧИ 11](#_Toc201070628)

[1. Анализ основных блоков алгоритма 12](#_Toc201070629)

[**1.1. S2P 12**](#_Toc201070630)

[**1.2. Mapper 13**](#_Toc201070631)

[**1.3. IFFT 16**](#_Toc201070632)

[**1.4. P2S 28**](#_Toc201070633)

[2. Оптимизация алгоритма 29](#_Toc201070634)

[3. Реализация программы на микроконтроллере 31](#_Toc201070635)

[**3.1. Синхронизация 31**](#_Toc201070636)

[**3.2. Приоритетность 38**](#_Toc201070637)

[4. Тестирование алгоритма на микроконтроллерах 42](#_Toc201070638)

[ЗАКЛЮЧЕНИЕ 46](#_Toc201070639)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 47](#_Toc201070640)

[ПРИЛОЖЕНИЕ А 49](#_Toc201070641)

[ПРИЛОЖЕНИЕ Б 53](#_Toc201070642)

[ПРИЛОЖЕНИЕ В 57](#_Toc201070643)

[ПРИЛОЖЕНИЕ Г 69](#_Toc201070644)

# **ОПРЕДЕЛЕНИЯ**

Поднесущая – боковая полоса несущей волны радиочастоты, которая модулируется для передачи дополнительной информации

Ряд Фурье – способ представления произвольной сложной функции суммой более простых – синуса и косинуса

Дискретное преобразование Фурье – математическое преобразование, разлагающее последовательность значений сигнала на компоненты различных частот

Обратное дискретное преобразование Фурье – математическое преобразование, восстанавливающее временной сигнал из его частотных компонент

Гармоника – комплексная экспоненциальная функция определенной частоты, амплитуды и фазы, являющаяся базовым элементом для синтеза временного сигнала посредством обратного дискретного преобразования Фурье

FFTW3 - библиотека, предоставляющая эффективные алгоритмы вычисления прямого и обратного дискретного преобразования Фурье

Свертка – комбинирование двух сигналов

# **ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ**

OFDM (англ. Orthogonal frequency-division multiplexing) – мультиплексирование с ортогональным частотным разделением каналов

СВЧ – сверхвысокие частоты

STM32 – семейство 32-битных микроконтроллеров на базе архитектуры ARM Cortex-M, разработанной компанией STMicroelectronics.

ЦАП – цифро-аналоговый преобразователь

16-QAM (англ. Quadrature Amplitude Modulation) – 16-позиционная квадратурная амплитудная модуляция

ДПФ – дискретное преобразование Фурье

ОДПФ – обратное дискретное преобразование Фурье

ОПФ – обратное преобразование Фурье

БПФ – быстрое преобразование Фурье

FFTW3 (англ. Fastest Fourier Transform in the West, version 3) – самое быстрое преобразование Фурье на Западе, версия 3

RTOS (англ. Real-Time Operating System) – операционная система реального времени

UART (англ. Universal Asynchronous Receiver/Transmitter) - универсальный асинхронный приемопередатчик

# **ВВЕДЕНИЕ**

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. Полная схема передачи данных представлена на рисунке 1.

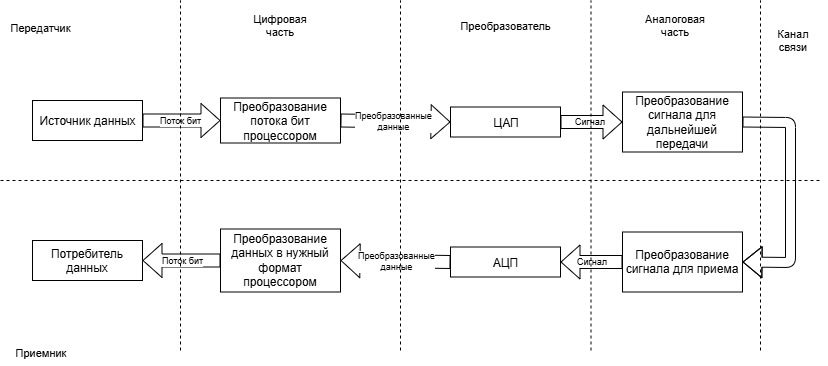


Рисунок 1 – Разделение устройства передачи информации на аналоговую и цифровую часть

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 – технология цифровой связи, в которой сигнал СВЧ образуется из большого количества близко расположенных низкочастотных ортогональных поднесущих – частот, которые занимают ограниченную полосу [1]. Спектры разных поднесущих перекрываются, но не мешают друг другу в следствии правильного выбора временного интервала, на котором все частоты кратны и ортогональны друг другу. OFDM сигнал представлен на рисунке2. Информация о битах записывается и передается в виде амплитуды и фазы прямоугольных импульсов.

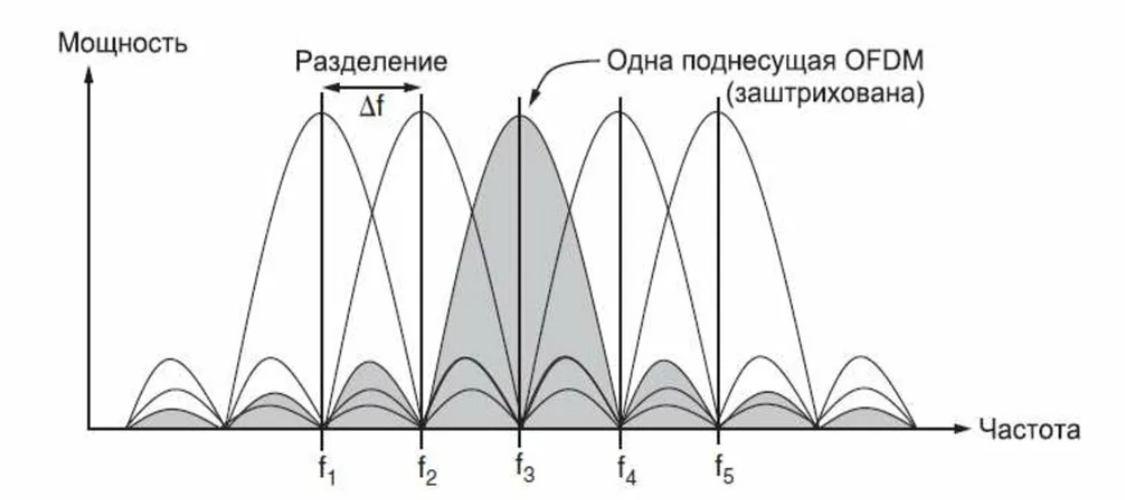
****

Рисунок 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 [2]. Сколько нужно таких волн? Ответить на данный вопрос можно, разложив прямоугольный импульс в ряд Фурье. Используя 8 частот, получится приближение, показанное на рисунке 3, оно напоминает начальный сигнал, но не в полной мере его отражает.

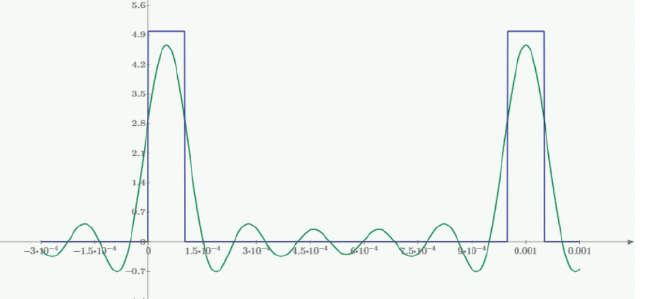
**

Рисунок 3 – Разложение прямоугольного сигнала в ряд Фурье с использованием 8 частот

Хорошее приближение к истинному виду сигнала достигается при использовании 64 частот или более, как представлено на рисунке 4. 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».

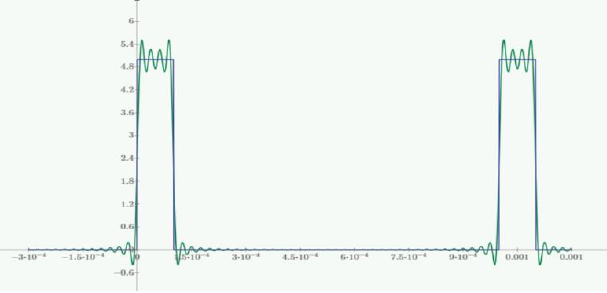
**

Рисунок 4 – Разложение прямоугольного сигнала в ряд Фурье с использованием 64 частот

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 [2]. В каждом подканале скорость обработки в раз меньше, где – число подканалов. Распараллеливание потока технически выполняет демультиплексор, и он описывается в виде алгоритма на языке Си.

В следующем блоке канального кодирования биты группируются в зависимости от отношения сигнал/шум и отображаются точками на комплексной плоскости. Пример таких соотношений представлен на рисунке 5. На нем представлена модуляция 16-QAM, применяемая в данной работе.

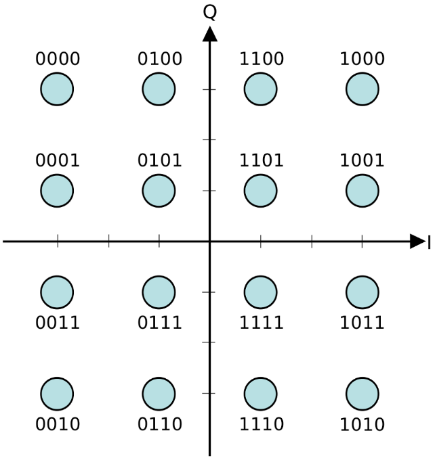


Рисунок 5 – Пример, соотношение 4 бит на комплексной плоскости

После этого комплексные числа, описывающие амплитуду и фазу каждой точки созвездия с помощью некоторого числа частот в полосе поднесущей при помощи обратного преобразования Фурье 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. Более подробно алгоритм описан в Приложении А.

**Задача** ВКРБ заключается в проведении анализа существующих алгоритмов OFDM, реализации и оптимизации алгоритма в одном подканале с использованием микроконтроллера STM32. Также необходимо добиться скорости вычислений в 70 мкс. 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**

Данный блок, представленный на рисунке 6, 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.

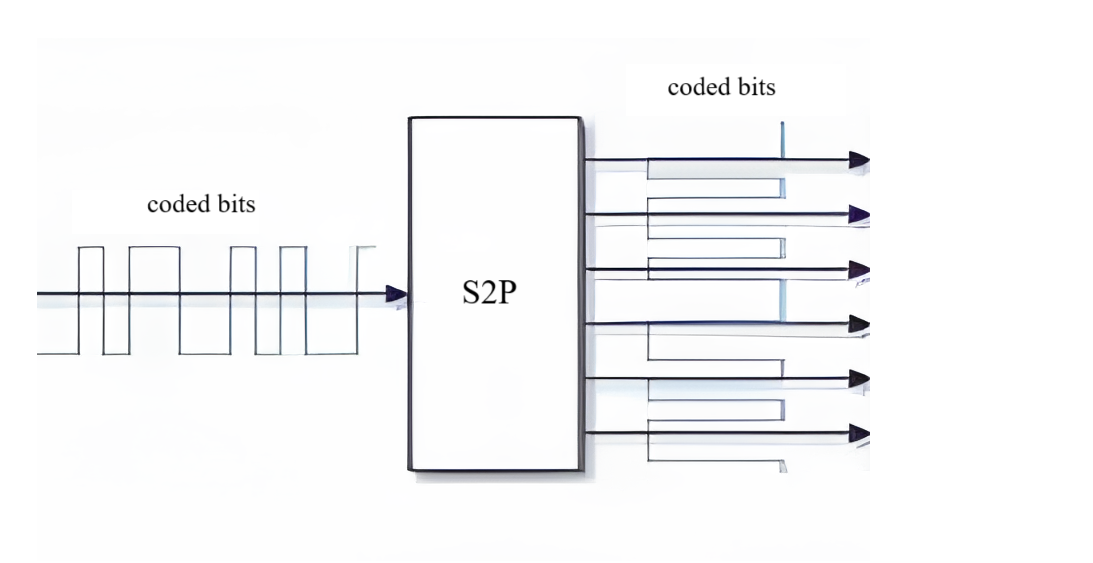


Рисунок 6 – Схема блока S2P

## **Mapper**

Блок Mapper, изображенный на рисунке 7, 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.

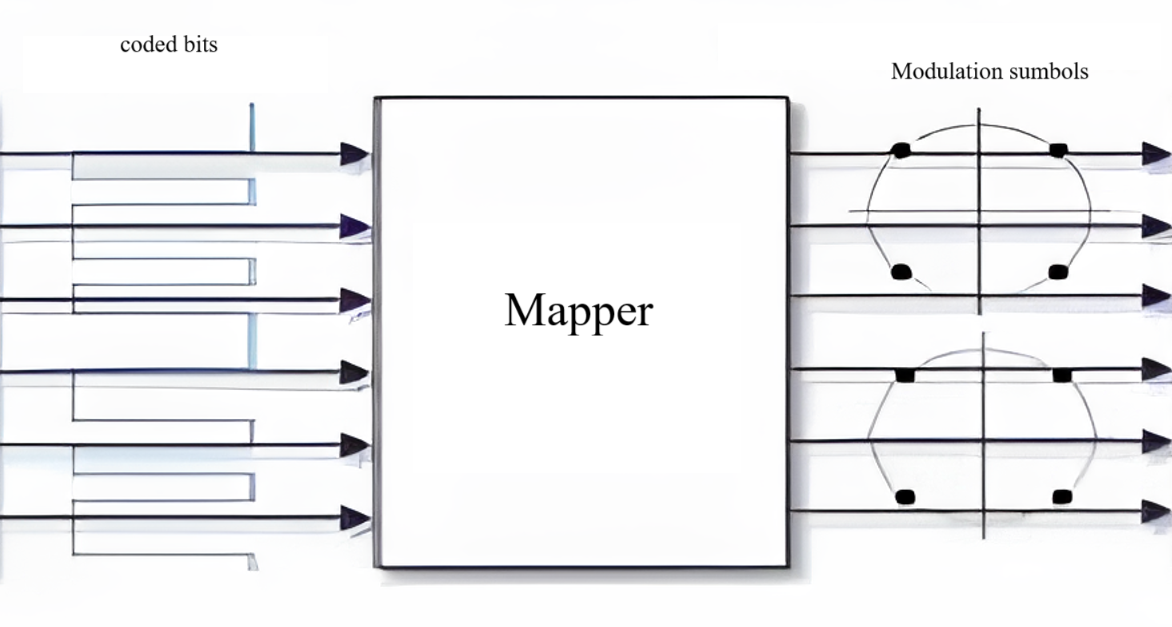
****

Рисунок 7 – Схема блока Mapper

Данный случай представлен на рисунке 8. Этот способ передачи данных неэффективен, так как если передавать данные в виде «1 – есть сигнал, 0 – нет сигнала», то это будет в разы проигрывать в скорости передачи информации методам кодирования (модуляции), которые реализуются в данном блоке. Остальные методы модуляции описаны в Приложении Б

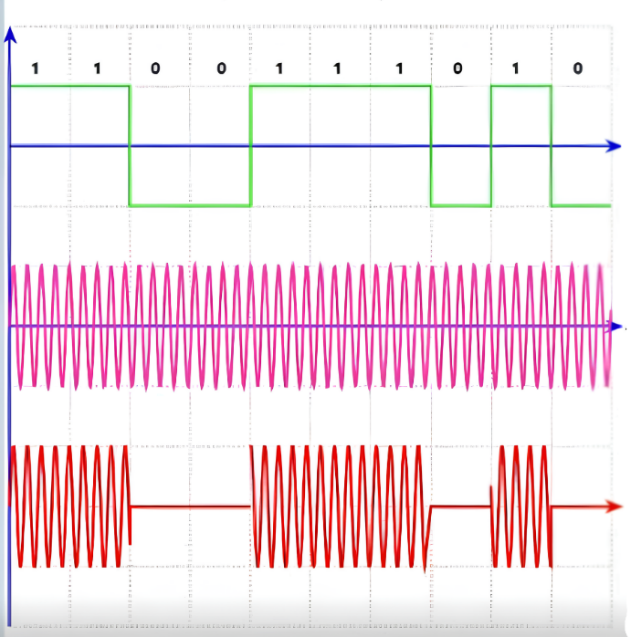
****

Рисунок 8 – Представление потока бит в виде «On-Off»

Для рассмотрения способов модуляции, рассмотрим комплексную плоскость. На ней имеется ось реальных значений и ось мнимых значений. 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. Каждый из методов имеет свои преимущества и недостатки, но самым эффективным является модуляция при изменении фазы и амплитуды.

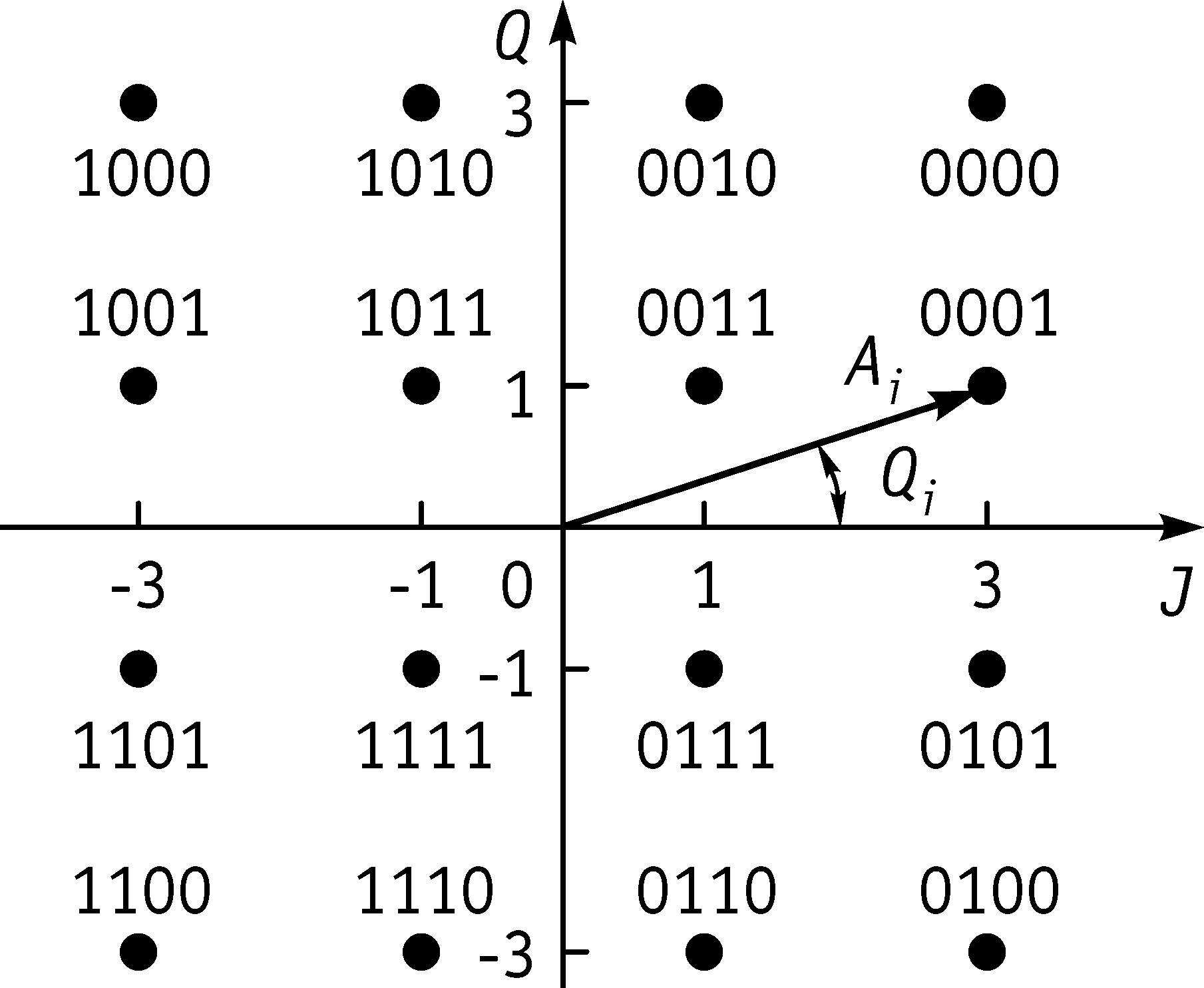


Рисунок 9 – Представление модуляции 16-QAM на комплексной плоскости.

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.

## **1.3 IFFT**

Схема данного блока изображена на рисунке 10. Он преобразовывает комплексное число или комплексные числа, полученные на прошлом этапе, в временной сигнал. Это позволяет сделать обратное преобразование Фурье. Рассмотрим преобразование Фурье [3].

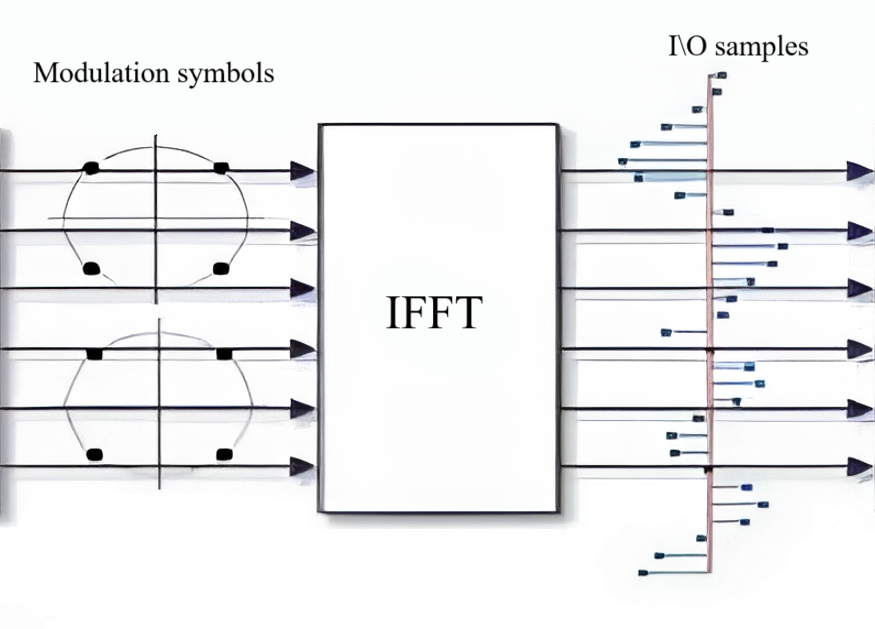
****

Рисунок 10 – Схема блока IFFT

Для дискретного сигнала  из  отсчетов (где ), дискретное преобразование Фурье (ДПФ), обозначаемое , определяется как:

для , где - -ый отсчет выходной последовательности (спектр сигнала в частотной области);

- длина последовательности;

- -ый отсчет входной последовательности (сигнала во временной области).

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 сигнал, представленный на рисунке 11, который необходимо передать. При преобразовании данного сигнала с помощью обратного преобразования Фурье (ОПФ) мы получим сигнал, представленный на рисунке 12, по форме напоминающий исходный, но который возможно передать в цифро-аналоговый преобразователь для последующей передачи.

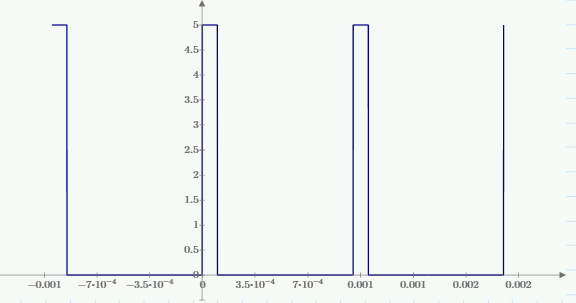


Рисунок 11 – Прямоугольный импульс, являющийся входным сигналом

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.

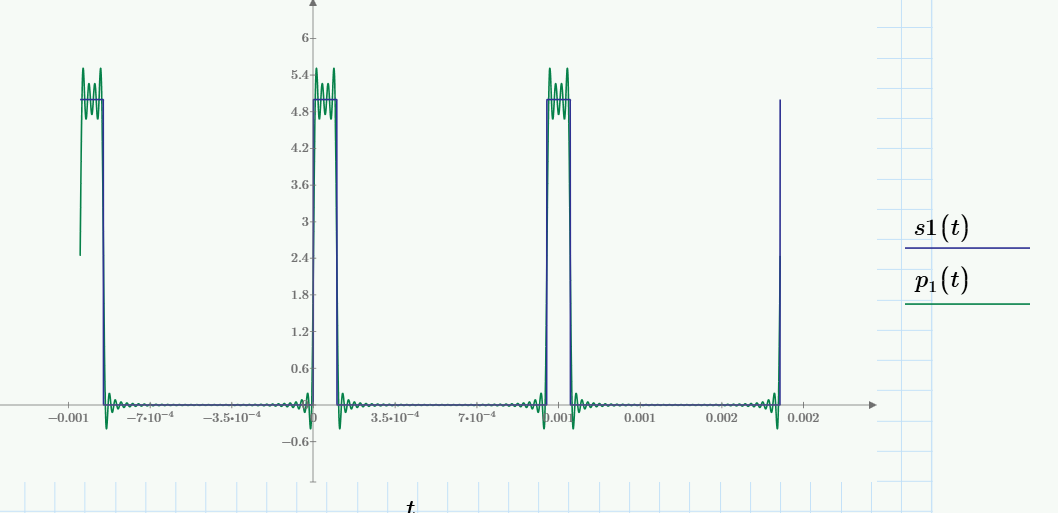


Рисунок 12 – Сигнал после ОПФ при использовании 40 гармоник

В качестве примера, рассмотрим преобразование того же входного сигнала, но используя одну гармонику. Получим вид, представленные на рисунке 13. По данному выходному сигналу очень сложно определить изначальный вид.

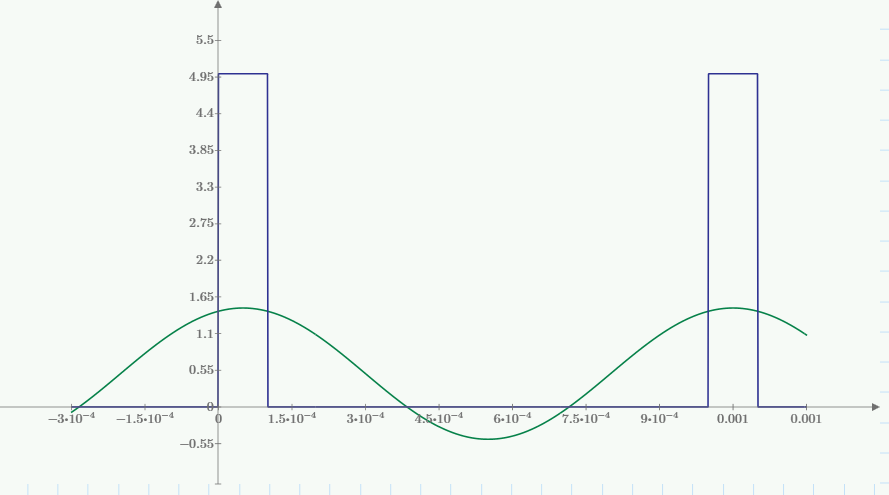


Рисунок 13 – Сигнал после ОПФ при использовании 1 гармоники

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.

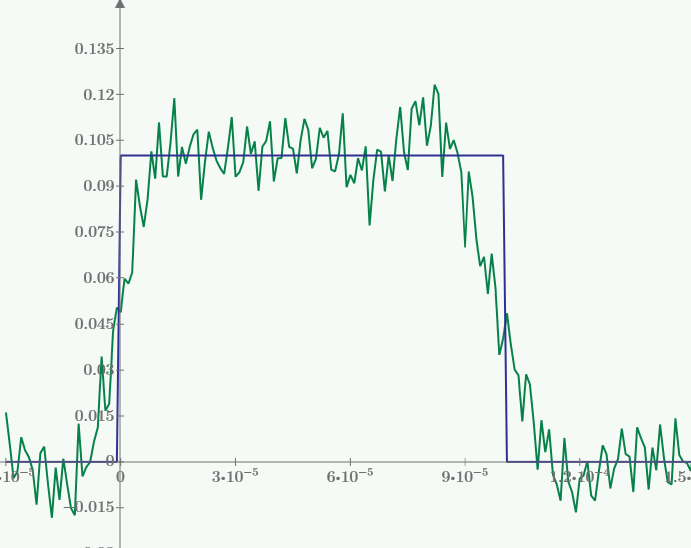


Рисунок 15 – Сигнал после ОПФ при использовании 4000 гармоник

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 [4]. Учитывая то, что с увеличением числа гармоник сигнал станет менее определяемый, требуется использовать ограниченное число гармоник, но такое, чтобы сигнал однозначно определялся.

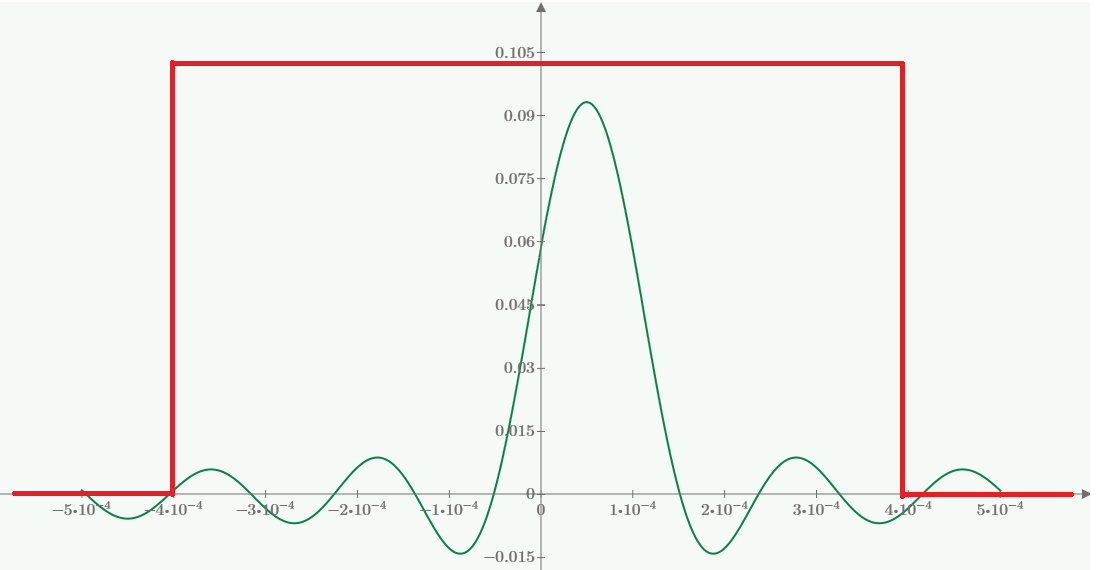


Рисунок 16 – Сигнал после ОПФ с использованием прямоугольного окна

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 на рисунке 17.

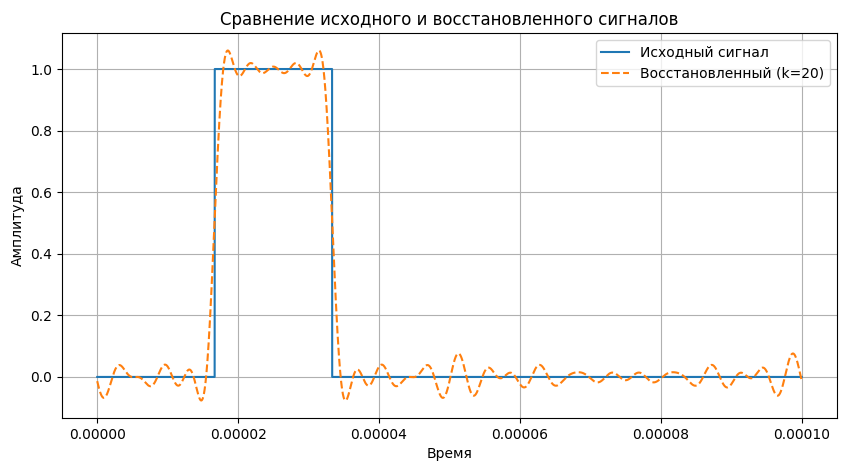


Рисунок 17 – Сравнение исходного и восстановленного сигнала с 20 гармониками

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. Итог выполнения неоптимизированного ОДПФ 10000 раз представлен на рисунке 18.



Рисунок 18 – Итог выполнения неоптимизированного ОДПФ 10000 раз

Итог выполнения оптимизированного ОДПФ представлен на рисунке 19. Из него можно увидеть, что данный алгоритм быстрее стандартных вычислений в 4-5 раз. Это связано как раз с тем, что вместо постоянного вычисления коэффициентов, алгоритм берет уже готовые из памяти.



Рисунок 19 – Итог выполнения оптимизированного ОДПФ 10000 раз

Перспективным направлением дальнейшей оптимизации вычислительной нагрузки является переход к использованию алгоритмов быстрого преобразования Фурье (БПФ). Несмотря на то, что реализация алгоритмов БПФ представляет собой более сложную задачу по сравнению со стандартными алгоритмами дискретного преобразования Фурье (ДПФ), их применение позволяет существенно повысить производительность системы за счет значительного снижения вычислительной сложности. В то время как стандартные алгоритмы ДПФ, даже при условии их оптимизации, характеризуются квадратичной вычислительной сложностью (O(n^2)), алгоритмы БПФ обладают логарифмической сложностью (O(n log n)). Это означает, что при увеличении размера входных данных вычислительная нагрузка в алгоритмах БПФ растет значительно медленнее, чем в алгоритмах ДПФ. Данный факт является критически важным для систем, работающих с большими объемами данных или требующих высокой скорости обработки сигнала, таких как системы связи и радиолокации. Переход к использованию алгоритмов БПФ позволит не только снизить вычислительную нагрузку на микроконтроллер, но и повысить общую эффективность системы, обеспечивая более быструю обработку сигнала и снижение энергопотребления. При этом, необходимо учитывать сложность реализации алгоритмов БПФ и тщательно оптимизировать их для конкретной платформы STM32, чтобы в полной мере реализовать их потенциал и избежать возможных проблем, связанных с ограниченными ресурсами микроконтроллера. Существует несколько различных алгоритмов БПФ, каждый из которых имеет свои особенности и преимущества:

* Алгоритм Кули-Туки (Cooley-Tukey): Один из самых известных и широко используемых алгоритмов БПФ. Он основан на принципе “разделяй и властвуй”, который позволяет эффективно разбивать задачу вычисления ДПФ на более мелкие подзадачи. Алгоритм Кули-Туки наиболее эффективен, когда размер входных данных является степенью двойки.
* Алгоритм БПФ с разделением по времени (Decimation-in-Time - DIT): Вариация алгоритма Кули-Туки, в которой разделение происходит по времени. Этот алгоритм рекурсивно разбивает входную последовательность на две подпоследовательности: с четными и нечетными индексами.
* Алгоритм БПФ с разделением по частоте (Decimation-in-Frequency - DIF): Альтернативная вариация алгоритма Кули-Туки, в которой разделение происходит по частоте. Этот алгоритм рекурсивно разбивает выходную последовательность на две подпоследовательности: с низкой и высокой частотой.
* Алгоритм Винограда (Winograd FFT): Алгоритм БПФ, который использует минимальное количество умножений, что может быть важным для систем с ограниченными вычислительными ресурсами. Однако, алгоритм Винограда является более сложным в реализации, чем алгоритм Кули-Туки.
* Алгоритм Герцеля (Goertzel algorithm): Алгоритм, который позволяет эффективно вычислять отдельные компоненты ДПФ, а не все преобразование целиком. Алгоритм Герцеля может быть полезен, когда требуется вычислить только несколько определенных частот.

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. Это происходит за счет следующих преобразований:

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. Обычно данный алгоритм реализуют рекурсивно. Схема алгоритма представлена на рисунке 20. 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.

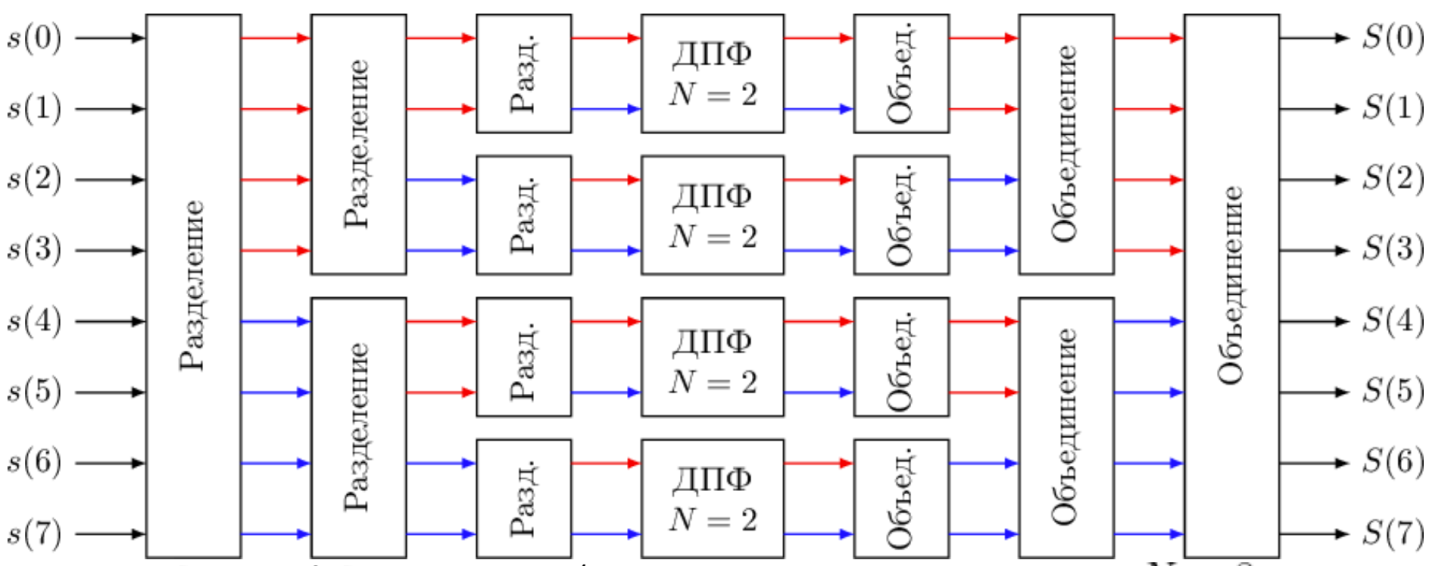


Рисунок 20 – Схема работы алгоритма Кули-Тьюки

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. 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.

В Приложении Д.2 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.

Следует подчеркнуть принципиальный момент: разработанная программная реализация алгоритма была протестирована в среде персонального компьютера, оснащенного высокопроизводительным процессором, обладающим значительным запасом вычислительной мощности. Однако, конечной целью проекта является реализация данного алгоритма на базе микроконтроллера STM32, который характеризуется существенно более скромными вычислительными ресурсами. Анализ результатов, полученных в ходе тестирования на ПК, позволяет сделать важный вывод: непосредственное перенесение традиционных алгоритмов обработки сигнала на платформу STM32 представляется нецелесообразным, поскольку их производительность значительно уступает производительности оптимизированных библиотечных функций. В связи с этим, дальнейшие исследования должны быть направлены на поиск путей оптимизации алгоритма, либо на разработку новых, более эффективных подходов, специально адаптированных для работы на микроконтроллерах STM32. Одним из перспективных направлений оптимизации является уменьшение количества используемых гармоник в процессе модуляции, что позволит существенно снизить вычислительную нагрузку и повысить производительность системы в целом. Также, необходимо учитывать особенности архитектуры STM32 и использовать специфические оптимизации, доступные для данной платформы. Таким образом, для успешной реализации проекта, необходимо разработать алгоритм, который будет эффективно использовать ограниченные ресурсы микроконтроллера STM32, обеспечивая при этом требуемую производительность и качество сигнала.

Попытки увеличить число гармоник не приведут к увеличению точности, но преобразования будут занимать больше времени. Это представлено на рисунке 21. Также можно сделать вывод, что не всегда оптимальное число гармоник должно быть степенью двойки.

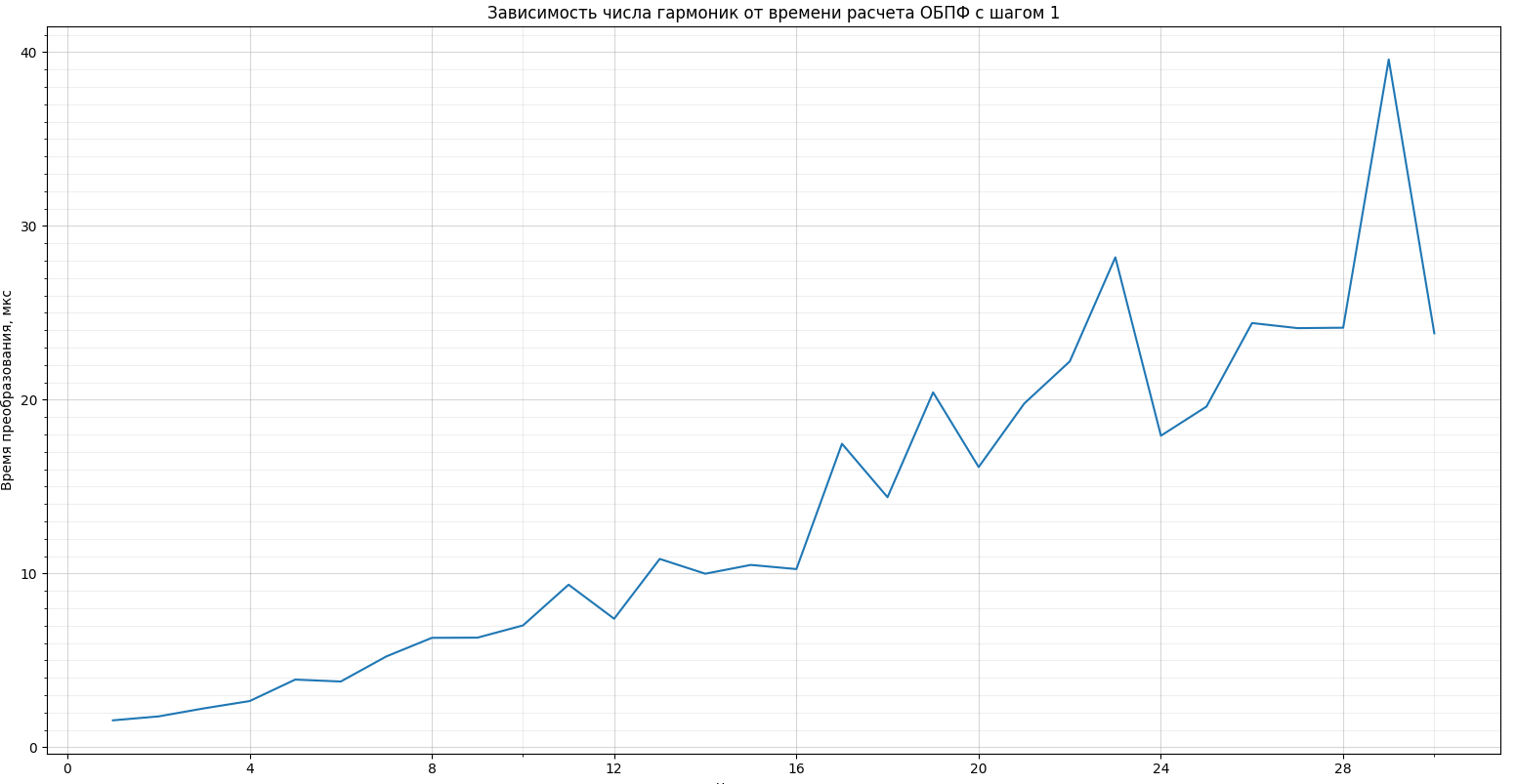


Рисунок 21 – График зависимости скорости работы ОБПФ от числа гармоник

Из-за конкретизированных преобразований в блоке Mapper на вход данного блока могут поступать 16 конкретных комплексных чисел. Имея фиксированные данные, можно заранее рассчитать все возможные варианты сигнала на выходе блока IFFT. Данное упрощение не повлияет на точность вычислений, но вместо расчета ОПФ можно использовать заранее посчитанные данные, из-за чего множество операций умножения, сложения и возведения в степень заменится операцией присваивания в более меньшем объеме, что в разы увеличит скорость работы блока, хоть и потребует хранить некоторый объем данных.

Для подтверждения корректности работы программы в основную реализацию был добавлен вывод данных на каждом этапе с обратным процессом. В Приложении Г представлен вывод, проанализировав который можно сказать, что процесс кодирования и декодирования проходит успешно. В Приложении В представлен листинг программы.

## **P2S**

Данный блок, представленный на рисунке 22, 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. Сигналы будут передаваться подряд в функцию отправки данных. Задача данной функции – правильно расположить данные, чтобы функция передачи данных без лишних операций работала с единым набором данных.

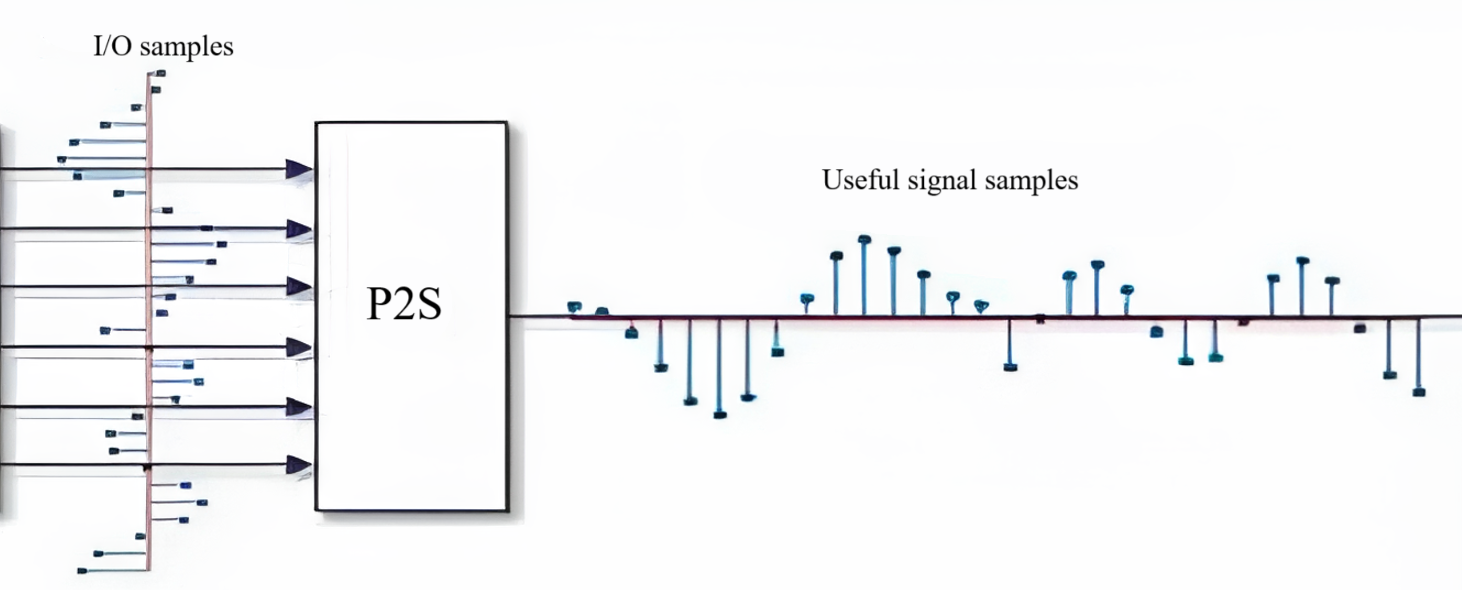
****

Рисунок 22 – Схема блока P2S

# **Оптимизация алгоритма**

Как ранее отмечалось, алгоритм должен выполняться быстро. Рассмотрим блок-схему алгоритма, представленную на рисунке 23. Из описания блоков можно сказать, что самым нагруженным местом в алгоритме является IFFT, где происходит обратное преобразование Фурье. Существует множество способов оптимизации, начиная от заранее просчитанных коэффициентов экспонент и заканчивая алгоритмами БПФ или специализированными библиотеками, к примеру, FFTW3 для ускорения вычислений.

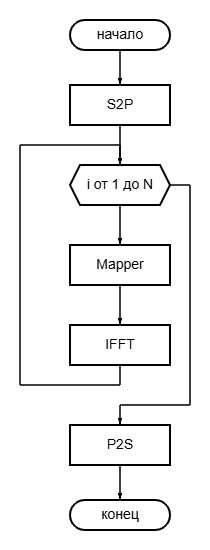


Рисунок 23 – Блок-схема алгоритма OFDM

Условия реализации данного алгоритма устанавливают, что в сигнале ОПФ будет передаваться только один сигнал из 16 возможных, что все 16 сигналов заранее определены и неизменны и что они передаются с использованием одного и того же числа гармоник (что означает, что размер выходных массивов для каждого сигнала одинаковый). То есть каждый раз при вычислениях в функции IFFT может прийти один из 16 возможных сигналов и в зависимости от этого сигнала будет получен соответствующий этому сигналу массив комплексных чисел, число вариантов которых также 16. Значит, вместо попыток оптимизации самого алгоритма ОПФ можно заранее посчитать 16 возможных вариантов выходных массивов, сохранить их в памяти и присваивать соответствующему сигналу свой массив.

Более того, теперь можно объединить блоки Mapper и IFFT и вместо преобразования набора бит в соответствующее комплексное число, передачи его в блок IFFT и присваивания этому числу соответствующий массив можно сразу присваивать набору бит соответствующий массив. Условная схема оптимизации блоков изображена на рисунке 24.

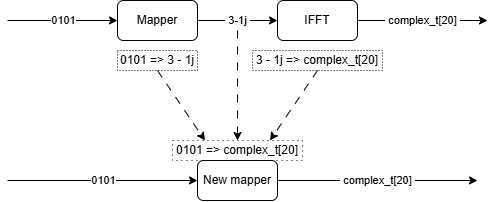


Рисунок 24 – Схема оптимизации IFFT и Mapper

В итоге за счет хранения 16 массивов по 20 комплексных чисел получилось сократить число операций с: передачи данных в функцию Mapper, нахождение соответствующего сигнала, передачи этого сигнала в функцию IFFT, проведение ОПФ над сигналом с 20-тью гармониками (20\*20 вычислений по формуле ОПФ), передача данных из IFFT; до: передачи данных в функцию New mapper, присваивание сигналу соответствующего ему массива ОПФ, передача данных из New mapper [9, 10].

# **Реализация программы на микроконтроллере**

## **Синхронизация**

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 [5]. Существует несколько решений для организации параллельной работы на STM32.

Один из них – RTOS. Он 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 [6, 8].

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. Прерывания бывают 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) [7, 11].

Всего в системе используется три прерывания.

* Инициализация: активирует цикл работы программы. Запускает работу всех функций. Подача сигнала на выбранный канал как инициализация начала передачи;
* Отправка данных: отправляет на передающий модуль данные, полученные на предыдущем цикле работы вычислительной функции. Частота определяется необходимой скоростью передачи данных;
* Прием битового потока: отвечает за прием входящего бита и его запись в память для последующей обработки. Частота этого прерывания зависит от скорости входящего битового потока. Когда данные сформированы, передает их в переменную в вычислительную функцию и выставляет флаг принятых данных в значение 1.

На рисунке 25 представлена подробная блок-схема, иллюстрирующая логику работы системы обработки и передачи данных. Процесс начинается с поступления в систему информации в виде непрерывного битового потока, характеризующегося определенной частотой следования бит. Функция получения данных, являющаяся первым этапом обработки, осуществляет захват этих бит из потока и их последовательную запись в специально выделенную переменную, либо, в случае многоканальной системы, в массив переменных, предназначенный для хранения данных с каждого канала. После завершения процесса считывания и записи необходимого количества бит, соответствующего заданному формату данных, собранная информация передается в ключевую вычислительную функцию, где происходит дальнейшая обработка и модуляция сигнала. Одновременно с передачей данных выставляется флаг актуальности данных, специфичный для данной переменной (или канала). Этот флаг служит сигналом для вычислительной функции о том, что входные данные готовы к обработке и могут быть использованы в расчетах. Данный подход становится возможным благодаря тому, что в вычислительной функции, на самом первом этапе ее работы, входящие данные, полученные из битового потока, незамедлительно преобразуются в требуемый, строго определенный набор бит, размещенный в отдельных, специально предназначенных переменных. Такая организация позволяет обеспечить гибкость и модульность системы, упрощая процесс обработки данных и позволяя оптимизировать алгоритмы вычислений. Кроме того, разделение данных по отдельным переменным обеспечивает возможность параллельной обработки информации, что повышает общую производительность системы.

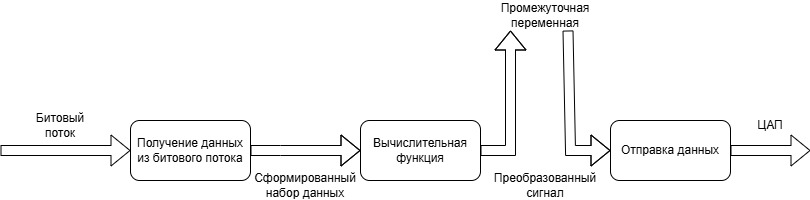


Рисунок 25 – Схема взаимосвязи блоков

Далее в работу включается ключевая вычислительная функция, ответственная за обработку сигнала и формирование выходных данных. Активация данной функции происходит по сигналу, поступающему от установленного флага, свидетельствующего о готовности входных данных. По получении этого сигнала функция приступает к последовательному выполнению сложного алгоритма, детальная блок-схема которого представлена на рисунке 26. На начальном этапе происходит обнуление флага актуальности входных данных, что необходимо для синхронизации и исключения повторной обработки одних и тех же данных. Затем, полученный битовый поток, содержащий информацию, подлежащую передаче, тщательно анализируется и раскладывается на N отдельных переменных, каждая из которых представляет собой элемент модулируемого сигнала. На следующем, критически важном этапе, для каждой из этих N переменных, в процессе модуляции, незамедлительно назначаются соответствующие гармоники. Предварительно, перед началом основного цикла обработки данных, производится комплексный расчет необходимых гармоник, что значительно повышает эффективность и скорость работы алгоритма. После расчета и присвоения гармоник каждой переменной, осуществляется процесс их объединения в единый, целостный массив. Этот массив представляет собой модулированный сигнал, готовый к дальнейшей передаче. В случае, когда флаг актуальности выходных данных находится в выключенном состоянии, что свидетельствует об отсутствии необходимости в немедленной передаче сигнала, сформированные и обработанные данные копируются во временную переменную. Это позволяет сохранить результаты вычислений и избежать их потери до момента, когда система будет готова к их использованию. Такой подход обеспечивает непрерывность и надежность работы приемопередатчика, а также позволяет эффективно управлять потоком данных и ресурсами системы.

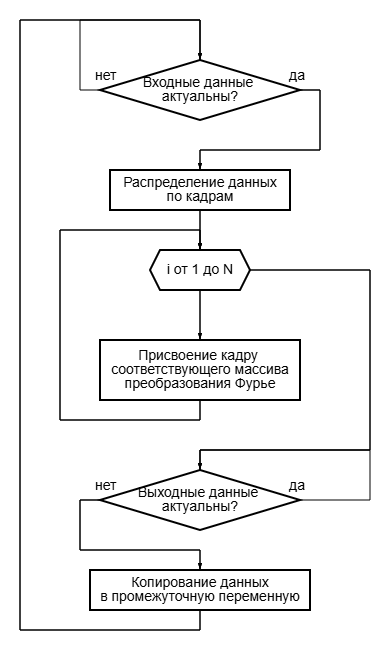


Рисунок 26 – Блок-схема основной вычислительной функции

Функция отправки данных играет ключевую роль в формировании и передаче выходного сигнала. Она осуществляет перенос обработанных данных с заданной, строго контролируемой частотой в цифро-аналоговый преобразователь (ЦАП) для последующего формирования аналогового сигнала и его передачи в канал связи. После завершения передачи данных, соответствующей последней переменной в процессе модуляции, происходит важный этап обновления данных. Информация, сохраненная во временной переменной, тщательно скопированной с основного массива данных, переносится обратно в основную переменную данной функции, обеспечивая актуальность выходного сигнала для следующего цикла передачи. Параллельно с этим, выставляется флаг неактуальности выходных данных, сигнализирующий о том, что текущий цикл передачи завершен и функция готова к обработке новой порции данных.

Особое внимание уделяется тактированию функции отправки данных, которое жестко задается и контролируется. Это обусловлено тем, что данная функция напрямую определяет синхронизацию передаваемого сигнала с приемником, а также обеспечивает его стабильность и соответствие заданным параметрам. Любые отклонения в тактировании могут привести к сбоям в синхронизации и ухудшению качества передаваемого сигнала. Частота работы данной функции является комплексным параметром, который зависит от целого ряда факторов, включая количество используемых гармоник, число каналов передачи данных, выбранный метод модуляции, а также входную частоту сигнала. Эти факторы оказывают непосредственное влияние на объем данных, которые необходимо обработать и передать за единицу времени, что, в свою очередь, определяет требуемую частоту работы функции отправки данных. Если взять N каналов в системе, P гармоник, модуляцию 16-QAM и скорость входного потока – K Гц, то частота работы данной функции должна быть . По данной формуле рассчитывается необходимая для данного набора скорость, так как при уменьшении скорости не все данные будут успевать доходить, а при увеличении данные не будут успевать обрабатываться входной функцией.

## **Приоритетность**

Приоритетность прерываний представляет собой механизм, позволяющий системе микроконтроллера определять порядок обработки различных событий, возникающих асинхронно. Каждому прерыванию присваивается определенный уровень приоритета, указывающий, насколько срочным является выполнение связанного с ним кода. Прерывания с более высоким приоритетом прерывают выполнение кода, связанного с прерываниями более низкого приоритета, обеспечивая своевременную реакцию на критически важные события.

Необходимость в использовании приоритетности прерываний продиктована реалиями работы микроконтроллеров в условиях многозадачности и асинхронности. В реальных системах, особенно в системах управления, сбора данных или связи, микроконтроллер зачастую сталкивается с ситуациями, когда несколько событий, требующих немедленной реакции, возникают одновременно или с минимальным временным интервалом. Эти события могут быть связаны с внешними сигналами, таймерами, протоколами связи или внутренними ошибками. Без эффективного механизма приоритетов, позволяющего определить порядок обработки таких событий, микроконтроллер вынужден был бы обрабатывать прерывания строго в порядке их поступления, формируя своеобразную очередь. Такой подход, однако, может привести к критическим задержкам в обработке наиболее важных событий, например, сигналов аварийной остановки, что, в свою очередь, способно вызвать непредсказуемое и, возможно, опасное поведение всей системы, привести к сбоям в работе оборудования или даже к возникновению аварийных ситуаций. Приоритетность прерываний позволяет избежать этих проблем, обеспечивая своевременную и гарантированную реакцию на критически важные события, независимо от наличия менее важных прерываний, ожидающих обработки.

Некорректная расстановка приоритетов прерываний может иметь серьезные негативные последствия для функционирования системы. Если, например, прерывание, отвечающее за обработку критической ошибки, имеет более низкий приоритет, чем прерывание, связанное с отображением информации на экране, то система может проигнорировать критическую ошибку и продолжить работу в неисправном состоянии, что потенциально может привести к аварийной ситуации или потере данных. Кроме того, неправильная приоритезация прерываний может приводить к инверсии приоритетов, когда задача с высоким приоритетом вынуждена ожидать завершения задачи с низким приоритетом, что значительно снижает общую производительность системы и нарушает требования реального времени. Поэтому, грамотное проектирование системы приоритетов прерываний является критически важным этапом разработки надежных и эффективных систем на базе микроконтроллеров.

Для установки приоритетов прерываний углубимся в работу функций. На STM32 нельзя задавать сдвиг тактирования, то есть нельзя сделать так, чтобы прерывания одной и той же частоты вызывались в разное время. Визуализация представлена на рисунке27. Эту проблему можно решить расстановкой правильных приоритеты для каждого прерывания. Приоритеты прерывания определяют, какое прерывание может быть прервано другим прерыванием.

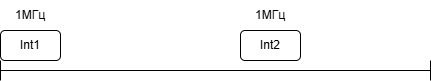


Рисунок 27 – Визуализация невозможной ситуации тактирования

Рассмотрим функциональные блоки. Ранее было сказано, что вычислительная функция работает не как прерывание, а в качестве основной функции работы программы. Функция начала вычислений также не будет рассматриваться, так как это механизм инициализации вычислений, который не входит в основной цикл работы программы.

Рассмотрим отправку данных в ЦАП. За счет отправки данных из данного блока обеспечивается основная синхронизация между приемником и передатчиком. Из этого можно сказать, что данное прерывание должно иметь наивысший приоритет и минимум действий. Рассмотрим получение данных из битового потока. Для этого проанализируем схему входа битового потока, представленную на рисунке 28. По ней видно, что данные можно фиксировать не в конкретной точке, а на протяжении всего отрезка, пока данные не изменились. Можно сделать вывод, что данное прерывание должно иметь приоритет ниже, чем прерывание отправки данных в ЦАП.

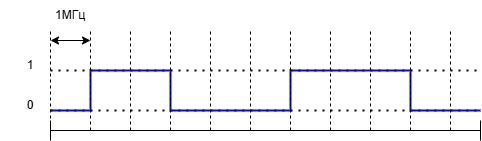


Рисунок28 – Визуализация битового потока со скоростью передачи данных

Применив данные правила приоритетности и тактирования, получим схему распределения задач процессора, представленную на рисунке 29. Здесь представлено то, как процессор будет обрабатывать задачи за четверть цикла основного алгоритма, то есть за то время, в течении которого входящий бит из потока не будет меняться. В начальной точке представленной схемы вызывается 2 прерывания, OUT и IN. Из-за приоритетности, управление будет передано прерыванию OUT, после чего будет работать прерывание IN. Далее в оставшееся время работы цикла будет работать основная вычислительная функция main, которая будет прерываться OUT.

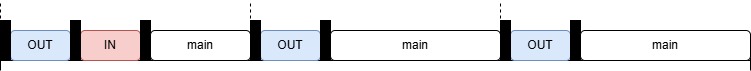


Рисунок 29 – Схема распределения задач на процессоре

Представим альтернативный сценарий, характеризующийся относительно невысокой частотой тактирования процессора, при котором схема распределения вычислительных задач принимает вид, изображенный на рисунке 30. В данной конфигурации, основная вычислительная нагрузка ложится на функцию main, которая выполняет ключевую часть алгоритма обработки сигнала. На первый взгляд, такая схема может вызвать опасения относительно возможности завершения полного рабочего цикла за требуемое время, особенно учитывая необходимость выполнения других задач, таких как управление периферией и обработка прерываний. Однако, необходимо учитывать, что основной алгоритм обработки был подвергнут существенной оптимизации и упрощению. Кроме того, следует отметить, что рассматриваемая схема на рисунке 30 отражает лишь четвертую часть общего вычислительного цикла, а не весь процесс целиком. Поэтому, несмотря на кажущуюся перегруженность функции main, нельзя с уверенностью утверждать, что времени, отведенного на выполнение вычислительной функции, окажется недостаточно для успешного завершения цикла. Для получения окончательного и неоспоримого подтверждения, необходимо провести серию всесторонних тестов производительности на различных типах микроконтроллеров, характеризующихся различной частотой тактирования. Эти тесты позволят оценить реальное время выполнения вычислительной функции в различных условиях и определить оптимальную конфигурацию системы для достижения максимальной производительности. Только на основании результатов этих тестов можно будет сделать обоснованные выводы о применимости данной схемы распределения задач и ее эффективности в различных сценариях.

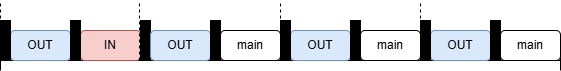


Рисунок 30 – Схема распределения задач на процессоре с меньшей частотой

# **Тестирование алгоритма на микроконтроллерах**

Для тестирования алгоритма был выбран микроконтроллер на базе ядра Cortex-M3: STM32F103RB. Данное ядро является одним из самых популярных в использовании как в учебных или личных целей, так и в различных повседневных устройствах.

Оно построено на 32-битной архитектуре, имеет 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), энергоэффективно, оптимизировано для работы с ограниченными ресурсами.

Микроконтроллер STM32F103RB имеет следующие характеристики:

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

Оптимальнее было бы использовать процессор Cortex-M4. Он 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.

Проект конфигурировался с использованием STM32CubeMX. На микроконтроллере был подключен USART-порт для получения данных, 2 модуля таймера для тактирования прерываний. При тестировании менялись коэффициенты тактирования так, чтобы удовлетворять следующему условию: , где – частота прерывания, получающего данные, – частота прерывания, отправляющее данные. Для тестирования приоритетность прерываний были 1 для OUT, 2 для IN. Были выбраны данные прерывания для того, чтобы поставить прерывания UART выше, чем OUT, который имеет наивысший приоритет, чтобы гарантировано видеть вывод при недостаточной скорости обработки данных. Частота работы ядра была установлена 64 МГц (частота 72 МГц недоступна к выставлению в самой программе), тактирование осуществлялось от внутренней RC цепи.

После конфигурации и полной отладки, в режиме сборки Release и с выставлением дополнительных флагов компиляции (-O2) программа была запущена и протестирована. Тестирование проходило следующим образом: составлена Таблица 1 для возможных значений частот, удовлетворяющих условию выше; в прерывания была добавлена проверка, которая активируется при запросе новых данных; если данные нельзя запросить – поток выводит об этом информацию в последовательный порт, иначе просто получает данные и продолжает работать.

Таблица 1 – Отчетность тестирования программы при изменяющихся частотах прерываний IN и OUT

|  |  |  |
| --- | --- | --- |
| IN(KHz) | OUT(KHz) | Прохождение теста |
| 1 | 5 | + |
| 2 | 10 | + |
| 4 | 20 | + |
| 5 | 25 | + |
| 8 | 40 | + |
| 10 | 50 | + |
| 16 | 80 | + |
| 20 | 100 | + |
| 25 | 125 | + |
| 32 | 160 | + |
| 40 | 200 | + |
| 50 | 250 | + |
| 64 | 320 | + |
| 80 | 400 | - |

Анализ полученных результатов позволяет сделать вывод о том, что на выбранных настройках тактирования и конфигурации периферийных устройств, максимальная производительность используемого микроконтроллера STM32F103RB ограничивается частотой 64 КГц для прерывания, связанного с получением одного бита данных. Иными словами, это означает, что достижимая скорость приема и обработки входящих данных на данном микроконтроллере составляет 64 000 бит в секунду (64 Кбит/с). Следовательно, если один микроконтроллер STM32F103RB будет выполнять все задачи, связанные с приемом, передачей и обработкой данных, включая захват входящего битового потока, модуляцию и демодуляцию, а также передачу обработанного сигнала, то он будет способен осуществлять прием и отправку данных на скорости, не превышающей 64 000 бит в секунду. Дополнительно, в рамках проведенного исследования, было выполнено измерение времени работы исключительно вычислительной функции, без учета затрат времени на прерывания и ожидание поступления новых данных. Результаты тестирования показали, что вычислительная функция, работая в изолированном режиме, без каких-либо задержек и пауз, способна выполнить поставленную задачу за время, составляющее всего 2 микросекунды. Данный показатель подчеркивает высокую эффективность алгоритма и оптимизированность кода вычислительной функции, что, в свою очередь, позволяет минимизировать задержки и повысить общую производительность системы обработки сигнала.

Перспективным направлением развития системы является переход к многопроцессорной архитектуре, при которой функциональность распределяется между несколькими независимыми процессорами, каждый из которых отвечает за выполнение определенного набора задач. Такой подход позволяет значительно повысить производительность системы, распараллелить вычислительные процессы и снизить нагрузку на каждый отдельный процессор. В рамках данной архитектуры, каждый процессор будет специализироваться на выполнении конкретной задачи: один процессор может отвечать за прием и первичную обработку данных, другой – за сложные вычисления и модуляцию, а третий – за управление периферийными устройствами и передачу сигнала. Подобная специализация позволяет оптимизировать каждый процессор под конкретный тип нагрузки, что, в свою очередь, повышает эффективность использования ресурсов и снижает энергопотребление. Однако, для успешной работы многопроцессорной системы необходимо обеспечить эффективный и надежный обмен данными между процессорами. Существует несколько способов организации такого обмена, каждый из которых имеет свои преимущества и недостатки:

* Общая память: Процессоры могут обмениваться данными через общую область памяти, доступную для всех участников системы. Этот метод обеспечивает высокую скорость обмена данными, но требует careful synchronization, чтобы избежать конфликтов и повреждения данных. Для синхронизации могут использоваться семафоры, мьютексы и другие механизмы.
* Прямой обмен данными (DMA): Процессоры могут напрямую передавать данные друг другу, используя механизмы прямого доступа к памяти (DMA). Это позволяет минимизировать нагрузку на центральные процессоры и обеспечивает высокую скорость обмена данными.
* Последовательные интерфейсы (SPI, UART, I2C): Процессоры могут обмениваться данными, используя стандартные последовательные интерфейсы. Этот метод является простым в реализации, но имеет относительно низкую скорость обмена данными.
* Ethernet: Для систем, требующих высокой скорости обмена данными и возможности масштабирования, можно использовать сеть Ethernet. Этот метод обеспечивает высокую пропускную способность, но требует более сложной аппаратной и программной реализации.

# **ЗАКЛЮЧЕНИЕ**

В рамках данной работы был проведен анализ алгоритмов OFDM, разбор работы каждого функционального блока и его программные реализации, после чего была проведена оптимизация алгоритмов под конкретные условия. Далее были рассмотрены проблемы синхронизации при отправке и получении данных и методы решения данной проблемы в рамках микроконтроллеров. Также в рамках решения задачи на одном ядре было посчитано необходимое соотношение тактирования между прерываниями. В итоге была разработана, реализована и протестирована программа полного цикла передачи информации, основанная на системах прерывания по таймеру.

Экспериментальным путем было выяснено, что микроконтроллер STM32F103RB на базе ядра Cortex-M3 с тактовой частотой 64 МГц, используя 2 аппаратных таймера с разной частотой для реализации прерываний может получать, обрабатывать и передавать данные со скоростью 64000 бит\с, используя 20 гармоник для передачи одного бита информации. Следующим шагом данной работы является использование полученных данных и алгоритмов для распределения задач на несколько отдельных микроконтроллеров оптимальным образом в зависимости от необходимой скорости работы функционального или функциональных блоков с целью распределения функций этапов вычислений на отдельные ядра. Данная реализация позволит в разы увеличить скорость работы всей системы из-за отсутствия необходимости выполнять 3 задачи параллельно на одном ядре, а также уменьшит необходимые требования к вычислительным возможностям используемых ядер.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Sohrabi F., Yu W. Hybrid Analog and Digital Beamforming for mmWave OFDM Large-Scale Antenna Arrays // IEEE Journal on Selected Areas in Communications. – 2017. – Т. 35. – No 9. – С. 2141-2156. – DOI: 10.1109/JSAC.2017.2721882.
2. Suh S., et al. Low-Power Discrete Fourier Transform for OFDM: A Programmable Analog Approach // IEEE Transactions on Very Large Scale Integration (VLSI) Systems. – 2012. – Т. 20. – No 6. – С. 1025-1034. – DOI: 10.1109/TVLSI.2011.2157508.
3. Кандидов В. П., Чесноков С. С., Шленов С. А. Дискретное преобразование Фурье: Учебно-методическое пособие. – Москва: Московский государственный университет имени М. В. Ломоносова, Физический факультет, 2019. – ISBN 978-5-8279-0179-2.
4. Юнаковский А. Гармонический анализ. Ряды Фурье, преобразование Фурье и приложения БПФ: Учебное пособие. – Издательский Дом Интеллект, 2024. – ISBN 978-5-91559-318-2.
5. Noviello C. Mastering STM32. – Leanpub, 2018. – Перевод: Дмитрий Карасёв, 2021. [Электронный ресурс]. URL: https://psv4.userapi.com/s/v1/d/nNFqfKlntE8UuofK\_ZwZs3REvPJBNHOj61DNTSOnUeK1Td322ZcO1iRe3UzLUyqiKBDxNrRGmJGrFmBS62afV646tkW5Pw9gNXRIEcmnjsyUp6ilir2VuQ/Karmin\_Noviello\_-\_Osvoenie\_STM32.pdf (дата обращения: 15.04.2025).
6. Мединцев В. Операционные системы микроконтроллеров – На примере операционной системы реального времени FreeRTOS. – [б. м.]: Издательские решения, 2023. – 228 с. – ISBN 978-5-0060-0974-5.
7. Белов А. В. Микроконтроллеры AVR: от азов программирования до создания практических устройств. – Изд. 2-е, перераб. и доп. + виртуальный диск с видеокурсами. — СПб.: Наука и Техника, 2020. – 544 с. – ISBN 978-5-94387-874-9.
8. Васильев А. С., Лашманов О. Ю., Пантюшин А. В. Основы программирования микроконтроллеров. – СПб: Университет ИТМО, 2016. – 95с. [Электронный ресурс]. URL: https://books.ifmo.ru/file/pdf/2031.pdf (дата обращения: 20.05.2025).
9. Гантерот К. Оптимизация программ на C++. Проверенные методы повышения производительности. – Диалектика, 2019. – ISBN 978-5-907144-58-3.
10. Касперски К. Техника оптимизации программ. Эффективное использование памяти. – БХВ-Петербург, 2003. – ISBN 5-94157-232-8.
11. Гей У. Основы STM32. Изучаем FreeRTOS, libopencm3 и GCC. – ДМК Пресс, 2024. – ISBN 978-5-93700-294-5.

# **ПРИЛОЖЕНИЕ А**

Подробное описание алгоритма OFDM представлено на:

Лист 1-3. Описание работы алгоритма OFDM.

Принцип работы приемопередатчика на основе OFDM. 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.

# **ПРИЛОЖЕНИЕ Б**

Подробное описание возможных типов модуляции представлено на:

Лист 1-3. Описание типов модуляции.

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:

При фазовой модуляции (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.

При амплитудной модуляции (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.

В квадратурной амплитудной модуляции (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 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.

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 В**

Листинг программы эмулирования блоков для корректной отладки представлен на:

Лист 1-11. Листинг программы эмулирования блоков для корректной отладки.

Листинг 1. Программа эмулирования блоков для корректной отладки

/\*

Обязательные флаги при компиляции: -lfftw3 -lm

\*/

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<cstring>

#include<iostream>

#include<math.h>

#include <bitset>

#include<fftw3.h>

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

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

constexpr int DATA\_LEN = 14191;

constexpr int COUNT\_CADR = 12; // Количество кадров

constexpr int COUNT\_TESTS = 1;

constexpr double normalizeCoef = 1/sqrt(N); // Нормализующий коэффициент

#define RECOWER // Раскомментировать если нужно провести обратный процесс

unsigned long timeStart;

int i;

int iGl = 0;

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\*);

int main()

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

{

// Выделение памяти для входных и выходных данных

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,

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

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:

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

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;

unsigned char cadrs[COUNT\_CADR];

char buff[DATA\_LEN] = "asdivjjfnaksakdfjdshsdfsdhkaskj";

int maxLen = strlen(buff);

#ifdef RECOWER

char buff\_out[COUNT\_CADR];

unsigned char cadrs\_out[COUNT\_CADR];

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

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");

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

// timeStart = time(NULL);

long l = 0;

long long glob = 0;

int lss = 1;

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

{

// printf("Тест %d\n", lss);

l = 0;

// 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)

while (iGl < maxLen)

{

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");

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

// 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);

#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

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

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]);

#endif

++l;

}

printf("Число пройденных циклов: %ld\n", l);

glob += l;

}

printf("Итоговое среднее число пройденных циклов: %lld\n", glob/COUNT\_TESTS);

// Здесь делаем ОБПФ для каждого кадра по отдельности

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

// 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;

}

}

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

{

char c;

int i = -1;

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

buff[i] = 0;

\*maxLen = i;

}

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

// 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.

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

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

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)

}

}

}

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)

{

Продолжение листинга 1. Программа эмулирования блоков для корректной отладки

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;

}

}

}

# **ПРИЛОЖЕНИЕ Г**

Выводы программы во время отладки алгоритмов представлены на:

Лист 1-5. Листинг вывода программы.

Лист 5. Листинг вывода потребления ресурсов при определенном числе гармоник.

Листинг 2. Вывод программы

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

Продолжение листинга 2. Вывод программы

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

Продолжение листинга 2. Вывод программы

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

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:

Продолжение листинга 2. Вывод программы

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

Продолжение листинга 2. Вывод программы

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

Листинг 3. Вывод программы расчета потребляемых ресурсов

Count garmon = 20, Count tests = 100000

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