Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности

**Высшая школа интеллектуальных систем и суперкомпьютерных технологий**

**Разработка спектроанализатора на основе платы цифрового радио HackRF в среде визуального программирования LABVIEW**

по дисциплине «Аппаратно-программные средства систем управления»

Выполнил:

студент гр.5140901/31501 Д.С. Кузик

Руководитель:

Сушников Виктор Александрович

Санкт-Петербург

2024

Оглавление

[Описание аппаратной платформы. 3](#_Toc186726673)

[Работа с платой цифрового радио. 5](#_Toc186726674)

[Режимы работы HackRF One в режиме приёмника. 5](#_Toc186726675)

[Получение и обработка данных в sweep-режиме. 5](#_Toc186726676)

[Разработка TCP-сервера. 10](#_Toc186726677)

[Разработка спектроанализатора в среде LABVIEW. 13](#_Toc186726678)

[Получение и отображение спектров. 13](#_Toc186726679)

[Управление шириной спектра. 14](#_Toc186726680)

[Тестирование системы. 15](#_Toc186726681)

[Вывод. 16](#_Toc186726682)

[Приложения. 17](#_Toc186726683)

# Описание аппаратной платформы.

HackRF One – это Software Defined Radio (плата цифрового радио) платформа, способная передавать или принимать радиосигналы в диапазоне от 1 МГц до 6 ГГц. HackRF One - это аппаратная платформа с открытым исходным кодом, предназначенная для тестирования и разработки современных приложений по обработке и передачи данных по радиоканалу.

Характеристики

1. Полудуплексный приемопередатчик.
2. Рабочая частота: от 1 МГц до 6 ГГц.
3. Поддерживаемая частота дискретизации: от 2 до 20 MSPS (квадратурная).
4. разрешение: 8 бит IQ.
5. Интерфейс: Высокоскоростной USB (с разъемом USB Micro-B.
6. Источник питания: питание по шине USB.
7. Питание порта антенны с программным управлением (макс. 50 мА при напряжении 3,0-3,3 В).
8. Разъем для антенны SMA (50 Ом).
9. Тактовый вход и выход SMA-разъема для синхронизации (не обязателен для использования).
10. Имеются программируемые кнопки.
11. Имеются контактные разъемы для расширения функционала.

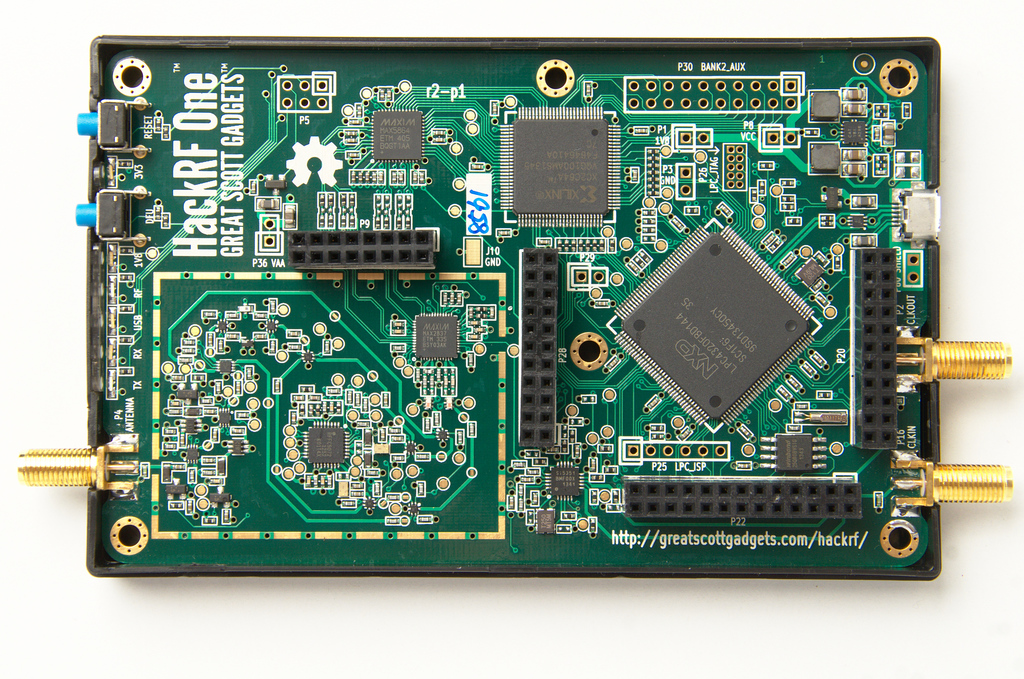


Рисунок 1 HackRF One

# Работа с платой цифрового радио.

## Режимы работы HackRF One в режиме приёмника.

При разработке спектроанализатора необходимо выбрать режим работы приёмника. У HackRF существует два основных режима работы:

**Приём IQ сигналов в полосе до 20 МГц**. HackRF One способен принимать IQ сигналы в полосе до 20 МГц. Этот режим позволяет захватывать и анализировать радиосигналы в реальном времени. IQ сигналы представляют собой комплексные сигналы, состоящие из внутренней (I) и квадратурной (Q) компонент, что позволяет точно воспроизводить амплитуду и фазу радиосигнала. Этот режим особенно полезен для анализа и демодуляции сигналов, но относительно узкая полоса (20 МГц) может стать проблемой при разработке спектроанализатора, для которого не нужна чрезмерная точность полученных измерений, но важна ширина измеряемой полосы.

**Sweep-режим.** Sweep-режим HackRF One сильно отличается от режима приёма IQ сигналов. В этом режиме устройство работает как широкополосный спектроанализатор, последовательно сканирует заданный диапазон частот с частотой до 7 ГГц в секунду, захватывая данные по каждой частоте и возвращая хост-компьютеру уже проебразованные данные с помощью быстрого преобразования Фурье (которое происходи на FPGA платы). Это позволяет быстро оценить спектр радиочастот и выявить активные сигналы в широком диапазоне. Sweep-режим особенно полезен для обнаружения и анализа широкополосных сигналов, а также для мониторинга радиочастотного спектра.

Соответственно, был выбран sweep-режим из-за его быстродействия и ширины полосы.

## Получение и обработка данных в sweep-режиме.

Для получения данных от HackRF One в sweep-режиме на Windows, потребуется использовать соответствующее программное обеспечение. Вместе с HackRF поставляются утилиты для ОС GNU Linux и Windows, позволяющие управлять работой платы.

Для sweep-режима существует утилита “hackrf\_sweep”, которая позволяет получить данные от sweep проходов по заданному диапазону.

Результат работы утилиты выглядит следующим образом:

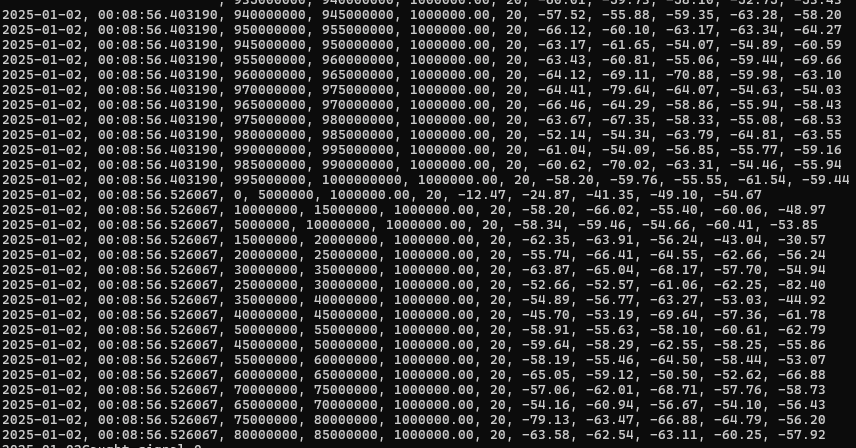


Рисунок 2 Работа утилиты «hackrf\_sweep»

Здесь:

1. Первый и второй столбец – дата и время прохода.
2. Третий столбец: начальная частота прохода.
3. Четвёртый стобец: конечная частота прохода.
4. Пятый столбец: диапазон каждого подпрохода.
5. Шестой столбец: количество измерений, сделанных прохода.
6. Последующие столбцы: амплитуда измеренного сигнала в каждом поддиапазоне, дБ.

Для преобразования данных утилиты в точки графика спектра был написан python-класс:

class HackrfSweepParser:

    def \_\_init\_\_(self, server):

        self.current\_buffer = []

        self.server = server

        self.current\_ranges = (0, 6000)

        self.process = None

        self.parser\_thread = None

    def buffer\_to\_packed\_points(self, buffer):

        result = []

        sorted\_buffer = sorted(buffer, key=lambda x: float(x[1]))

        for line in sorted\_buffer:

            \_, hz\_low, hz\_high, hz\_bin\_width, dbs = line

            for i in range(0, 5):

                result.append(float((int(hz\_low) + int(hz\_low) + float(hz\_bin\_width) \* (i + 1)) / 2)) #x

                result.append(float(dbs[i])) #y

        return result

    def parse\_hackrf\_sweep(self):

        # Команда для запуска hackrf\_sweep

        command = ["hackrf\_sweep", "-f", f"{int(self.current\_ranges[0])}:{int(self.current\_ranges[1])}"]

        try:

            # Запуск процесса

            self.process = subprocess.Popen(

                command,

                stdout=subprocess.PIPE,

                stderr=subprocess.PIPE,

                text=True

            )

            print("Запущено hackrf\_sweep. Ожидание данных...")

            # Обработка вывода в реальном времени

            previous\_date\_time = None

            for line in self.process.stdout:

                # Удаляем лишние пробелы и проверяем, содержит ли строка данные

                line = line.strip()

                if "," in line:

                    # Разделяем строку на части

                    fields = line.split(", ")

                    if len(fields) > 6:  # Проверяем наличие необходимого количества полей

                        date\_time = fields[0] + fields[1]

                        hz\_low = fields[2]

                        hz\_high = fields[3]

                        hz\_bin\_width = fields[4]

                        dbs = [fields[6 + i] for i in range(5)]

                        if int(hz\_low) == int(self.current\_ranges[0])\*1e6 and len(self.current\_buffer) > 0:

                            result = self.buffer\_to\_packed\_points(self.current\_buffer)

                            data\_to\_send = DataPacker.pack\_data(result)

                            self.server.send\_data(data\_to\_send)

                            self.current\_buffer = []

                        self.current\_buffer.append((date\_time, hz\_low, hz\_high, hz\_bin\_width, dbs))

                        previous\_date\_time = date\_time

            print("Данные успешно сохранены в hackrf\_sweep\_output.csv.")

        except KeyboardInterrupt:

            print("Остановка выполнения по Ctrl+C.")

        except Exception as e:

            print(f"Произошла ошибка: {e}")

        finally:

            try:

                if self.process:

                    self.process.terminate()

            except Exception:

                pass

    def restart\_parser(self):

        # Остановка текущего процесса

        if self.process:

            self.process.terminate()

        # Перезапуск парсера в отдельном потоке

        if self.parser\_thread and self.parser\_thread.is\_alive():

            self.parser\_thread.join()

        self.parser\_thread = threading.Thread(target=self.parse\_hackrf\_sweep)

        self.parser\_thread.start()

Листинг 1 Обработка данных "hackrf\_sweep"

Что делает каждый метод:

1. **\_\_init\_\_**: Инициализирует объект класса, устанавливая начальные значения для буфера данных, сервера, диапазона частот и других параметров.
2. **buffer\_to\_packed\_points**: Преобразует буфер данных в формат, удобный для отправки. Он сортирует данные по частоте и создает список точек, где каждая точка состоит из частоты и амплитуды сигнала.
3. **parse\_hackrf\_sweep**: Запускает утилиту hackrf\_sweep для сканирования заданного диапазона частот. Обрабатывает вывод утилиты в реальном времени, сохраняя данные в буфер и отправляя их на сервер, когда буфер заполняется.
4. **restart\_parser**: Останавливает текущий процесс сканирования и перезапускает его в отдельном потоке.

# Разработка TCP-сервера.

Для передачи данных спектров между python программой и Labview был разработан Python TCP-сервер:

class DataPacker:

    @staticmethod

    def pack\_data(data):

        packed\_data = struct.pack(f'>{len(data)}d', \*data)

        packed\_size = struct.pack('>i', len(packed\_data))

        return packed\_size + packed\_data

class Server:

    def \_\_init\_\_(self, host, port):

        self.host = host

        self.port = port

        self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

        self.server\_socket.bind((self.host, self.port))

        self.server\_socket.listen(1)

        self.client\_socket = None

        self.client\_addr = None

    def start(self):

        print(f"Server listening on {self.host}:{self.port}")

        while True:

            print("Waiting for a connection...")

            self.client\_socket, self.client\_addr = self.server\_socket.accept()

            print(f"Connection from {self.client\_addr}")

            self.handle\_client()

    def handle\_client(self):

        try:

            while True:

                # Приём данных от клиента

                data = self.client\_socket.recv(16)

                if not data:

                    print('no data... continue...')

                    continue

                # Распаковка данных

                packed\_size = 16

                packed\_data = data

                new\_ranges = struct.unpack(f'>{packed\_size // 8}d', packed\_data)

                # Отображение данных в командной строке

                print(f"Received data: {new\_ranges}")

                # Проверка и перезапуск парсера, если диапазоны отличаются

                if new\_ranges != self.parser.current\_ranges:

                    self.parser.current\_ranges = new\_ranges

                    self.parser.restart\_parser()

        except (ConnectionResetError, BrokenPipeError):

            print(f"Connection from {self.client\_addr} was closed.")

        finally:

            self.client\_socket.close()

    def send\_data(self, data):

        if self.client\_socket:

            self.client\_socket.sendall(data)

Листинг 2 TCP-Сервер.

Рассмотрим данный код:

**Класс DataPacker**

pack\_data: Этот статический метод упаковывает данные в бинарный формат. Он использует модуль struct для упаковки данных в формат, который можно легко передать по сети.

struct.pack(f'>{len(data)}d', \*data): Упаковывает данные в формат, где каждое число представлено как double (8 байт). Важно: используемый порядок байт – Big-Endian т.к Labview по умолчанию используем именно его.

struct.pack('>i', len(packed\_data)): Упаковывает размер упакованных данных в формат int (4 байта).

**Класс Server**

\_\_init\_\_: Инициализирует сервер, создавая сокет и привязывая его к указанному хосту и порту. Сервер начинает прослушивание входящих соединений.

start: Запускает сервер и ждет входящих соединений. Когда клиент подключается, сервер принимает соединение и обрабатывает его.

handle\_client: Обрабатывает взаимодействие с клиентом.

1. Принимает данные от клиента.
2. Распаковывает данные, используя модуль struct.
3. Отображает полученные данные в командной строке.
4. Если полученные диапазоны частот отличаются от текущих, обновляет диапазоны и перезапускает парсер.

send\_data: Отправляет данные парсера клиенту, если соединение установлено.

Пример использования

1. Клиент подключается к серверу и отправляет данные о новых диапазонах частот.
2. Сервер принимает эти данные, распаковывает их и проверяет, отличаются ли они от текущих диапазонов.
3. Если диапазоны отличаются, сервер обновляет диапазоны и перезапускает парсер для сканирования новых частот.
4. Сервер отправляет данные парсера клиенту, используя метод send\_data.

# Разработка спектроанализатора в среде LABVIEW.

## Получение и отображение спектров.

Для получения данных от сервера использовался TCP-Клиент. Чтение и отображение спектров работает следующим образом:

**TCP Client Loop:**

Этот цикл отвечает за подключение к серверу и получение данных, а также их отображение.

1. Устанавливается TCP-соединение, используя IP и порт введённые на фронт-панели.
2. Если соединение установлено, то клиент читает первые 4 байта полученные от сервера. В этих 4ех байтах должен быть размер массива спектров в формате int32 BigEndian.
3. Читается массив спектров, размер которых определён на предыдущем шаге.
4. Распаковывает полученные данные (формат данных подходит для построения XY графика, полученный массив содержит последовательно x и y параметры для построения точек на графике).
5. Данные отображаются на XY графике.
6. Программа останавливает работу в случае получения TCP ошибки.

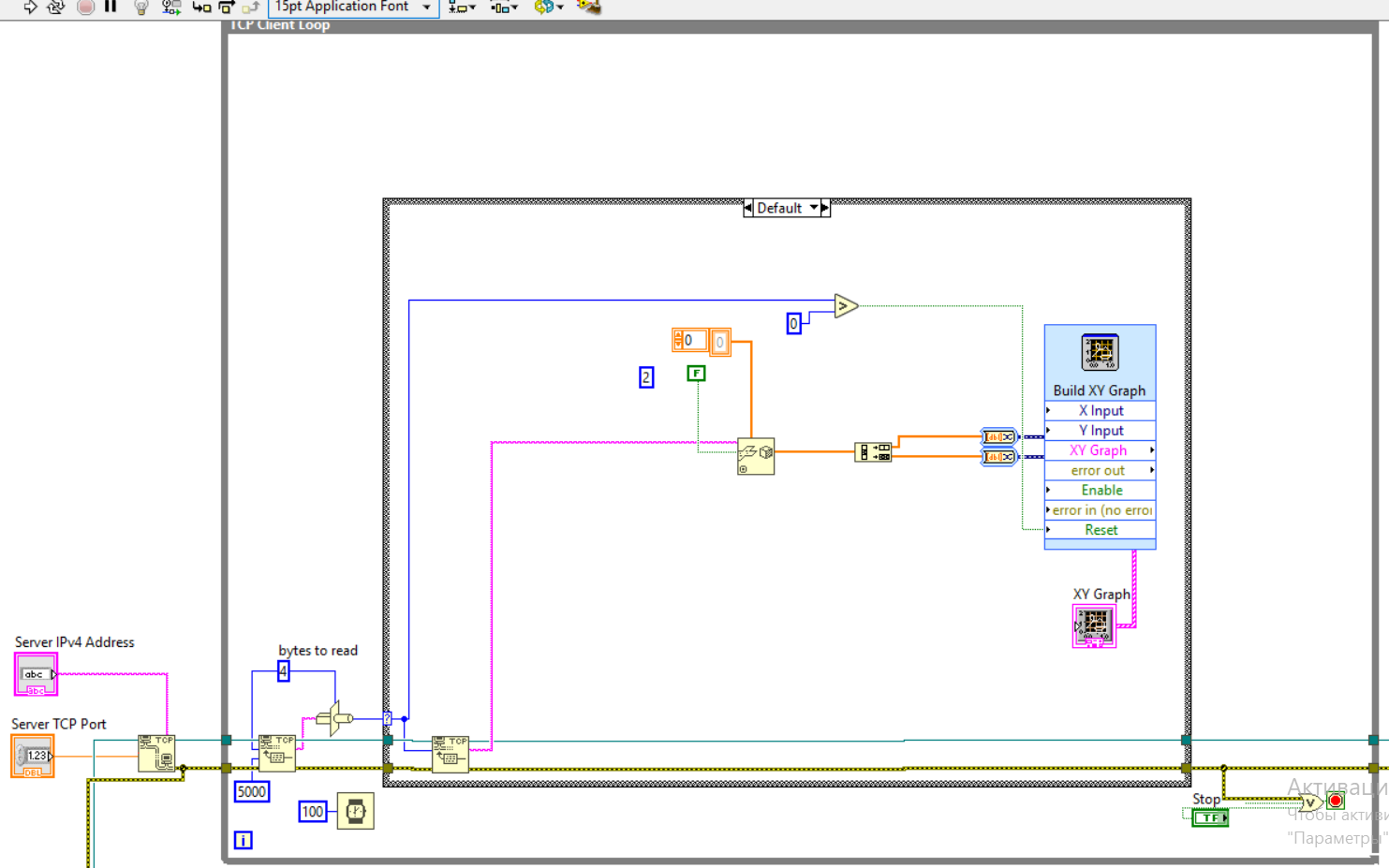


Рисунок 3 Labview TCP-клиент. Чтение и отображение спектров.

## Управление шириной спектра.

Для управления шириной спектра TCP-клиент отправляет серверу два int32 BigEndian числа, объёдинённые в один массив байт. Первое число представляет из себя нижнюю границу частот в МГц, а второе верхнюю границу.

Отправка значений происходит по кнопке.

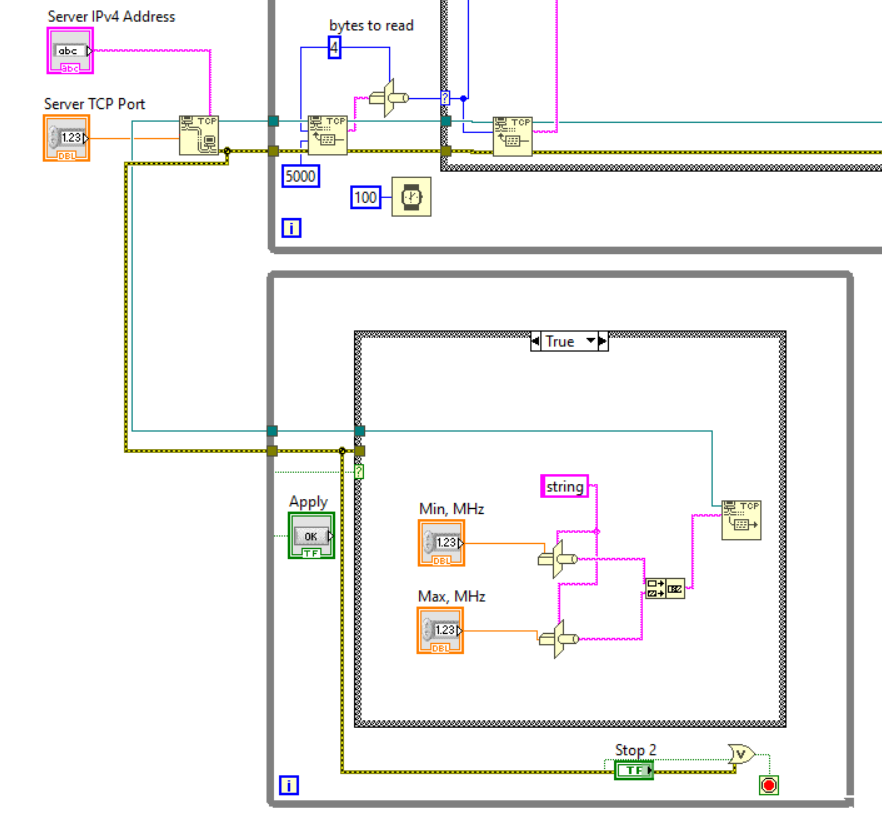


Рисунок 4 Блок-схема отправки новых диапазонов для анализа спектра

## Фронт-панель.

На фронт панели находятся элементы управления и отображения данных:

1. IP и порт TCP-сервера с источником спектров.
2. Минимальная и максимальная границы частот спектров, кнопка для отправки новых границ на сервер.
3. Последняя полученная ошибка от TCP-клиента.
4. Кнопки для остановки циклов TCP-клиента.
5. График спектра и легенда графика.

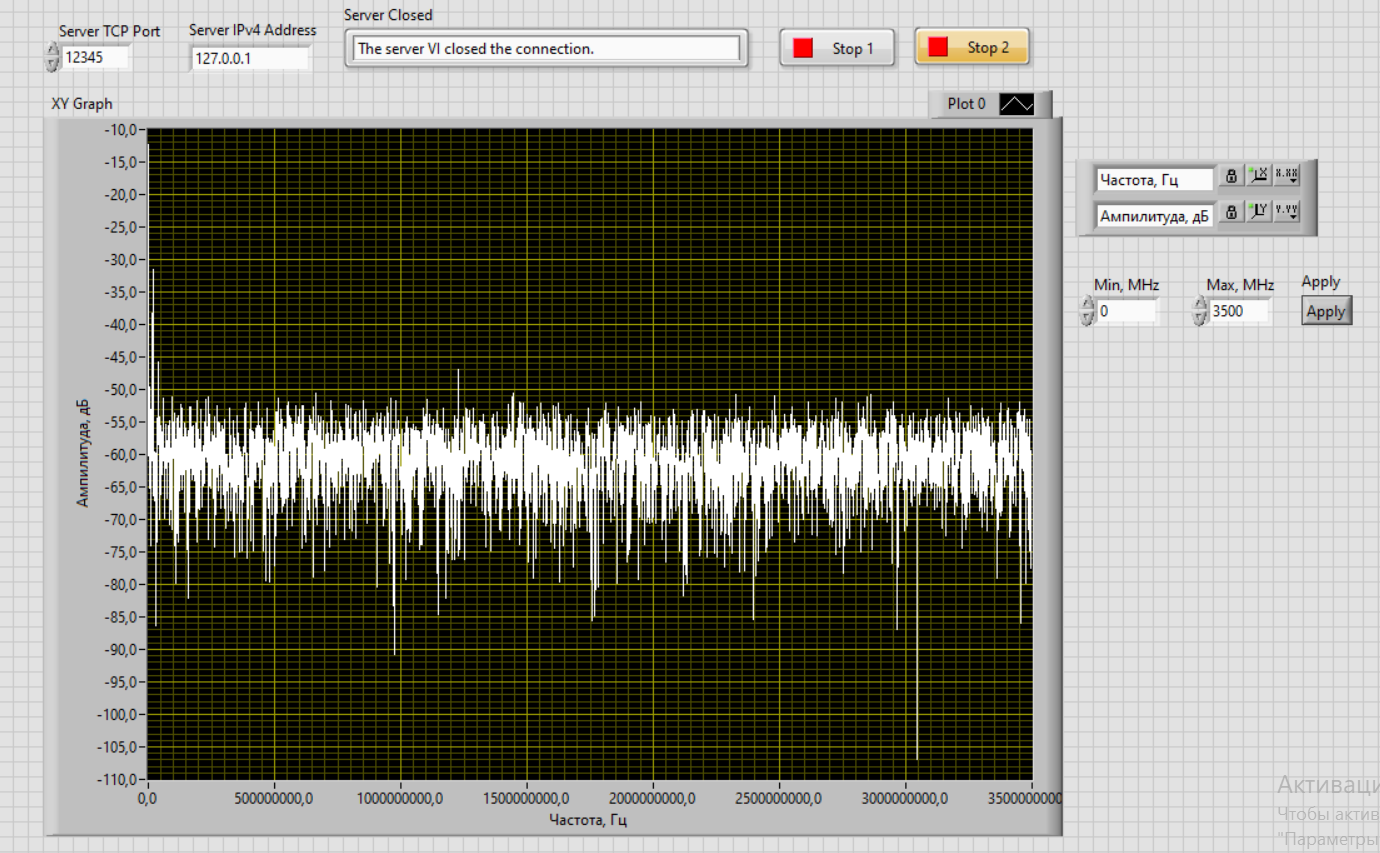


Рисунок 5 Фронт панель разработанного спректроанализатора.

# Тестирование системы.

Подключим HackRF плату без антенны к персональному компьютеру через USB-интерфейс, запустим python сервер:

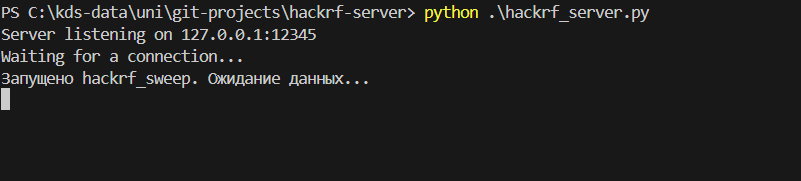


Рисунок 6 Запуск сервера

Подключимся Labview-клиентом к серверу:

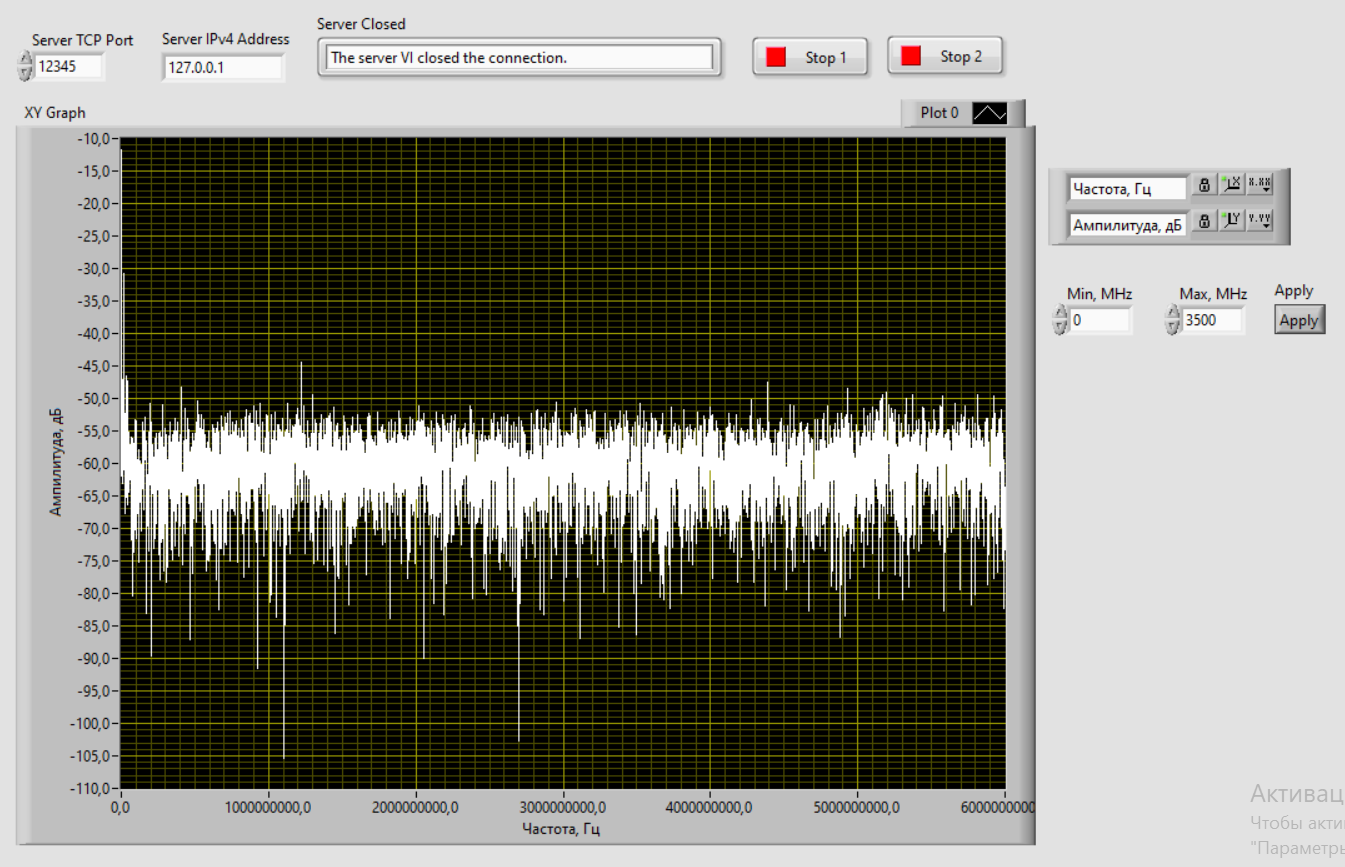


Рисунок 7 Подключение к серверу

Изменим диапазон приёма на 0, 900:

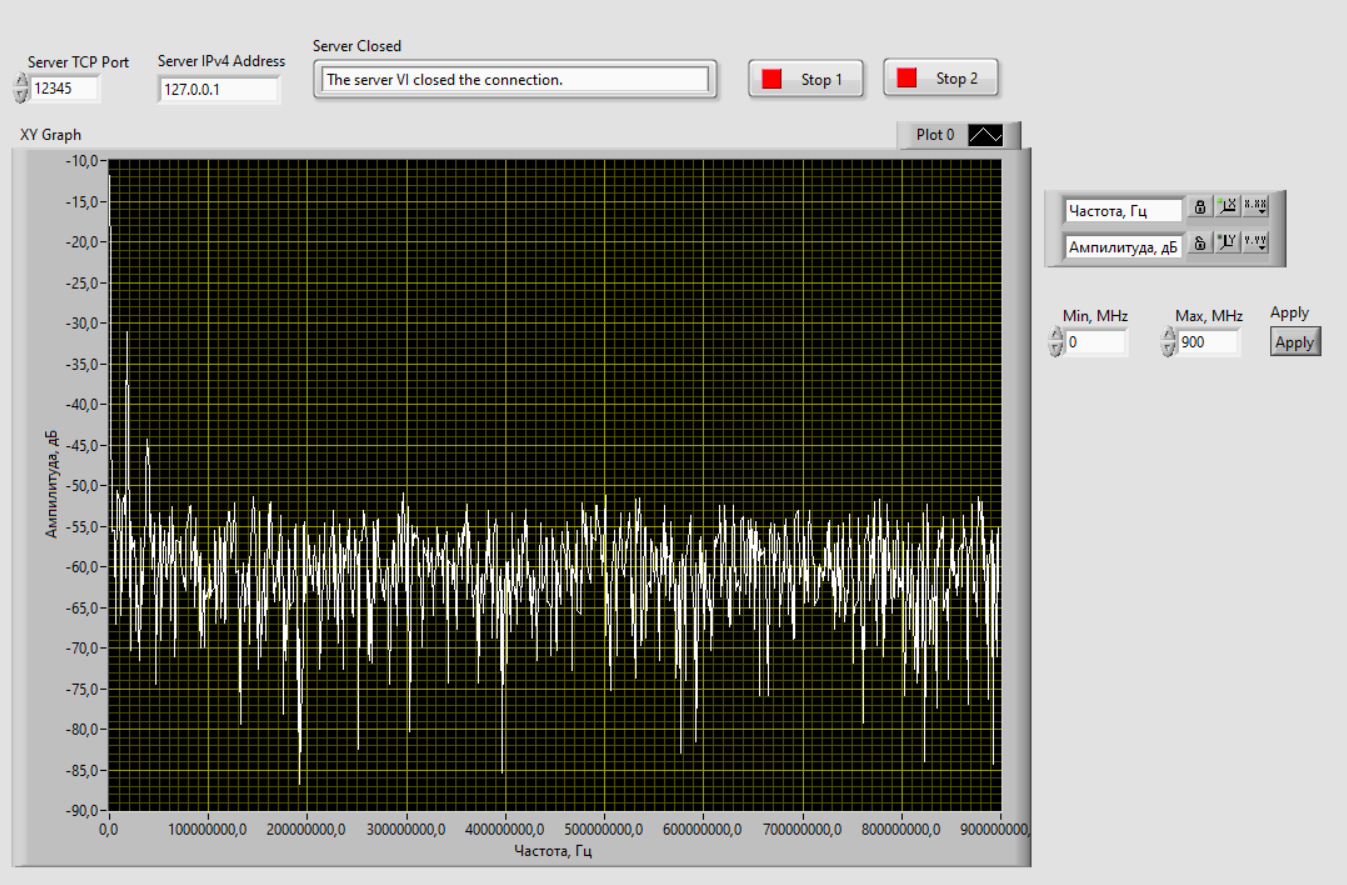


Рисунок 8 Изменение диапазона приёма на [0, 900] МГц

Накрутим антенну и сравним уровни сигнала:

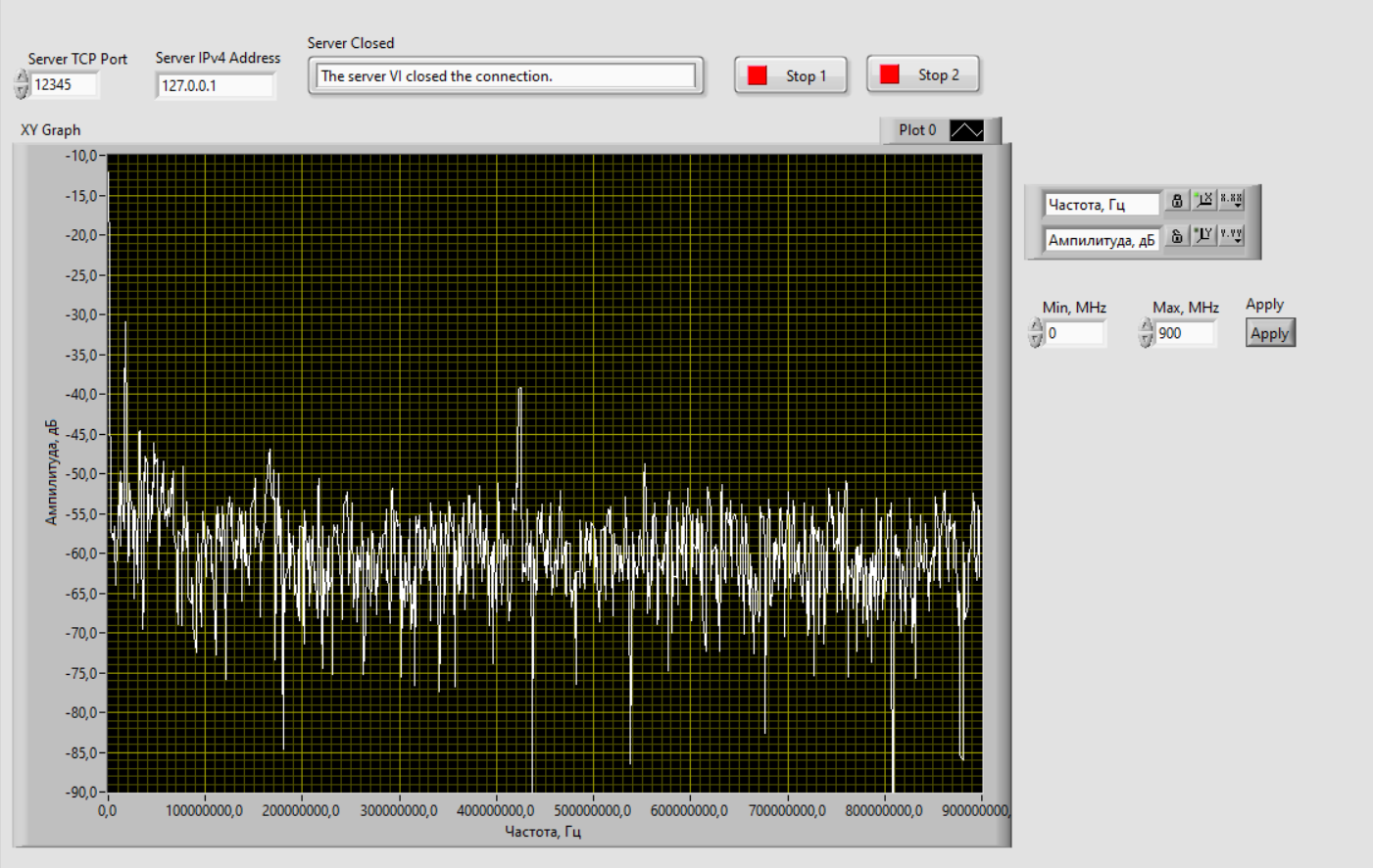


Рисунок 9 Уровни сигнала после накручивания антенны

На рисунке 9 видно, что уровни сигнала на диапазонах [400, 430] Мгц, [100, 200] МГц стали заметно сильнее.

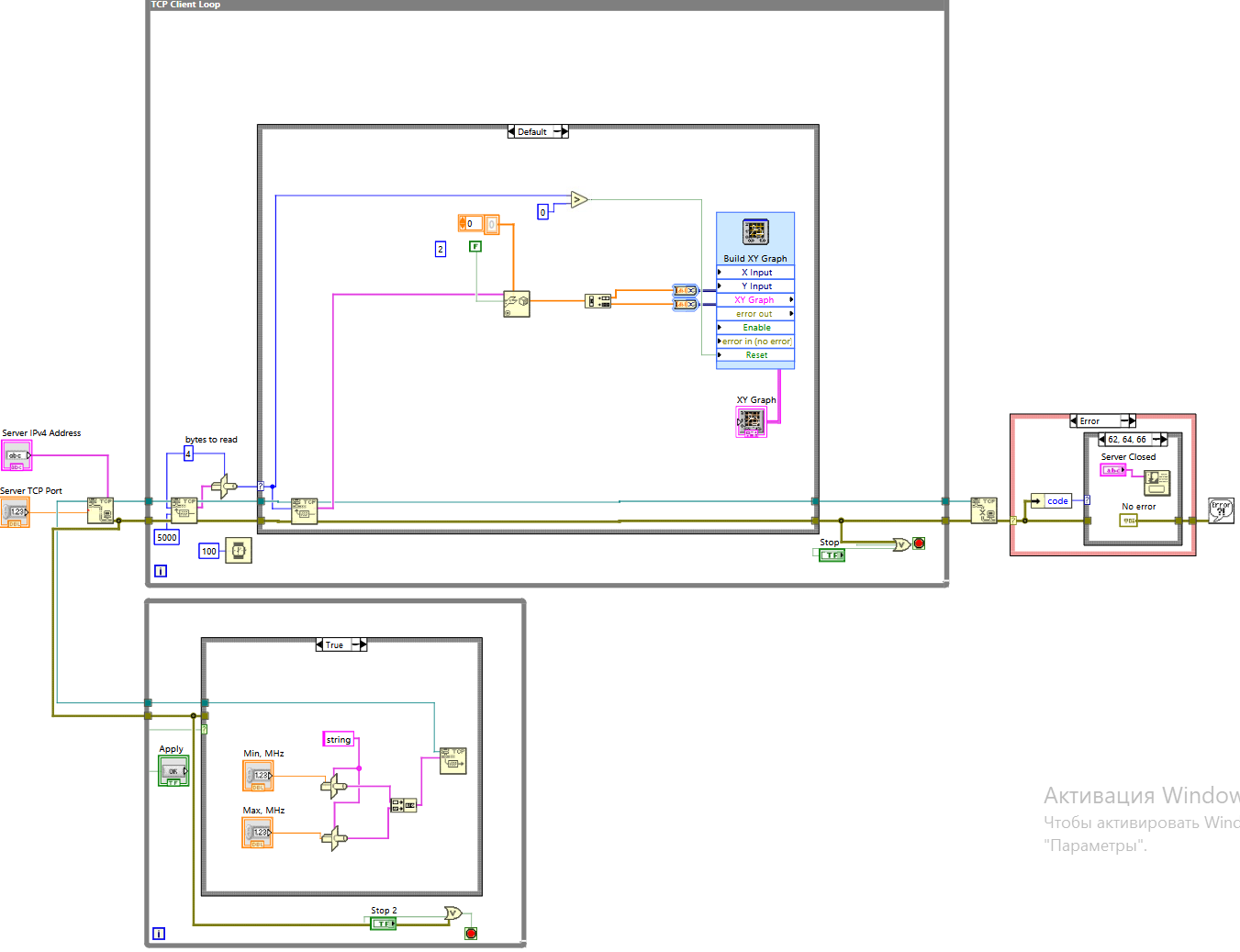
# Вывод.

В ходе работы была успешно разработан спектроанализатора на основе платы цифрового радио HackRF One и среды визуального программирования LabVIEW. Использование sweep-режима позволило эффективно анализировать широкий диапазон радиочастот и передавать данные через TCP-сервер на языке Python. Тестирование системы подтвердило её работоспособность.

# 

# Приложения.

## Блок-схема Labview.



## Python TCP-сервер.

import subprocess

import socket

import struct

import binascii

import time

from abc import ABC, abstractmethod

class DataPacker:

    @staticmethod

    def pack\_data(data):

        packed\_data = struct.pack(f'>{len(data)}d', \*data)

        packed\_size = struct.pack('>i', len(packed\_data))

        return packed\_size + packed\_data

class Server:

    def \_\_init\_\_(self, host, port):

        self.host = host

        self.port = port

        self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

        self.server\_socket.bind((self.host, self.port))

        self.server\_socket.listen(1)

        self.client\_socket = None

        self.client\_addr = None

    def start(self):

        print(f"Server listening on {self.host}:{self.port}")

        while True:

            print("Waiting for a connection...")

            self.client\_socket, self.client\_addr = self.server\_socket.accept()

            print(f"Connection from {self.client\_addr}")

            self.handle\_client()

    def handle\_client(self):

        try:

            while True:

                # Приём данных от клиента

                data = self.client\_socket.recv(16)

                if not data:

                    print('no data... continue...')

                    continue

                # Распаковка данных

                #packed\_size = struct.unpack('>i', data[:4])[0]

                packed\_size = 16

                packed\_data = data

                new\_ranges = struct.unpack(f'>{packed\_size // 8}d', packed\_data)

                # Отображение данных в командной строке

                print(f"Received data: {new\_ranges}")

                # Проверка и перезапуск парсера, если диапазоны отличаются

                if new\_ranges != self.parser.current\_ranges:

                    self.parser.current\_ranges = new\_ranges

                    self.parser.restart\_parser()

        except (ConnectionResetError, BrokenPipeError):

            print(f"Connection from {self.client\_addr} was closed.")

        finally:

            self.client\_socket.close()

    def send\_data(self, data):

        if self.client\_socket:

            self.client\_socket.sendall(data)

class HackrfSweepParser:

    def \_\_init\_\_(self, server):

        self.current\_buffer = []

        self.server = server

        self.current\_ranges = (0, 6000)

        self.process = None

        self.parser\_thread = None

    def buffer\_to\_packed\_points(self, buffer):

        result = []

        sorted\_buffer = sorted(buffer, key=lambda x: float(x[1]))

        for line in sorted\_buffer:

            \_, hz\_low, hz\_high, hz\_bin\_width, dbs = line

            for i in range(0, 5):

                result.append(float((int(hz\_low) + int(hz\_low) + float(hz\_bin\_width) \* (i + 1)) / 2)) #x

                result.append(float(dbs[i])) #y

        return result

    def parse\_hackrf\_sweep(self):

        # Команда для запуска hackrf\_sweep

        command = ["hackrf\_sweep", "-f", f"{int(self.current\_ranges[0])}:{int(self.current\_ranges[1])}"]

        try:

            # Запуск процесса

            self.process = subprocess.Popen(

                command,

                stdout=subprocess.PIPE,

                stderr=subprocess.PIPE,

                text=True

            )

            print("Запущено hackrf\_sweep. Ожидание данных...")

            # Обработка вывода в реальном времени

            previous\_date\_time = None

            for line in self.process.stdout:

                # Удаляем лишние пробелы и проверяем, содержит ли строка данные

                line = line.strip()

                if "," in line:

                    # Разделяем строку на части

                    fields = line.split(", ")

                    if len(fields) > 6:  # Проверяем наличие необходимого количества полей

                        date\_time = fields[0] + fields[1]

                        hz\_low = fields[2]

                        hz\_high = fields[3]

                        hz\_bin\_width = fields[4]

                        dbs = [fields[6 + i] for i in range(5)]

                        if int(hz\_low) == int(self.current\_ranges[0])\*1e6 and len(self.current\_buffer) > 0:

                            result = self.buffer\_to\_packed\_points(self.current\_buffer)

                            data\_to\_send = DataPacker.pack\_data(result)

                            self.server.send\_data(data\_to\_send)

                            self.current\_buffer = []

                        self.current\_buffer.append((date\_time, hz\_low, hz\_high, hz\_bin\_width, dbs))

                        previous\_date\_time = date\_time

            print("Данные успешно сохранены в hackrf\_sweep\_output.csv.")

        except KeyboardInterrupt:

            print("Остановка выполнения по Ctrl+C.")

        except Exception as e:

            print(f"Произошла ошибка: {e}")

        finally:

            try:

                if self.process:

                    self.process.terminate()

            except Exception:

                pass

    def restart\_parser(self):

        # Остановка текущего процесса

        if self.process:

            self.process.terminate()

        # Перезапуск парсера в отдельном потоке

        if self.parser\_thread and self.parser\_thread.is\_alive():

            self.parser\_thread.join()

        self.parser\_thread = threading.Thread(target=self.parse\_hackrf\_sweep)

        self.parser\_thread.start()

if \_\_name\_\_ == "\_\_main\_\_":

    host = '127.0.0.1'

    port = 12345

    server = Server(host, port)

    parser = HackrfSweepParser(server)

    server.parser = parser  # Добавление ссылки на парсер в сервер

    # Start the server in a separate thread

    import threading

    server\_thread = threading.Thread(target=server.start)

    server\_thread.start()

    # Start parsing hackrf\_sweep data

    parser.parse\_hackrf\_sweep()