

Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации
федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

09.03.01 Информатика и вычислительная техника
(направление подготовки/специальность)
Программное обеспечение мобильных систем
(профиль/специализация)
Очная
(форма обучения)

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ
(вид практики)

Тип практики

Технологическая (проектно-технологическая) практика на предприятии ООО «Бюро 1440»
(наименование профильной организации/структурного подразделения СибГУТИ)

ТЕМА ИНДИВИДУАЛЬНОГО ЗАДАНИЯ
TODO

Выполнил:

студент института информатики и вычислительной техники
Гмыря Ярослав Александрович
группа ИА-331

«____» _____ 202__г.

_____ / Гмыря Я.А. /
(подпись) (ФИО)

Проверил¹

Руководитель практики от профильной
организации

«____» _____ 202__г.

_____ / Андреев А. В. /
(подпись) (ФИО)

Проверил:

Руководитель практики от СибГУТИ

«____» _____ 202__г.

_____ / Брагин К. И. /
(подпись) (ФИО)

отметка ² _____

«____» _____ 202__г.

Новосибирск 2025

¹ В случае прохождения практики в профильной организации

² Заполняется во время промежуточной аттестации

**План-график проведения
Производственной практики**
вид практики
Гмыря Ярослав Александрович
Фамилия Имя Отчество студента

института ИВТ, курса 3, гр. ИА-331

Направление: 09.03.01 Информатика и вычислительная техника

Код – Наименование направления (специальности)

Направленность (профиль)/ специализация: Программное обеспечение мобильных систем

Место прохождения практики: г. Новосибирск, ул. Бориса Богаткова, д. 51, ауд. 314

Объем практики: 360/10 часов/ЗЕ

Тип практики: Технологическая (проектно-технологическая) практика

Срок практики: с 16.09.2025 по 26.05.2025 (раз в неделю);

Содержание практики³:

Тема индивидуального задания практики TODO

Наименование видов деятельности	Дата (начало – окончание)
Архитектура Adalm Pluto SDR. GNU Radio. Построение радио-приёмника.	16.09.2025
Введение в архитектуру SDR-устройств. Знакомство с библиотеками Soapy SDR, Libio для работы с Adalm Pluto SDR. Инициализация SDR-устройства. Работа с буфером: получение цифровых IQ-отсчетов.	23.09.2025
Принципы работы библиотеки Soapy SDR и работы с Adalm Pluto. Работа с библиотеками Soapy SDR, Libio . Формирование и передача с SDR сигналов произвольной формы.	30.09.2025
Архитектура SDR-устройств. Продолжение. Примеры формирования I/Q-сэмплов произвольной формы. Работа с буфером приема SDR.	7.10.2025
Прием сигналов с фазовой модуляцией BPSK/QPSK. Имитация аналоговой передачи звука и его прием с использованием SDR. Анализ влияния чувствительности приемника и усиления передатчика на качество принятых отсчетов сигнала (сэмплов).	14.10.2025
Имитация аналоговой передачи звука и его прием с использованием SDR. Анализ влияния чувствительности приемника и усиления передатчика на качество принятых отсчетов сигнала (сэмплов) - продолжение	21.10.2025

В соответствии с рабочей программой практики

Руководитель практики от профильной

организации*

« » 2025 г.

_____ / Андреев А. В. /

(подпись)

(ФИО)

Руководитель практики от СибГУТИ

« » 2025 г.

_____ / Брагин К. И. /

(подпись)

(ФИО)

³ В случае прохождения практики в профильной организации.

Отзыв о работе студента

(ФИО студента)

Уровень освоения компетенций

Компетенции	Уровень сформированности компетенций
ПК-1 Способен разрабатывать требования и проектировать программное обеспечение	
ПК-3 Способен осуществлять эксплуатацию и развитие транспортных сетей и сетей передачи данных, включая спутниковые системы	

Уровень компетенций: высокий, средний, низкий, не сформирована

Руководитель практики от СибГУТИ:

Старший преподаватель

Кафедры ТС и ВС

должность руководителя практики
практики

подпись

Брагин К.И.

ФИО руководителя

Оглавление

Практика 1.....	5
Практика 2.....	27
Практика 3.....	40
Практика 4.....	53
Практика 5-6.....	73

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»

Кафедра телекоммуникационных систем и вычислительных средств
(ТС и ВС)

Отчет по производственной практике
по дисциплине
SDR

по теме:
АРХИТЕКТУРА ADALM PLUTO SDR. GNU RADIO. ПОСТРОЕНИЕ
РАДИО-ПРИЁМНИКА

Студент:
Группа ИА-331

Я.А Гмыря

Предподаватели:
Лектор
Семинарист
Семинарист

Калачиков А.А
Ахпашев А.В
Попович И.А

Новосибирск 2025 г.

СОДЕРЖАНИЕ

1 ЦЕЛЬ И ЗАДАЧИ	3
2 ЛЕКЦИЯ	4
3 ПРАКТИЧЕСКАЯ ЧАСТЬ	10
4 ВЫВОД	22

ЦЕЛЬ И ЗАДАЧИ

Цель: узнать, что такое SDR, изучить принципы его работы и внутреннюю архитектуру на базовом уровне. Познакомиться с инструментом GNU Radio и создать с его помощью программу для SDR, позволяющую принимать радио.

Задачи:

1. Прослушать и законспектировать лекцию, познакомиться с основами SDR-систем.
2. На основе полученных знаний создать в GNU Radio программу для SDR, позволяющую принимать радио.

ЛЕКЦИЯ

Что такое SDR?

Software-Defined Radio (SDR) - радиосистема, в которой часть аппаратных компонентов (фильтры, модуляторы и т.п.) реализованы на программном уровне.

Базовая архитектура системы радиосвязи

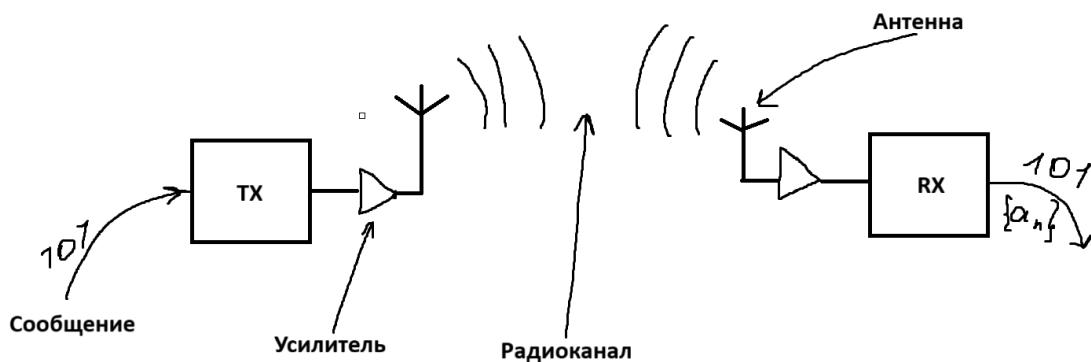


Рисунок 1 — Архитектура системы радиосвязи в целом

Описание компонентов архитектуры

Базовая архитектура состоит из **передатчика (TX)** и **приемника (RX)**. Между ними находится **радиоканал - среда**, в которой распространяется сигнал. У **TX** и **RX** есть **антенна - устройство**, которое излучает или принимает электромагнитные волны, и преобразует их в электрический ток и обратно, в самом простом случае это просто кусок проволоки. Также и у **TX** и у **RX** есть **усилитель**, который усиливает отправляемый/принимаемый сигнал.

Описание процесса обмена данными

На стороне **TX** формируется **сообщение**, которое необходимо передать. Это сообщение поступает в передатчик в виде набора **нулей и единиц**.

TX преобразует нули и единицы определенным образом в электрические колебания, которые через **антенну** излучаются в виде электромагнитных колебаний в радиоканал.

В этом же радиоканале находится **приемник**, **антенна** которого принимает эти электромагнитные колебания и преобразует в электрический ток. После этого электрические колебания определенным образом преобразуются в набор нулей и единиц (сообщение, которое отправлял **TX**). Стоит отметить, что **прием сообщения** намного сложнее, чем отправка. Это связано с изменениями, которым подвергается сигнал во время прохождения через **радиоканал**. Сигнал изменяется случайным образом, поэтому точно сказать, как изменится сигнал, мы не можем, мы можем это только предположить с какой-то точностью. Эта проблема решается путем добавления в исходный сигнал **избыточности**, которая позволяет с более высокой точностью принять сигнал на стороне приемника. Такой избыточностью может быть **контрольная сумма - число**, которое вычисляется по определенному алгоритму, который учитывает позицию бита и его значение, т.е если хоть в какой нибудь позиции изменится значение бита, то **контрольная сумма** будет уже другой, что сигнализирует об искажении сигнала.

Внутренняя архитектура TX

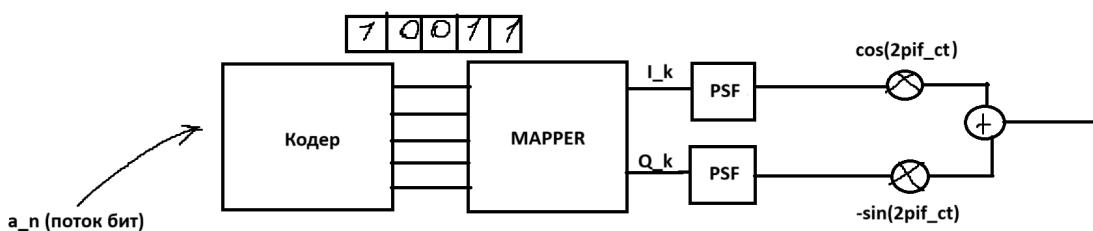


Рисунок 2 — TX архитектура

Coder

В нашем случае кодер будет выполнять единственную задачу - формировать из потока бит блоки (допустим по 8 бит) и направлять их в Mapper.

Mapper

Устройство, выполняющее **отображение исходных данных на множество символов или сигналов** в соответствии с выбранной схемой модуляции.

Символ - элемент сигнального множества.

Сигнальное множество - набор состояний радиосигнала.

Иными словами: mapper берет блок битов (у нас это 8 бит) и сопоставляет его сигналу с определенными характеристиками, для этого в mapper хранится таблица с комбинациями битов и соответствующие им символы. Если в блоке 8 бит, то всего должно быть 256 символов (на каждую возможную комбинацию).

b0	b1	state
0	0	state1
0	1	state2
1	0	state3
1	1	state4

Рисунок 3 — Пример таблицы в маппере

Также для визуализации данного процесса используется созвездие символов

Созвездие символов - это **графическое представление множества возможных символов модуляции** в комплексной плоскости. Каждый символ соответствует определённой комбинации параметров сигнала и изображается в виде точки на диаграмме.

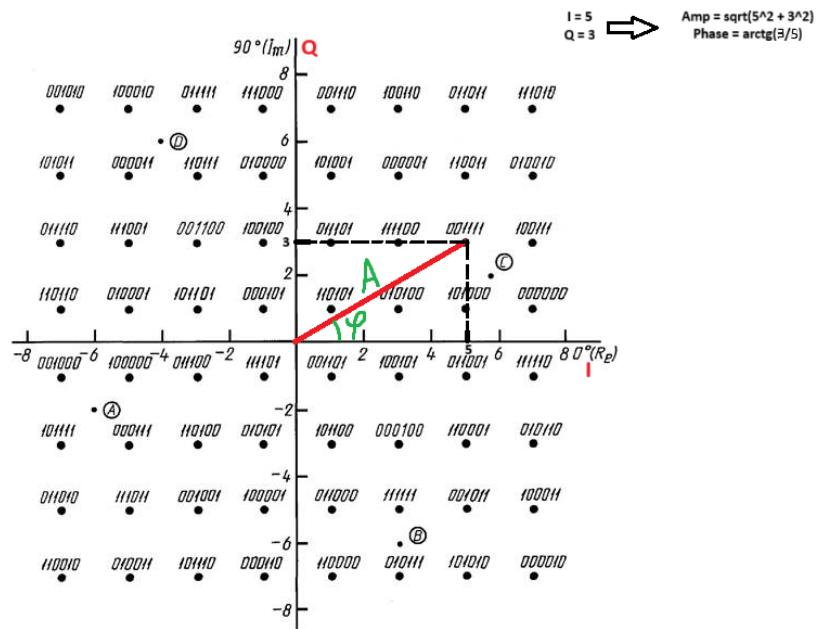


Рисунок 4 — Пример созвездия символов

Каждой точке соответствуют координаты (I, Q), где I - действительная составляющая, Q - мнимая. Зная координаты, можем вычислить длину радиус-вектора до этой точки, это будет амплитудой этого сигнала, угол между действительной осью и радиус-вектором - фаза сигнала.

От mapper идет 2 выхода, один для I составляющей, другой для Q составляющей, которые поступают на вход формирующего фильтра.

Формирующий фильтр

Устройство, преобразующее последовательности символов в непрерывный сигнал с заданной формой. Этот фильтр должен превратить символы (I и Q) в длительные (передаваемые) символы (символы, растянутые по времени с длиной T_s)

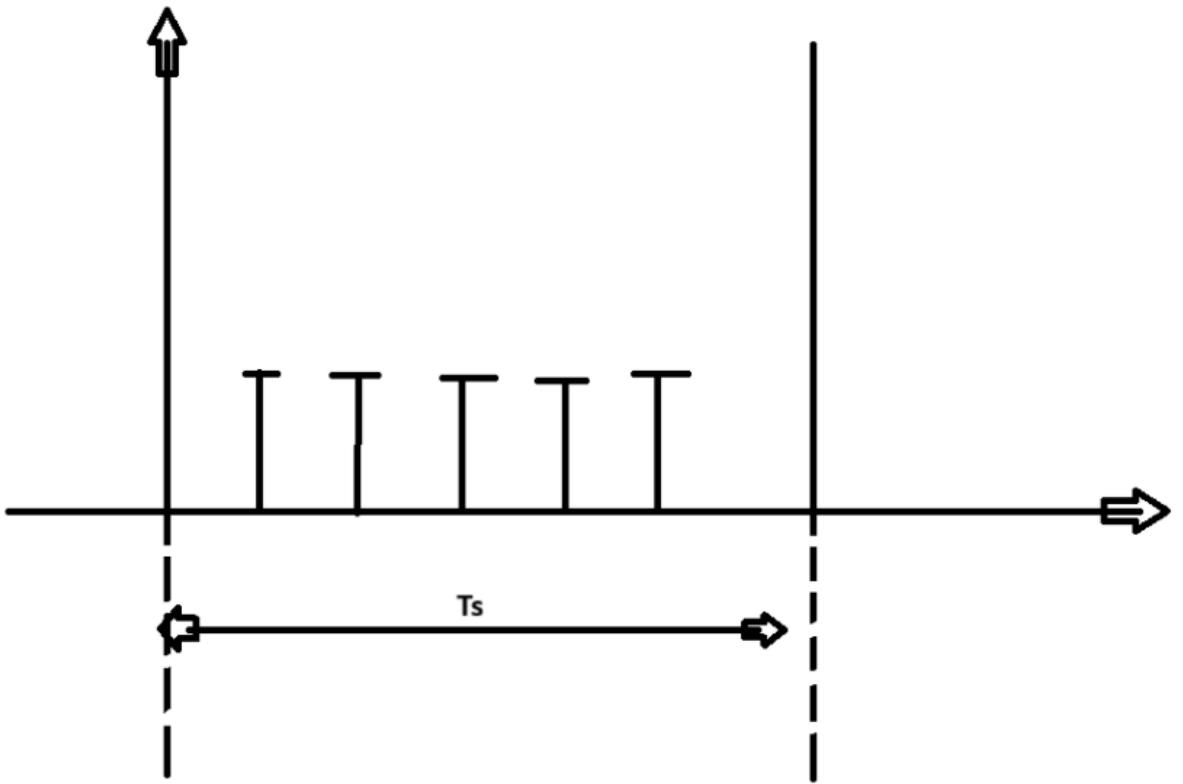


Рисунок 5 — Пример длительного символа

До этого момента все выполнялось программно. Всё, что будет дальше - работа самой SDR.

Далее происходит генерация непрерывного сигнала. Математически этот процесс можно записать следующим образом:

$$s(t) = I(t) \cos(2\pi f_c t) - Q(t) \sin(2\pi f_c t)$$

f_c здесь - несущая частота (высокочастотное колебание).

ПРАКТИЧЕСКАЯ ЧАСТЬ

Adalm Pluto SDR

Adalm Pluto SDR - модель SDR, разработанная компанией **Analog Devices** для обучения основам SDR.

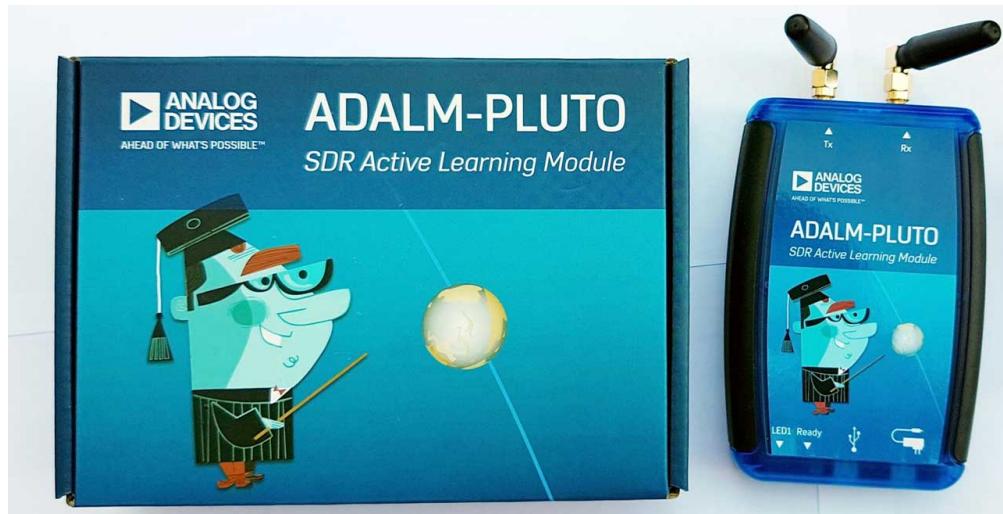


Рисунок 6 — Внешний вид Adalm Pluto

Архитектура Adalm Pluto SDR

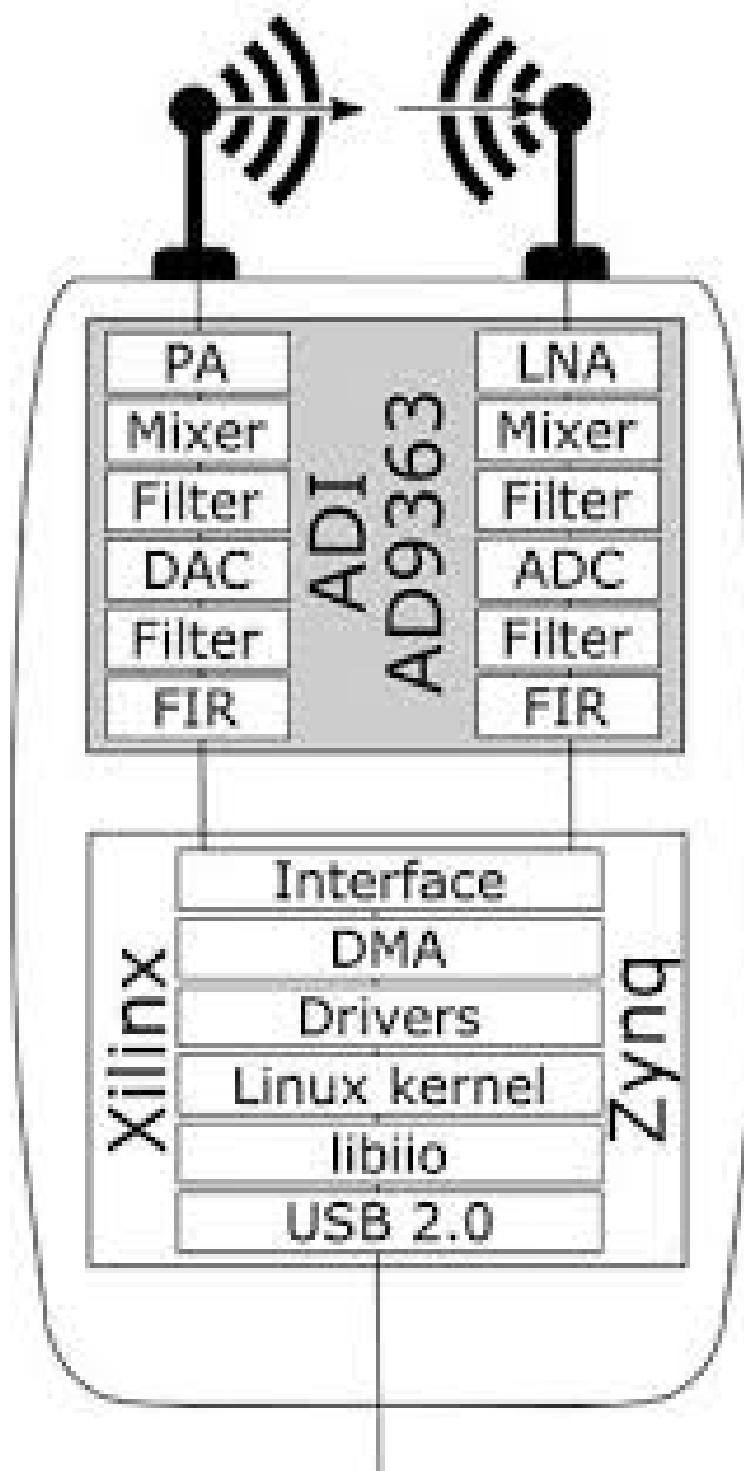


Рисунок 7 — Архитектура Adalm Pluto

Описание основных блоков Adalm Pluto

PA (Power Amplifier)

Функция: Усиление сигнала.

LNA (Low Noise Amplifier)

Функция: Усиление слабого приёмного сигнала с минимальным добавлением шума.

ADC / DAC

ADC (Analog-to-Digital Conversion): оцифровка аналогового сигнала.

DAC (Digital-to-Analog Conversion): преобразование цифровых семплов в аналоговый сигнал.

FIR (Finite Impulse Response)

Функция: детальная фильтрация, коррекция формы спектра.

Mixer

TX: перенос низкочастотного сигнала на несущую частоту.

RX: перенос высокочастотного сигнала на низкочастотный.

Filter

TX: фильтрация выходного сигнала перед передачей.

RX: фильтрация принимаемого сигнала.

libii0

Описание: библиотека, которая облегчает работу с устройствами ввода/вывода в Linux. Она даёт API для обмена данными и управления устройствами.

Linux Kernel

Описание: ядро Linux. Управляет железом.

Drivers

Описание: обеспечение корректной работы Linux с устройствами.

Xilinx Zynq

Описание: семейство микросхем от компании Xilinx. На одном кристалле объединены ARM-процессор, на котором работает Linux, и ПЛИС для более скоростных вычислений.

GNU Radio



Рисунок 8 — Логотип GNU Radio

GNU Radio - это инструмент с открытым исходным кодом для разработки программного обеспечения в сфере программно-определенного радио.

Он позволяет при помощи «строительных блоков» создавать конфигурации радиоустройств, не написав ни одной строчки кода, и запускать программы непосредственно с использованием SDR-модулей, например:

Adalm-Pluto, LimeSDR и др.

В библиотеке имеется широкий спектр функций для цифровой обработки сигналов. Модули написаны на **C++**, а их взаимодействие реализовано на **Python**. Приложения можно строить как через **API GNU Radio**, так и посредством графического интерфейса **GNU Radio Companion (GRC)**.

Построение схемы в **GNU Radio**

Блок **options**

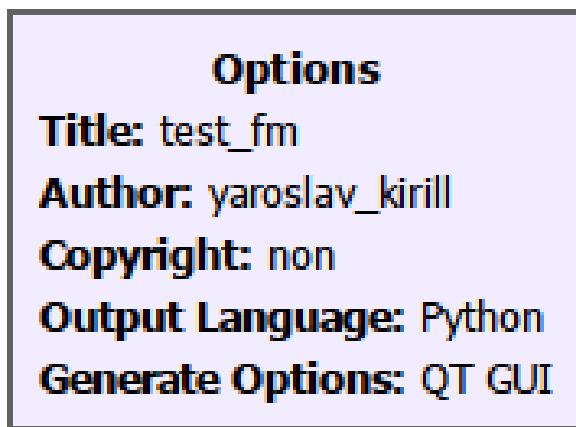


Рисунок 9 — Блок **options**

Этот блок задает настройки проекта. Самое важное здесь: **Output Language** и **Generate Options**.

Output Language — язык, на котором будет сгенерирован код программы.

Generate Options — используемый графический интерфейс.

Блок variable



Рисунок 10 — Блок variable

В этом блоке можно задать переменную (почти как в языке программирования). Переменная имеет ID (имя) и значение. Здесь задается samp_rate (частота дискретизации), равная 2.4×10^6 Hz.

QT GUI Range

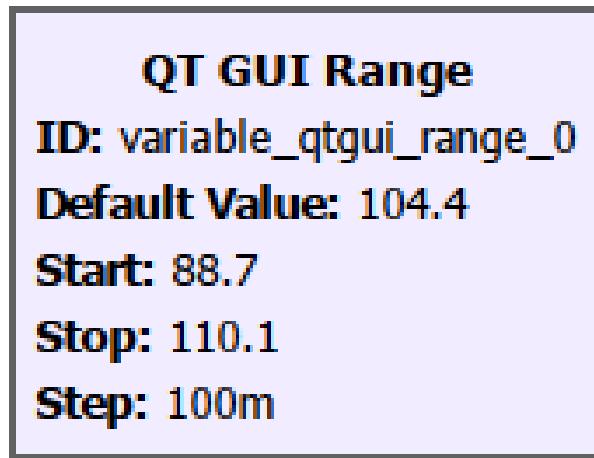


Рисунок 11 — Блок QT GUI Range

Этот блок задает ползунок из QT, позволяющий удобно менять значение переменной во время работы программы. Это позволяет не перезапускать программу, когда нам требуется поменять какое-либо значение. Здесь задается ползунок для настройки частоты приема FM волны.

Основные параметры блока:

- **Default Value** — значение, которое будет устанавливаться при запуске программы;
- **Start** — минимальное значение;
- **Stop** — максимальное значение;
- **Step** — шаг изменения при сдвиге ползунка.

PlutoSDR Source

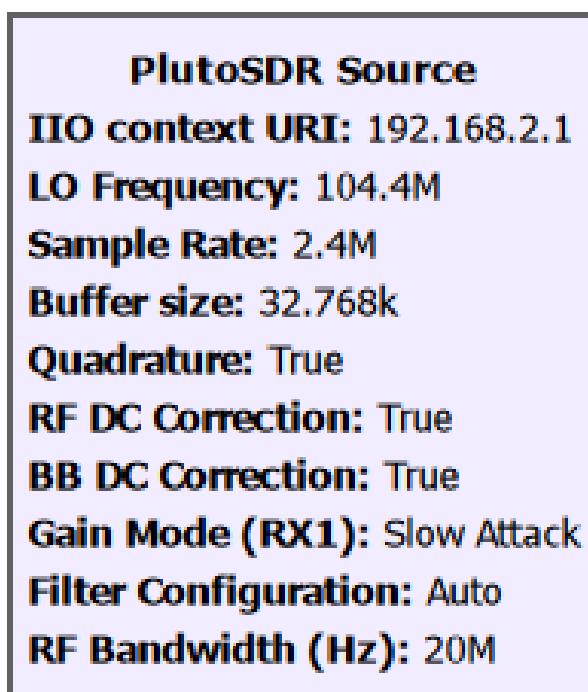


Рисунок 12 — Блок PlutoSDR Source

Этот блок отвечает за приём данных от устройства ADALM-Pluto (PlutoSDR). Он подключается к SDR, управляет настройками, получает поток отсчётов.

Параметры:

- **IIO context URI** — IP адрес Adalm Pluto, нужен, потому что PlutoSDR может подключаться по USB или сети (Ethernet/USB-Ethernet);

- **Sample Rate** — частота дискретизации АЦП внутри PlutoSDR. Определяет, с какой частотой будут делаться отсчеты при оцифровке;
- **Buffer Size** — встроенный буфер для временного хранения данных перед их передачей в компьютер;
- **Quadrature** — задаём представление сигнала в виде I/Q семплов;

Low Pass Filter



Рисунок 13 — Блок Low Pass Filter

Ограничивает полосу сигнала, выделяя только FM-станцию.

Параметры:

- **Decimation** — снижение частоты дискретизации в n раз;
- **Gain** — усиление амплитуды после фильтрации;
- **Sample Rate** — дефолтная частота дискретизации;
- **Cutoff Freq** — полоса 100 кГц (ширина FM сигнала).

QT GUI Frequency Sink

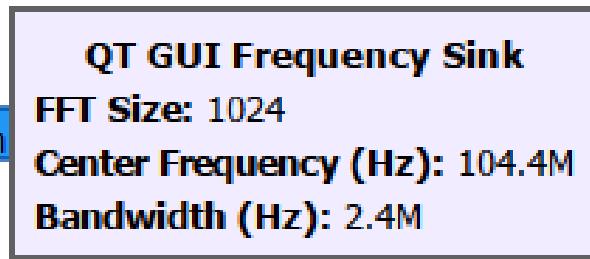


Рисунок 14 — Блок QT GUI Frequency Sink

Этот блок при помощи QT задает спектральное представление сигнала, которое меняется в реальном времени. Таких блоков 2: до фильтра (напрямую из блока source) и после фильтра. Первый показывает весь эфир, а второй — захваченный сигнал (именно FM частоту).

Параметры:

- **FFT Size** — кол-во точек для спектра;
- **Center Frequency** — центральная частота захвата;
- **Bandwidth** — полоса частот захвата.

QT GUI Time Sink

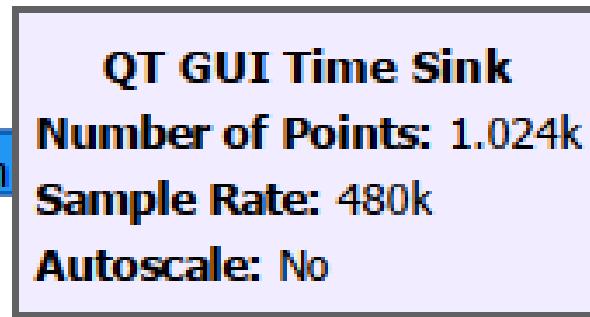


Рисунок 15 — Блок QT GUI Time Sink

Этот блок при помощи QT задает временное представление сигнала, которое меняется в реальном времени.

Параметры:

- **Number of Points** — кол-во точек, отображаемых в каждый момент времени;
- **Sample Rate** — частота дискретизации при отрисовке;
- **Autoscale** — нужно ли масштабировать сигнал по вертикали.

WBFM Receive



Рисунок 16 — Блок WBFM Receive

Блок демодуляции FM-сигнала.

Параметры:

- **Quadrature Rate** — входная частота дискретизации (после фильтра и децимации);
- **Audio Decimation** — уменьшение дискретизации для звука (в моем случае до 48k, чего вполне достаточно для звука).

На выходе — звуковой сигнал.

Audio Sink



Рисунок 17 — Блок Audio Sink

От **WBFM Receive** звук идет на блок **Audio Sink** — блок, который выводит звуковой поток на аудиокарту хоста.

Параметры:

- **Sample Rate** — стандартная частота звука.

Соединить блоки нужно следующим образом, тогда у нас получится работающая радиосистема, которая будет принимать FM радио.

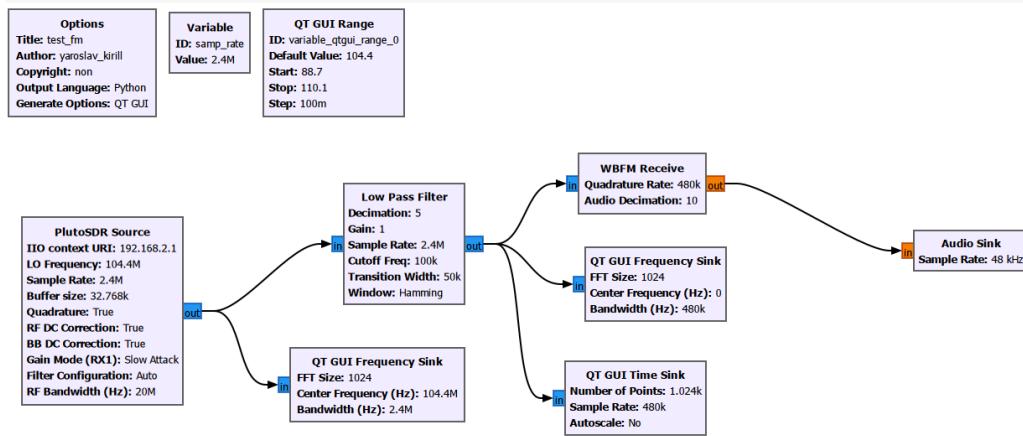


Рисунок 18 — Пример простой радиосистемы в GNURadio

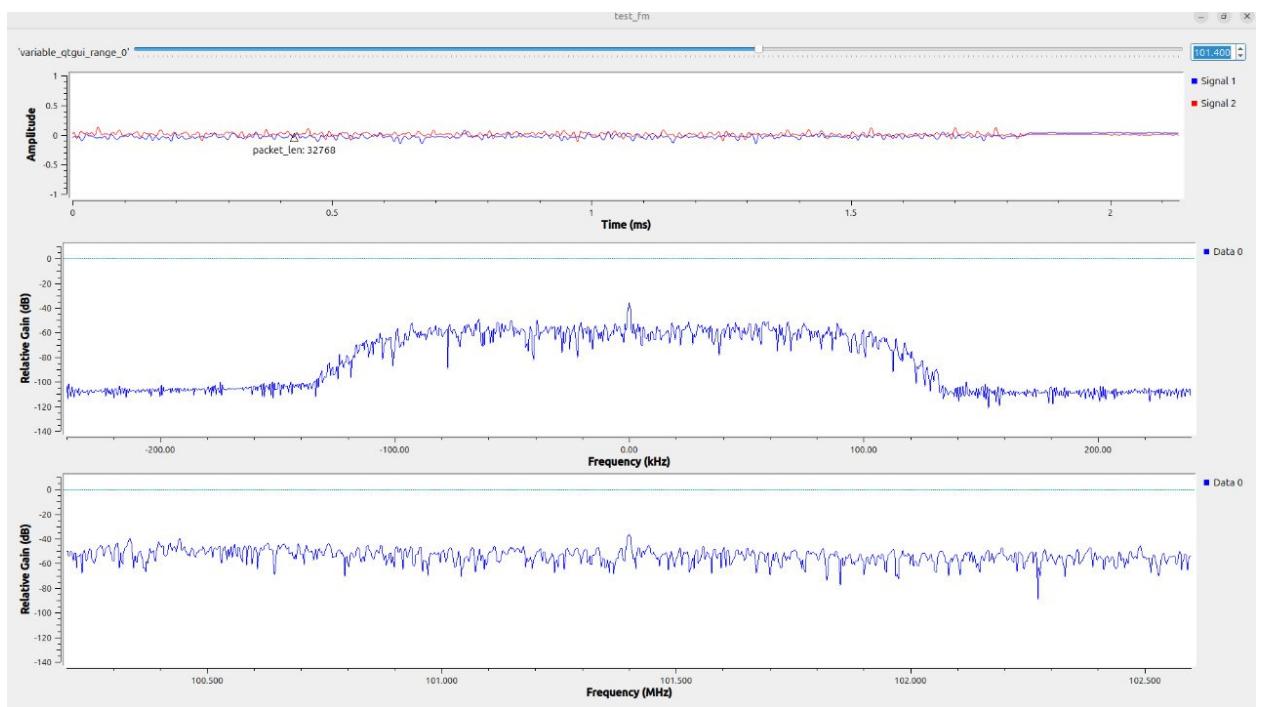


Рисунок 19 — Пример работы программы

ВЫВОД

В ходе проделанной работы я узнал, что такое SDR, изучил принципы его работы и внутреннюю архитектуру на базовом уровне. Познакомился с инструментом GNU Radio и создал с его помощью программу для SDR, позволяющую принимать FM радио.

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»

Кафедра телекоммуникационных систем и вычислительных средств
(ТС и ВС)

Отчет по производственной практике
по дисциплине
SDR

по теме:

ВВЕДЕНИЕ В АРХИТЕКТУРУ SDR-УСТРОЙСТВ. ЗНАКОМСТВО С
БИБЛИОТЕКАМИ SOAPY SDR, LIBIO ДЛЯ РАБОТЫ С ADALM PLUTO
SDR. ИНИЦИАЛИЗАЦИЯ SDR-УСТРОЙСТВА. РАБОТА С БУФЕРОМ:
ПОЛУЧЕНИЕ ЦИФРОВЫХ IQ-ОТСЧЕТОВ.

Студент:

Группа ИА-331

Я.А Гмыря

Предподаватели:

Лектор

Калачиков А.А

Семинарист

Ахпашев А.В

Семинарист

Попович И.А

Новосибирск 2025 г.

СОДЕРЖАНИЕ

1 ЦЕЛЬ И ЗАДАЧИ	3
2 ЛЕКЦИЯ	4
3 ПРАКТИЧЕСКАЯ ЧАСТЬ	7
4 ВЫВОД	13

ЦЕЛЬ И ЗАДАЧИ

Цель: Познакомиться с упрощенной архитектурой приемника. Научиться работать с SDR напрямую из C++. Отправить в эфир семплы, а потом принять их и проанализировать.

Задачи:

1. Прослушать и законспектировать лекцию, познакомиться с упрощенной архитектурой приемника.
2. Настроить работу ПК с SDR.
3. Поработать с SDR через C++.

ЛЕКЦИЯ

На прошлом занятии мы рассматривали архитектуру простого передатчика и то, как в нем формируется и отправляется сигнал.

На этом занятии разберем архитектуру простого приемника и то, как он принимает сигнал и преобразует его обратно в данные.

Архитектура простого приемника

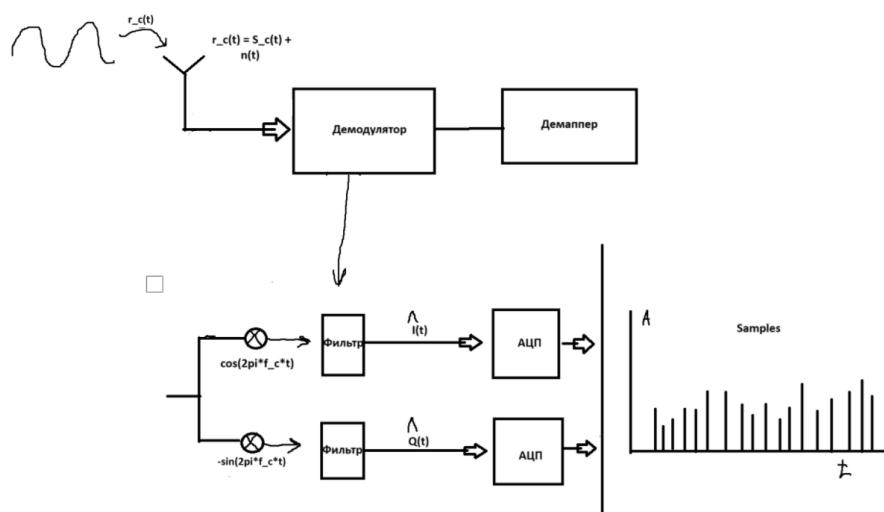


Рисунок 1 — Архитектура простого приемника

Сигнал поступает на антенну приемника и преобразуется в колебание электрического тока. Первым делом нам нужно выделить из этого высокочастотного сигнала сигнал низкой частоты, т.к с низкочастотным сигналом работать проще и вся информация содержится именно в нем. Для этого подаем сигнал на демодулятор

Демодуляция - процесс, обратный модуляции колебаний, выделение информационного (модулирующего) сигнала из модулированного колебания высокой (несущей) частоты. При передаче цифровых сигналов в результате демодуляции получается последовательность символов, передающих исходную информацию ($I(t)$ и $Q(t)$).

Принцип работы демодулятора

На схеме видим, что поступивший в демодулятор сигнал проходит через цепь, в которой он перемножается на несущие $\cos(\omega_c t)$ и $-\sin(\omega_c t)$. Но как это помогает нам узнать низкочастотный сигнал?

Вспомним из прошлого занятия, какой сигнал по итогу мы отправляли в радиоканал:

$$s(t) = I(t) \cos(2\pi f_c t) - Q(t) \sin(2\pi f_c t)$$

Рисунок 2 — Отправляемый сигнал

Вспомним тригонометрические формулы

Тема: Произведение синусов и косинусов.

$$\sin \alpha \cdot \sin \beta = \frac{1}{2} (\cos(\alpha - \beta) - \cos(\alpha + \beta))$$

$$\cos \alpha \cdot \cos \beta = \frac{1}{2} (\cos(\alpha - \beta) + \cos(\alpha + \beta))$$

$$\sin \alpha \cdot \cos \beta = \frac{1}{2} (\sin(\alpha - \beta) + \sin(\alpha + \beta))$$

Рисунок 3 — Тригонометрические формулы

Теперь произведем перемножения входного сигнала на несущие:

$$(I(t)\cos(\omega_c t) - Q(t)\sin(\omega_c t)) * \cos(\omega_c t)$$

$$= I(t)\cos(\omega_c t)\cos(\omega_c t) - Q(t)\sin(\omega_c t)\cos(\omega_c t)$$

$$\begin{aligned}
&= \frac{I(t)}{2}(\cos(2\omega_c t) + \cos(0)) - \frac{Q(t)}{2}(\sin(2\omega_c t) + \sin(0)) \\
&= \frac{I(t)}{2}\cos(2\omega_c t) - \frac{Q(t)}{2}\sin(2\omega_c t) + \boxed{\frac{I(t)}{2}}
\end{aligned}$$

У нас получилось выделить синфазную компоненту $I(t)$.

Проделаем те же действия с умножением на \sin

$$\begin{aligned}
&(I(t) \cos(\omega_c t) - Q(t) \sin(\omega_c t)) \cdot \sin(\omega_c t) \\
&= I(t) \cos(\omega_c t) \sin(\omega_c t) - Q(t) \sin^2(\omega_c t) \\
&= \frac{I(t)}{2} \sin(2\omega_c t) - \frac{Q(t)}{2}(1 - \cos(2\omega_c t)) \\
&= \frac{I(t)}{2} \sin(2\omega_c t) + \frac{Q(t)}{2} \cos(2\omega_c t) - \boxed{\frac{Q(t)}{2}}
\end{aligned}$$

Получили квадратурную компоненту $Q(t)$.

Помимо самих компонент остались и другие сигналы, которые нам не нужны, поэтому с помощью фильтра уберем их. На выходе получим чистые $\frac{I}{2}$ и $-\frac{Q}{2}$. Можно заметить, что после извлечения символов их амплитуда упала вдвое. Эта проблема решается путем усиления (на схеме не отображено)

Далее компоненты поступают на АЦП, где будут "нарезаны" на семплаы. В этих семплах нужно произвести символьную синхронизацию, чтобы правильно выделить переданную информацию, но это тема следующих занятий.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Передача данных между Adalm Pluto и хостом

Передача данных (IQ-сэмплов) между Adalm Pluto и хост-компьютером осуществляется посредством USB 2.0. Важно отметить, что в случае с SDR, данные необходимо передавать непрерывно в обе стороны (с хост-компьютера на SDR и обратно) одновременно. Хоть и теоретическая пропускная способность USB 2.0 равна 480 Mb/s, работа в полудуплексном режиме с передачей данных в обе стороны одновременно (с точки зрения пользователя) разительно снижается. Целевое значение частоты дискретизации желательно задавать в пределах 6 Msps.

Timestamping

В библиотеке SoapySDR реализованы функции получения временных меток (timestamp) с FPGA (Xilinx Zynq). Временные метки (timestamp) привязаны к каждому запросу данных с буфера ПЛИС, что, в свою очередь, позволяет синхронно получать/передавать данные в потоках RX/TX. Более того, из-за проблем с пропускной способностью USB 2.0 возникает проблема увеличения частоты десквртизации, при больших значениях которой, USB 2.0 не может обеспечить полноценную передачу и прием (одновременных) сэмплов из Adalm Pluto на хост-компьютер. Выявить данную проблему можно благодаря реализации функции временных меток с Xilinx Zynq.

Установка необходимых библиотек и зависимостей

SoapySDR

SoapySDR — открытая обобщённая API и библиотека времени выполнения для взаимодействия с SDR-устройствами. С помощью SoapySDR можно создавать экземпляры, настраивать и вести потоковую передачу данных с SDR-устройством в различных средах. Большинство готовых SDR-платформ поддерживаются SoapySDR, и многие открытые приложения используют

SoapySDR для интеграции с оборудованием. Кроме того, SoapySDR имеет привязки к средам разработки, таким как GNU Radio и Pothos.

```
sudo apt-get install python3-pip python3-setuptools
sudo apt-get install cmake g++ libpython3-dev python3-numpy swig
    python3-matplotlib

git clone --branch soapy-sdr-0.8.1
    https://github.com/TelecomDep/SoapySDR.git

cd SoapySDR
mkdir build && cd build

cmake ../

make -j 16
sudo make install
sudo ldconfig
```

Libiio

libiio — библиотека, разработанная компанией Analog Devices, которая предназначена для упрощения работы с устройствами ввода-вывода данных (I/O), особенно с программируемыми аналогово-цифровыми и цифроаналоговыми преобразователями (ADC/DAC), а также с радиооборудованием на базе платформы ADI (например, ADALM-PLUTO). Позволяет читать и записывать данные в реальном времени.

```
sudo apt-get install libxml2 libxml2-dev bison flex libcdk5-dev
    cmake

sudo apt-get install libusb-1.0-0-dev libaio-dev pkg-config
sudo apt install libavahi-common-dev libavahi-client-dev

git clone --branch v0.24 https://github.com/TelecomDep/libiio.git

cd libiio
mkdir build && cd build
cmake ../
make -j 16
sudo make install
```

LibAD9361

LibAD9361 - библиотека для работы с радиочипами семейства AD9361 от Analog Devices. В сочетании с libiio позволяет организовать потоковое чтение/запись данных в реальном времени.

```
git clone --branch v0.3
  https://github.com/TelecomDep/libad9361-iio.git
cd libad9361-iio

mkdir build && cd build

cmake ../

make -j 16
sudo make install
sudo ldconfig
```

SoapyPlutoSDR

SoapyPlutoSDR - библиотека, которая является расширением библиотеки SoapySDR, предназначенная для работы конкретно с Adalm Pluto.

```
git clone --branch sdr_gadget_timestamping
  https://github.com/TelecomDep/SoapyPlutoSDR.git
cd SoapyPlutoSDR

mkdir build && cd build

cmake ../

make -j 16
sudo make install
sudo ldconfig
```

Основные моменты работы с Adalm Pluto напрямую из C++

Подключение библиотек

```
// Init device
#include <SoapySDR/Device.h>
// Data types for writing samples
#include <SoapySDR/Formats.h>
```

Инициализация устройства

```
//create struct for init
SoapySDRKwargs args = {};

//Select device type
SoapySDRKwargs_set(&args, "driver", "plutosdr");
if (1) {
    // Sample transmission method (usb)
    SoapySDRKwargs_set(&args, "uri", "usb:");
} else {
    // Or IP
    SoapySDRKwargs_set(&args, "uri", "ip:192.168.2.1");
}
SoapySDRKwargs_set(&args, "direct", "1");
// Buffer size and timestamps
SoapySDRKwargs_set(&args, "timestamp_every", "1920");
SoapySDRKwargs_set(&args, "loopback", "0");
// Init
SoapySDRDevice *sdr = SoapySDRDevice_make(&args);
// Free memory
SoapySDRKwargs_clear(&args);
```

Формирование потоков и буферов

```
// create streams
SoapySDRStream *rxStream = SoapySDRDevice_setupStream(sdr,
    SOAPY_SDR_RX, SOAPY_SDR_CS16, channels, channel_count, NULL);
SoapySDRStream *txStream = SoapySDRDevice_setupStream(sdr,
    SOAPY_SDR_TX, SOAPY_SDR_CS16, channels, channel_count, NULL);

//start streaming
SoapySDRDevice_activateStream(sdr, rxStream, 0, 0, 0);
SoapySDRDevice_activateStream(sdr, txStream, 0, 0, 0);

// Get RX/TX MTU sizes

size_t rx_mtu = SoapySDRDevice_getStreamMTU(sdr, rxStream);
size_t tx_mtu = SoapySDRDevice_getStreamMTU(sdr, txStream);
```

```
// allocate memory for buffers (for RX/TX samples)
int16_t tx_buff[2 *tx_mtu];
int16_t rx_buffer[2 *rx_mtu];
```

Получение I/Q сэмплов

```
// start receive samples
for (size_t buffers_read = 0; buffers_read < iteration_count;
     buffers_read++)
{
    void *rx_buffs[] = {rx_buffer};
    // flags set by receive operation
    int flags;
    //timestamp for receive buffer
    long long timeNs;

    // Read samples from stream and write I/Q samples in file
    int sr = SoapySDRDeviceReadStream(sdr, rxStream, rx_buffs,
                                       rx_mtu, &flags, &timeNs, timeoutUs);
    // write in file
    for(int i = 0; i < rx_mtu * 2; i++){
        fprintf(file, "%d %d\n", rx_buffer[i], rx_buffer[i+1]);
    }
}
```

Освобождение памяти

```
//stop streaming
SoapySDRDevice_deactivateStream(sdr, rxStream, 0, 0);
SoapySDRDevice_deactivateStream(sdr, txStream, 0, 0);

//shutdown the stream
SoapySDRDevice_closeStream(sdr, rxStream);
SoapySDRDevice_closeStream(sdr, txStream);

//cleanup device handle
SoapySDRDevice_unmake(sdr);
```

После работы программы создается файл samples.txt, в котором будут храниться полученные семплы в формате: (I,Q). Для визуализации $I(t)$ и $Q(t)$ воспользуемся Python.

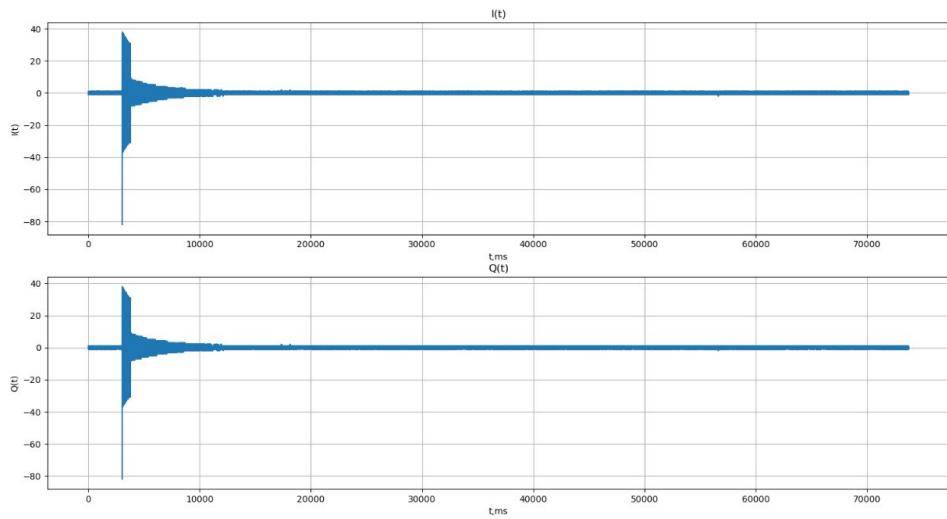


Рисунок 4 — Графики $I(t)$ и $Q(t)$

Можем наблюдать, что графики напоминают прямоугольный сигнал.

ВЫВОД

В ходе проделанной работы я познакомился с архитектурой простого приемника. Познакомился с работой с Adalm Pluto SDR напрямую из C++. Отправил, а потом принял семплы и сделал их анализ.

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»

Кафедра телекоммуникационных систем и вычислительных средств
(ТС и ВС)

Отчет по производственной практике
по дисциплине
SDR

по теме:

ПРИНЦИПЫ РАБОТЫ БИБЛИОТЕКИ SOAPY SDR И РАБОТЫ С ADALM
PLUTO. РАБОТА С БИБЛИОТЕКАМИ SOAPY SDR, LIBIO
ФОРМИРОВАНИЕ И ПЕРЕДАЧА С SDR СИГНАЛОВ ПРОИЗВОЛЬНОЙ
ФОРМЫ

Студент:

Группа ИА-331

Я.А Гмыря

Предподаватели:

Лектор

Ахпашев А.В

Семинарист

Ахпашев А.В

Семинарист

Попович И.А

Новосибирск 2025 г.

СОДЕРЖАНИЕ

1 ЦЕЛЬ И ЗАДАЧИ	3
2 ЛЕКЦИЯ	4
3 ПРАКТИЧЕСКАЯ ЧАСТЬ	8
4 ВЫВОД	13

ЦЕЛЬ И ЗАДАЧИ

Цель:

Лучше освоить библиотеку SoapySDR, сформировать собственные семплы и попытаться отправить их, а потом принять и визуализировать полученный сигнал.

Задачи:

1. Лучше разобраться в библиотеке SoapySDR
2. Сформировать свои семплы
3. Принять семплы
4. Визуализировать сигнал

ЛЕКЦИЯ

Введение

На прошлом занятии мы работали с Adalm Pluto напрямую из C++, прияли семплы и записали их в файл. На этом занятии чуть более подробно углубимся в этот процесс и попробуем сформировать свои собственные семплы и отправим их с SDR.

Структура семплов в Pluto SDR

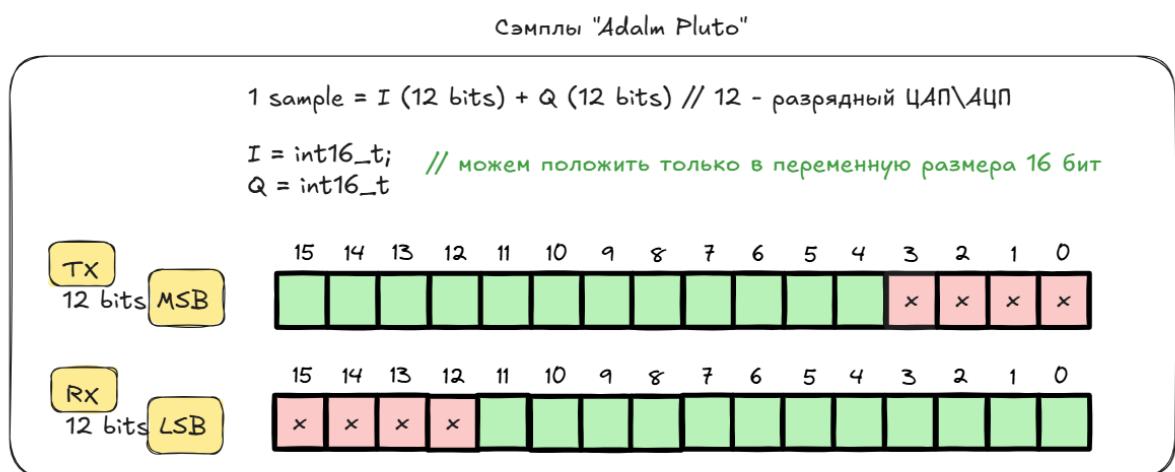


Рисунок 1 — Структура семплов

Pluto SDR имеет 12-ти битный АЦП, это значит, что для I и Q максимальное значение составляет $2^{12} = 4096$, но один бит займет знак, поэтому I и Q будут принимать значения из отреза [-2048;2047]. Для хранения I или Q в C++ используется тип данных `int16_t` (если брать меньше, то семпл не поместится). Таким образом один семпл занимает 4 байта памяти. Также есть вариант хранить семплы в одной переменной `int32_t`, но в таком случае для получения доступа к I или Q придется использовать битовые сдвиги или накладывать маски.

Стоит отметить, что Pluto SDR странно работает с TX семплами (которые хотим передавать) и интерпритирует как семплы последние 12 бит пере-

менной (если начинать отсчет от младшего бита), поэтому необходимо сдвигать значение I и Q на 4 бита влево («4») при работе с tx буффером, чтобы Pluto SDR корректно их интерпритировал.

Структура буфера семплов в Pluto SDR

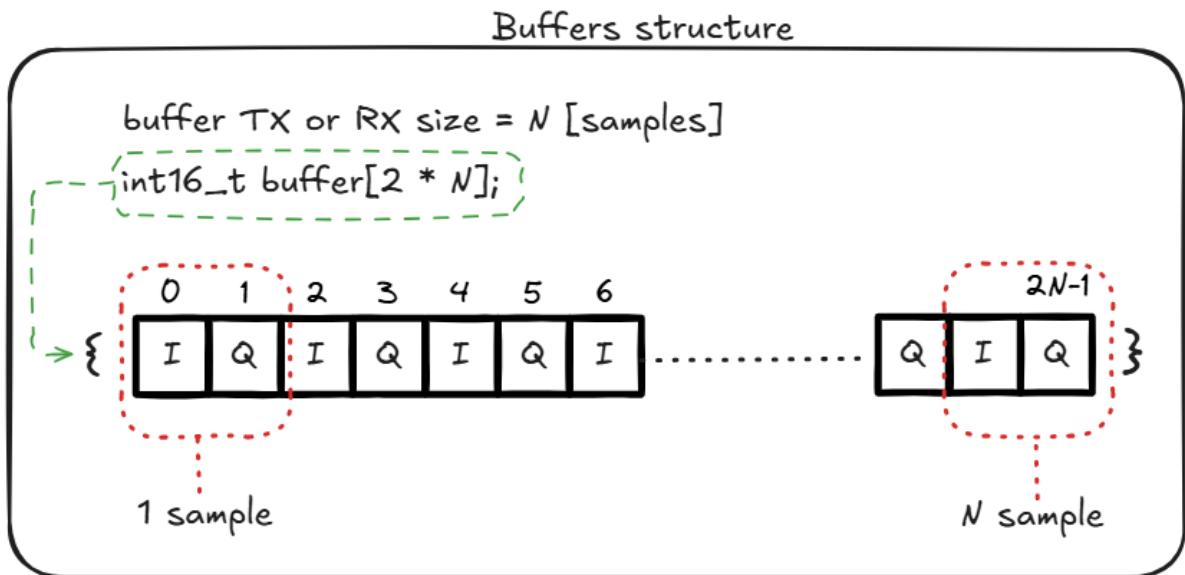


Рисунок 2 — Структура буфера

Семпл состоит из компонент I и Q, которые хранятся последовательно друг за другом, т.е в буфере будет иметь последовательность вида $I_0, Q_0, I_1, Q_1, \dots, I_n, Q_n$, поэтому если мы хотим принять/передать N семплов, то буффер должен быть размером $2N$, т.к семпл состоит из двух чисел.

Запись RX семплов в буффер в Pluto SDR

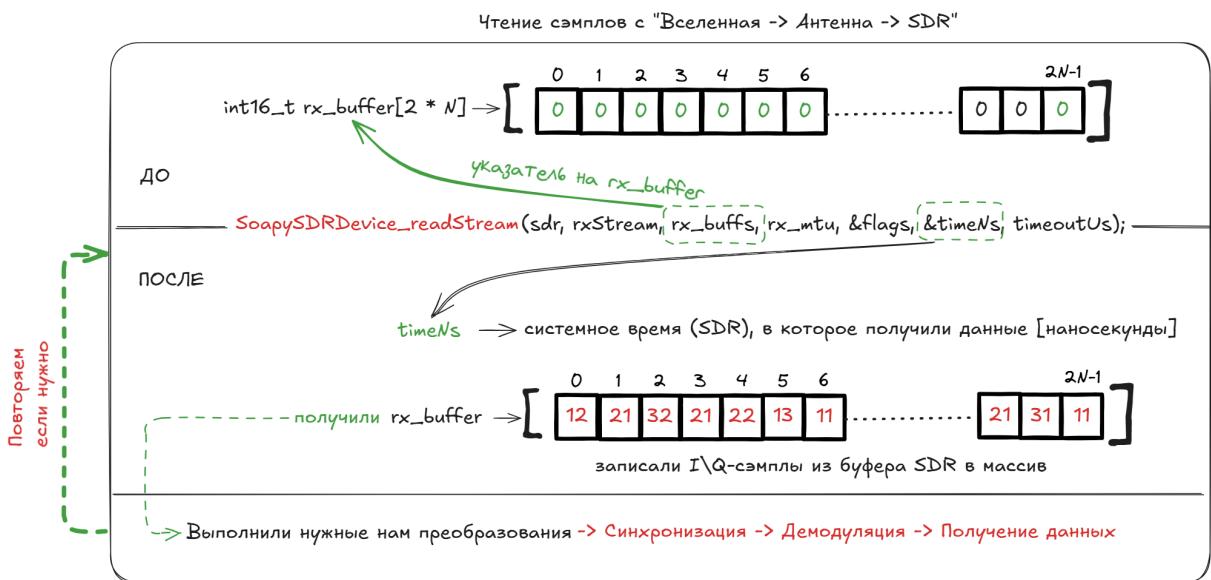


Рисунок 3 — Запись RX семплов

Для записи RX семплов в буффер используется функция `SoapySDRDevice_readStream()`, в которую необходимо передать указатель на буффер, в который хотим писать, и кол-во семплов, которые мы хотим записать, и еще некоторые дополнительные параметры. После выполнение функции в наш буффер будут записаны семплы, принятые SDR.

Создание своих семплов и их отправка в Pluto SDR

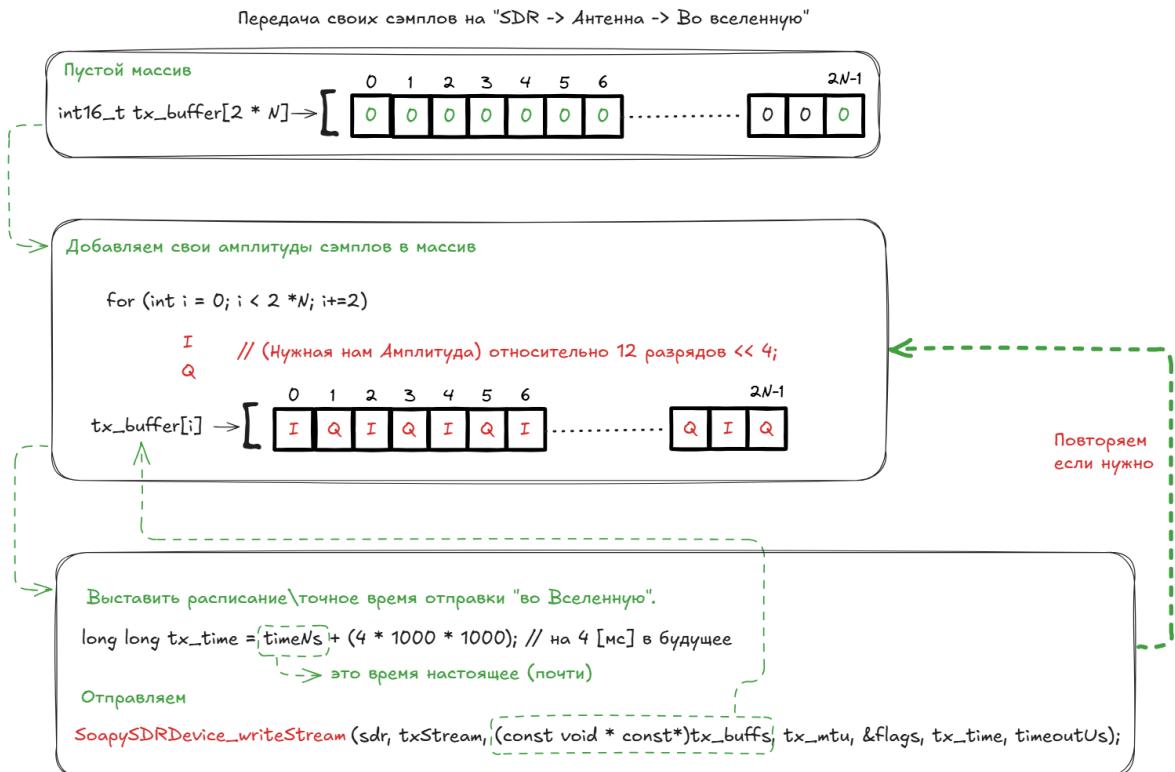


Рисунок 4 — Создание и отправка семплов

Для отправки семплов сначала необходимо заполнить tx буффер самими семплами. Формировать сами I и Q можно разными способами, но самое главное разместить их в правильном порядке. Для самой отправки используется функция SoapySDRDevice_writeStream(), в которую необходимо передать указатель на буффер с семплами и время, через которое семплы будут отправлены, а также дополнительные параметры. После вызова функции через 4нс наши семплы отправятся с Pluto SDR.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Введение

Необходимо заполнить tx буффер своими собственными семплами, отправить их, а потом принять и посмотреть, что примит Pluto SDR. То, как заполнять буффер семплами, решает студент.

Формирование семплов

Перевод строки в бинарный вид

Я буду формировать простой прямоугольный сигнал, который будет передавать небольшую строку. Чтобы передавать текст, нужно сначала перевести символы (char) в биты. Для этого я написал специальную функцию:

```
uint8_t* stob(char* str, int* out_bits_count) {
    // get string len
    int len = strlen(str);

    // 1 char = 8 bit
    *out_bits_count = len * sizeof(char);

    // massive for bits
    uint8_t* bits = (uint8_t*)malloc(*out_bits_count *
        sizeof(uint8_t));

    // check pointer
    if (bits == NULL){
        return NULL;
    }

    char c;

    // iterate on string
    for (int i = 0; i < len; i++) {
        // get char
        c = str[i];
        // iterate on bits massive
```

```
    for (int j = 0; j < 8; j++) {
        // convert char to bits
        bits[i * 8 + j] = (c >> (7 - j)) & 1;
    }

}

return bits;
}
```

Функция принимает саму строку и указатель на переменную, в которую запишет число битов, чтобы знать размер массива после завершения функции, и возвращает битовую последовательность. Пример работы функции:

Переведем в бинарный вид строку "Hello World":

```
int bits_count;
uint8 t* bits = stob("Hello World", &bits count);
```

На выходе получим такую последовательность:

```
0 1 0 0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0 1 1 0 0 0 1 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1
```

Рисунок 5 — "Hello World" в бинарном виде

Способ кодирования

Чтобы передать нули и единицы я буду использовать самый простой способ кодирования: 0 - ноль, 1 - максимальный I и минимальный O.

Заполнение буфера семплами

Для заполнения tx буфера я написал отдельную функцию, которая принимает битовую последовательность (сообщение), длину битовой последовательности и размер sempла, а возвращает массив sempлов.

```
int16_t* bits_to_samples(uint8_t* bits, int bits_count, int tx_mtu){  
  
    // allocate memory
```

```

int16_t* tx_buff = (int16_t*)malloc(sizeof(int16_t) * tx_mtu
* 2);

// iterate on bits
for (int i = 0; i < bits_count; i += 1)
{
    // fill tx_buff with samples
    for(int j = i*TAU_ON_BITS; j < i*TAU_ON_BITS + 20 && j <
tx_mtu*2; j+=2){
        if(bits[i]){
            tx_buff[j] = 2047 << 4; // I
            tx_buff[j+1] = -2047 << 4; // Q
        } else{
            tx_buff[j] = 0; //I
            tx_buff[j+1] = 0; //Q
        }
    }
}

return tx_buff;
}

```

Самое важное здесь - учитывать продолжительность импульса, т.е то кол-во семплов, которое будет приходиться на один бит информации. Я выбрал 10 семплов на бит (TAU), а это значит, что на 1 бит информации будет приходиться 20 элементов массива (TAU_ON_BITS). Чем выше длительность сигнала, тем выше шанс шанс верно декодировать его на применой стороне, но при этом падает скорость передачи данных. Работать заполнение будет так: берем первые 20 элементов массива и заполняем его идентичными значениями (в соответствии со состоянием бита), потом берем следующие 20 и т.д.

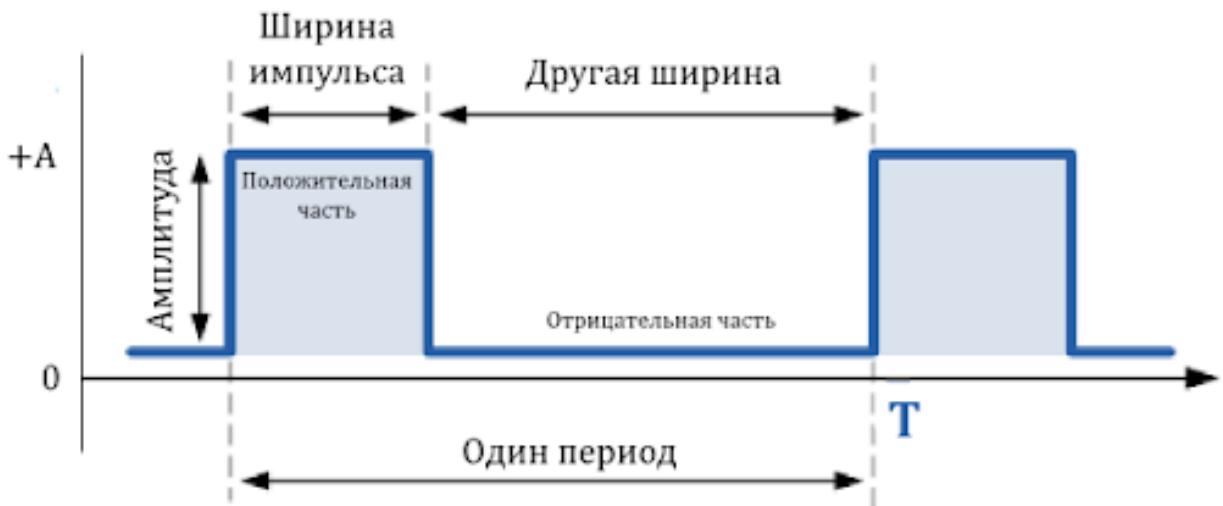


Рисунок 6 — Пример прямоугольного сигнала с обозначениями

Визуализация полученного сигнала с помощью скрипта на Python

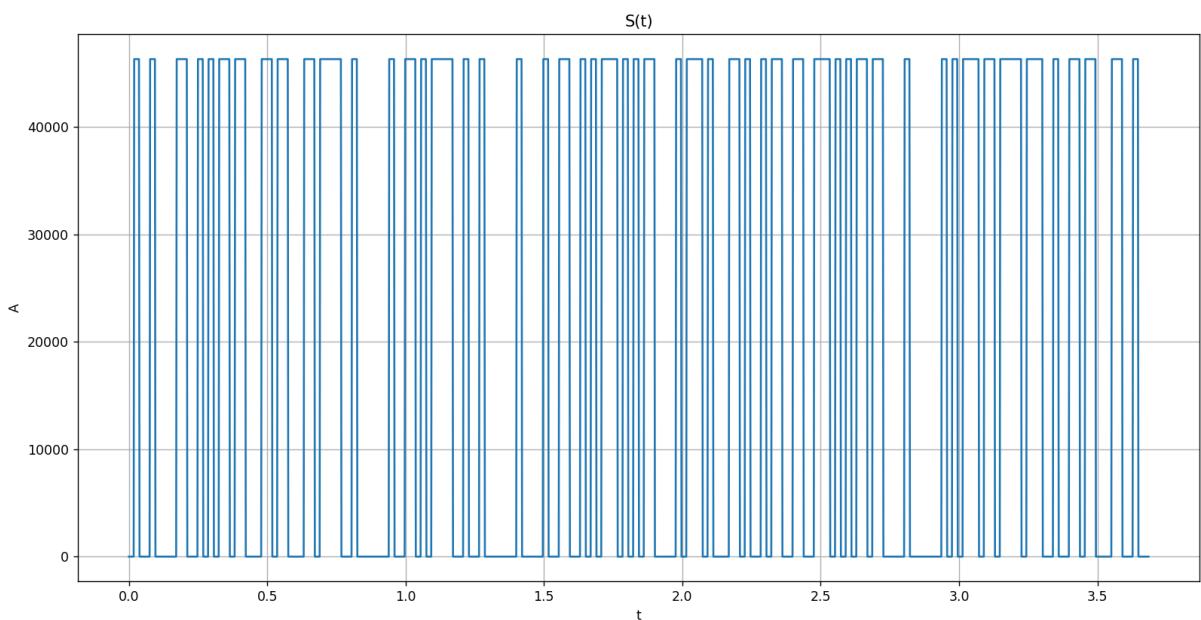


Рисунок 7 — Пример полученного прямоугольного сигнала

Этот прямоугольный сигнал содержит наше сообщение.

Требование к длине сообщения

Для Pluto SDR рекомендуется использовать для отправки/приема 1920 семплов или число семплов кратное 1920. Если на 1 бит приходится 10

семплов, то сообщение должно быть длиной 192 бита, а т.к я пытаюсь передать символы размером 8 бит, то необходима строка длиной 24 символа.

Прием семплов

На приеме получили следующий сигнал:

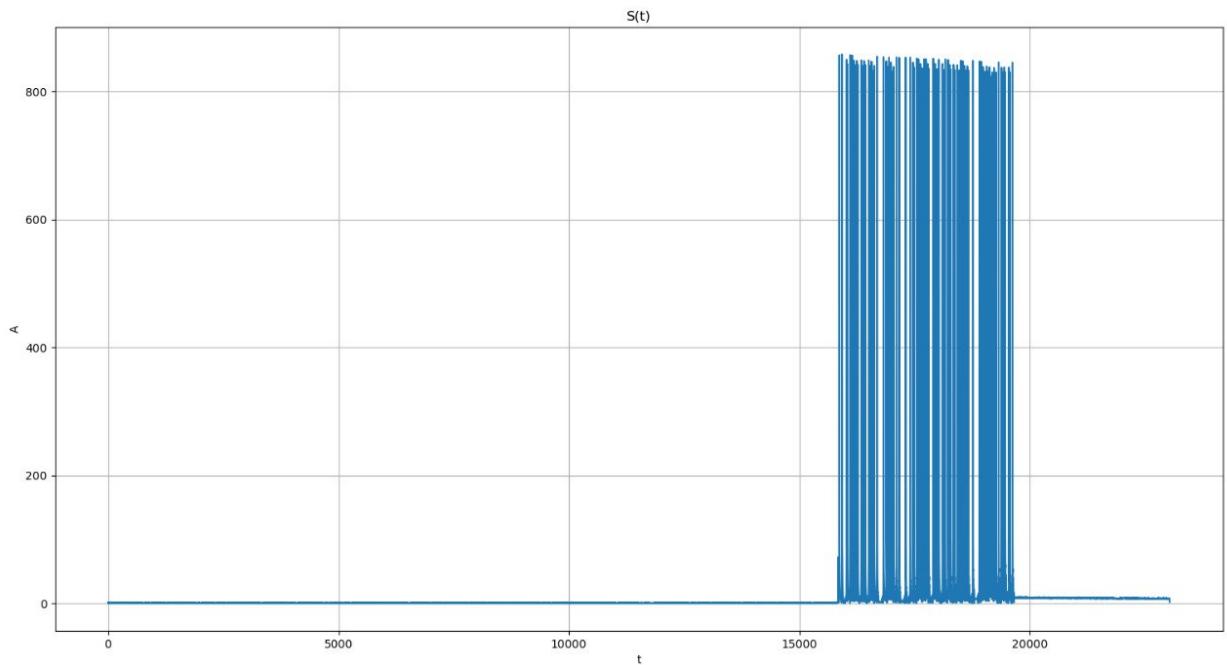


Рисунок 8 — Сигнал на приеме

Где то среди этого сигнала наше сообщение, но чтобы правильно его декодировать необходимо синхронизировать устройства с помощью специальных последовательностей. Эта тема будет рассматриваться в дальнейшем.

ВЫВОД

В ходе проделанной работы я лучше разобрался в библиотеке SoapySDR, сформировал собственные семплы, отправил их в радиоканал, а потом принял сигнал и визуализировал его.

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»

Кафедра телекоммуникационных систем и вычислительных средств
(ТС и ВС)

Отчет по производственной практике
по дисциплине
SDR

по теме:
АРХИТЕКТУРА SDR-УСТРОЙСТВ. ПРИМЕРЫ ФОРМИРОВАНИЯ
I/Q-СЭМПЛОВ ПРОИЗВОЛЬНОЙ ФОРМЫ. РАБОТА С БУФЕРОМ
ПРИЕМА SDR

Студент:
Группа ИА-331

Я.А Гмыря

Предподаватели:
Лектор
Семинарист
Семинарист

Калачиков А.А
Ахпашев А.В
Попович И.А

Новосибирск 2025 г.

СОДЕРЖАНИЕ

1 ЦЕЛЬ И ЗАДАЧИ	3
2 ЛЕКЦИЯ	4
3 ПРАКТИЧЕСКАЯ ЧАСТЬ	11
4 ВЫВОД	20

ЦЕЛЬ И ЗАДАЧИ

Цель:

Повторить архитектуру систем цифровой связи, изучить базовые типы модуляции BPSK и QPSK, программно реализовать их.

Задачи:

1. Прослушать лекцию.
2. На основе полученных знаний выполнить программную реализацию BPSK и QPSK модуляции.
3. Создать треугольный сигнал и сигнал параболической формы, отправить его с SDR, а потом принять и посмотреть, что получилось.

ЛЕКЦИЯ

Введение

На этом занятии вспомним архитектуру цифровой системы связи и рассмотрим BPSK и QPSK модуляции.

Архитектура цифровой системы связи

Задачи цифровой системы связи

Какие задачи у системы связи? Задача системы связи заключается в надежном передать поток бит на заданной скорости по каналу связи. Для передачи по каналу связи мы используем электромагнитные колебания - \sin и \cos какой-то частоты.

Архитектура передатчика

Базовая архитектура цифровой системы связи в простейшем случае состоит из приемника, передатчика и радиоканала. Рассмотрим упрощенную архитектуру передатчика:

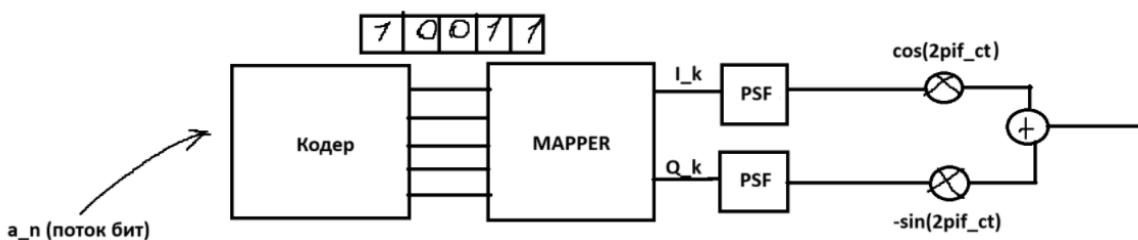


Рисунок 1 — Архитектура передатчика

Формируется поток бит, который поступает на кодер. В нашем случае кодер просто делит поток бит на блоки определенной длины. У кодера один вход, по которому последовательно поступают биты, а на выходе N-битная шина, которая уже параллельно передает биты на мапер. Кодер можно представить в виде схемы следующим образом:

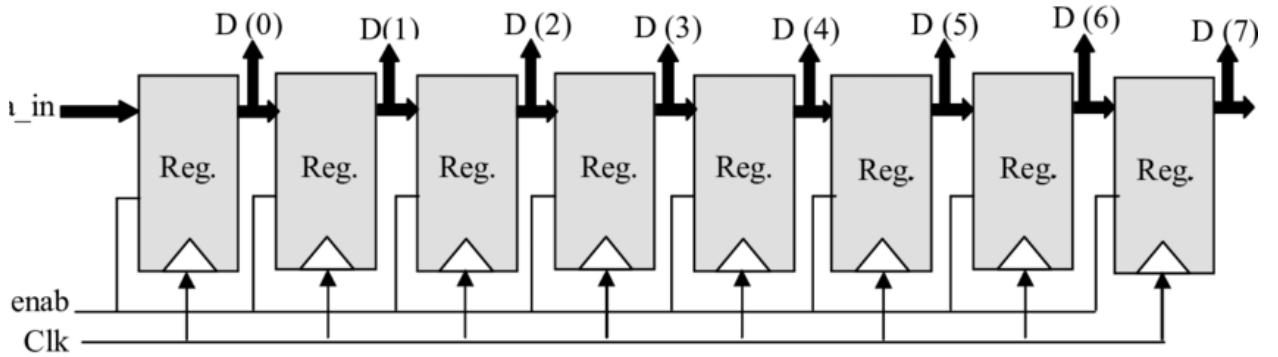


Рисунок 2 — Схема последовательно-параллельного преобразователя

Имеется N триггеров с общим тактовым сигналом. По каждому тактовому сигналу поток бит будет "продвигаться" по триггерам и попадать на N -битную шину, т.е параллельно выводиться из устройства.

Далее блоки битов попадают на маппер, который ставит в соответствие каждому блоку числа I и Q . Если блок бит имеет размерность N , то в маппере будет заложено 2^N комбинаций. На выходе маппера параллельно получим числа I и Q .

Сигнал, который мы хотим сформировать имеет вид $S_k(t) = I_k \cos(2\pi f_c t) - Q_k \sin(2\pi f_c t)$, где k - номер текущего символа, I_k , Q_k - координаты символа в сигнальной диаграмме. Эти координаты мы получаем на выходе маппера. Эти числа в будущем станут параметрами колебания.

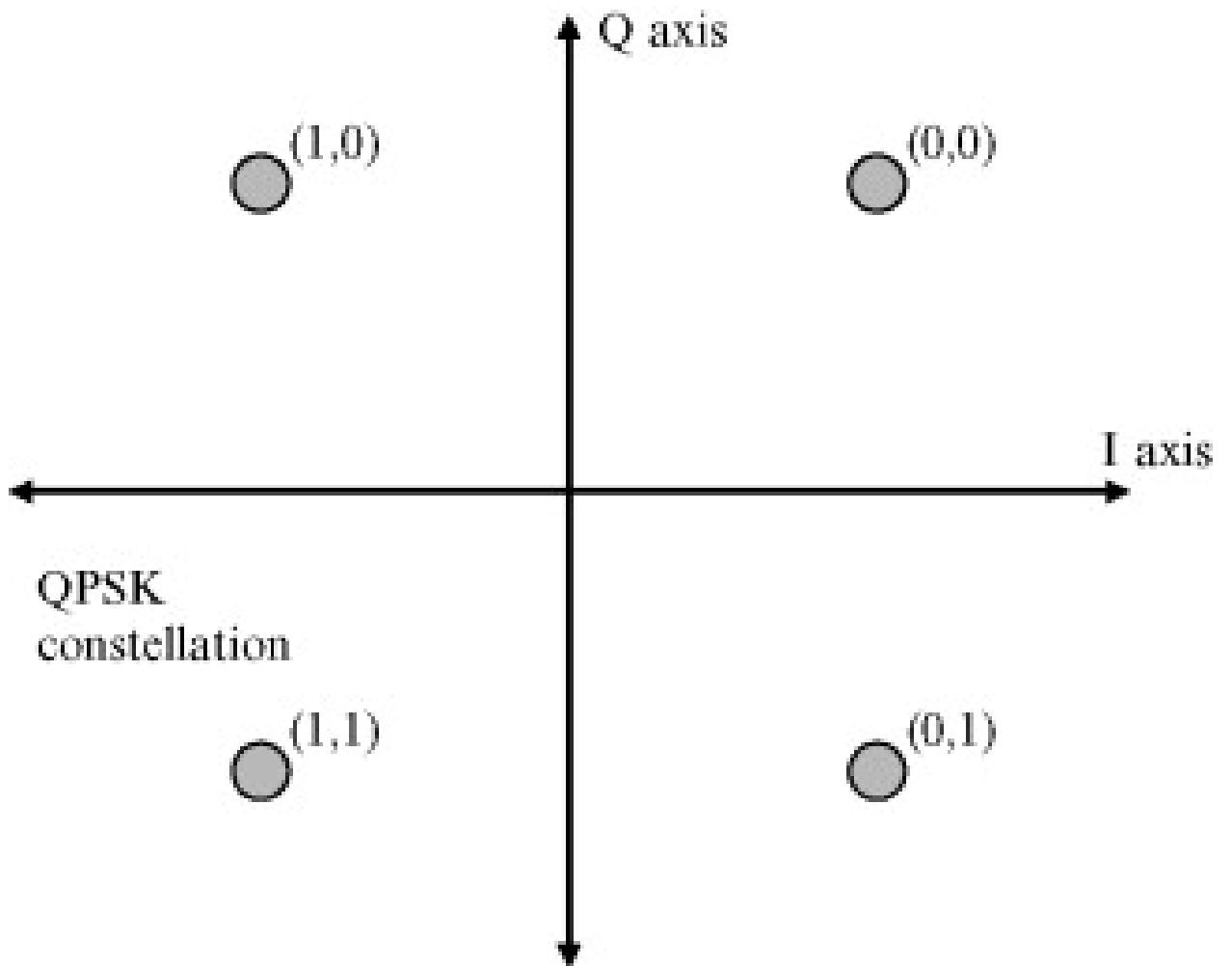


Рисунок 3 — Пример сигнальной диаграммы для QPSK модуляции

Далее I и Q попадают на Pulse Shaping Filter (PSF). Фильтр выполняет 2 задачи: определяет ширину спектра радиосигнала и его форму, а также применяется на приемной стороне для символьной синхронизации. Фильтр характеризуется важным параметром во временной области - импульсной характеристикиой $g(t)$. Импульсная характеристика может быть различной, но для простоты восприятия будем рассматривать только прямоугольную характеристику. На выходе фильтра получим прямоугольные отрезки сигналов.

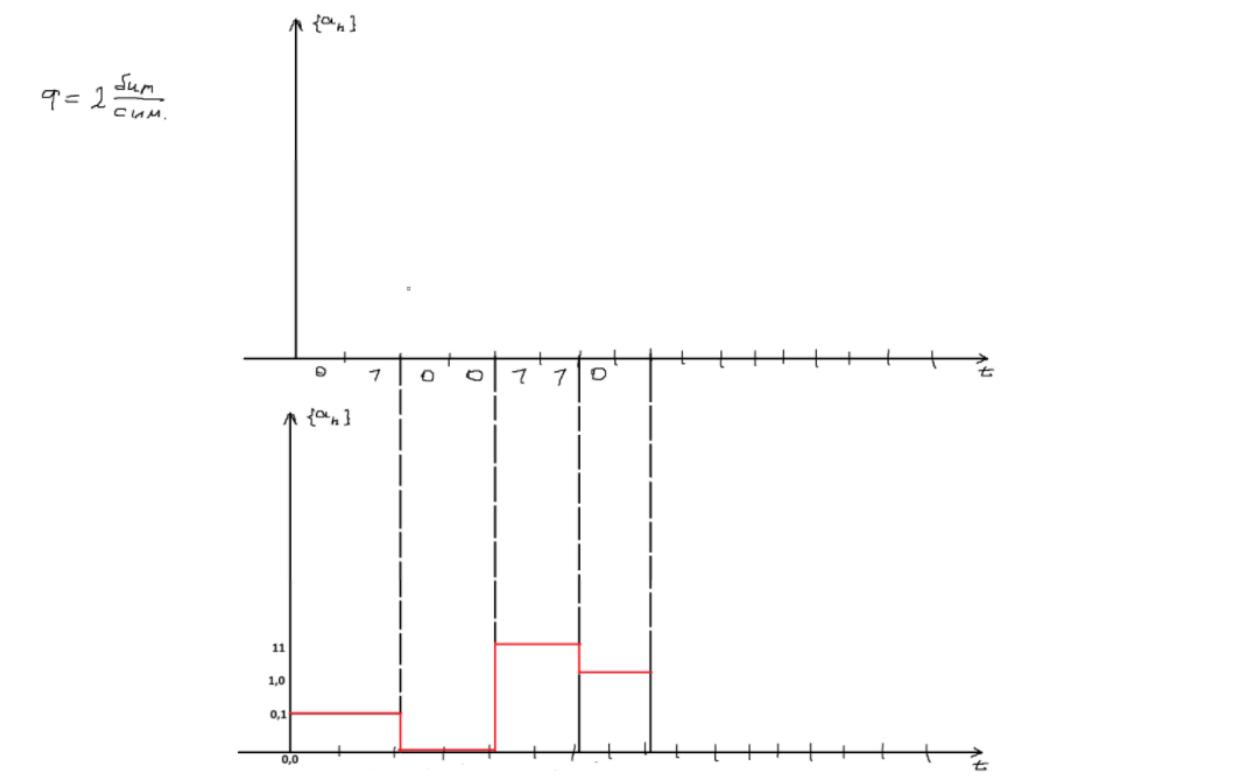


Рисунок 4 — Пример формирования символов

Длительность символов вычисляется как $1/R_s$, где R_s - символьная скорость. В свою очередь символьная скорость R_s вычисляется как $R_b/\log_2(M)$, где R_b - битовая скорость (та скорость, с которой биты поступают на маппер), M - кол-во точек созвездия. После маппера битовая скорость переходит в символьную, которая определяет длительность прямоугольного импульса.

Схемы модуляции

BPSK

BPSK (Binary Phase Shift Keying) - двухпозиционная фазовая манипуляция. В такой схеме модуляции на 1 бит приходится 1 символ.

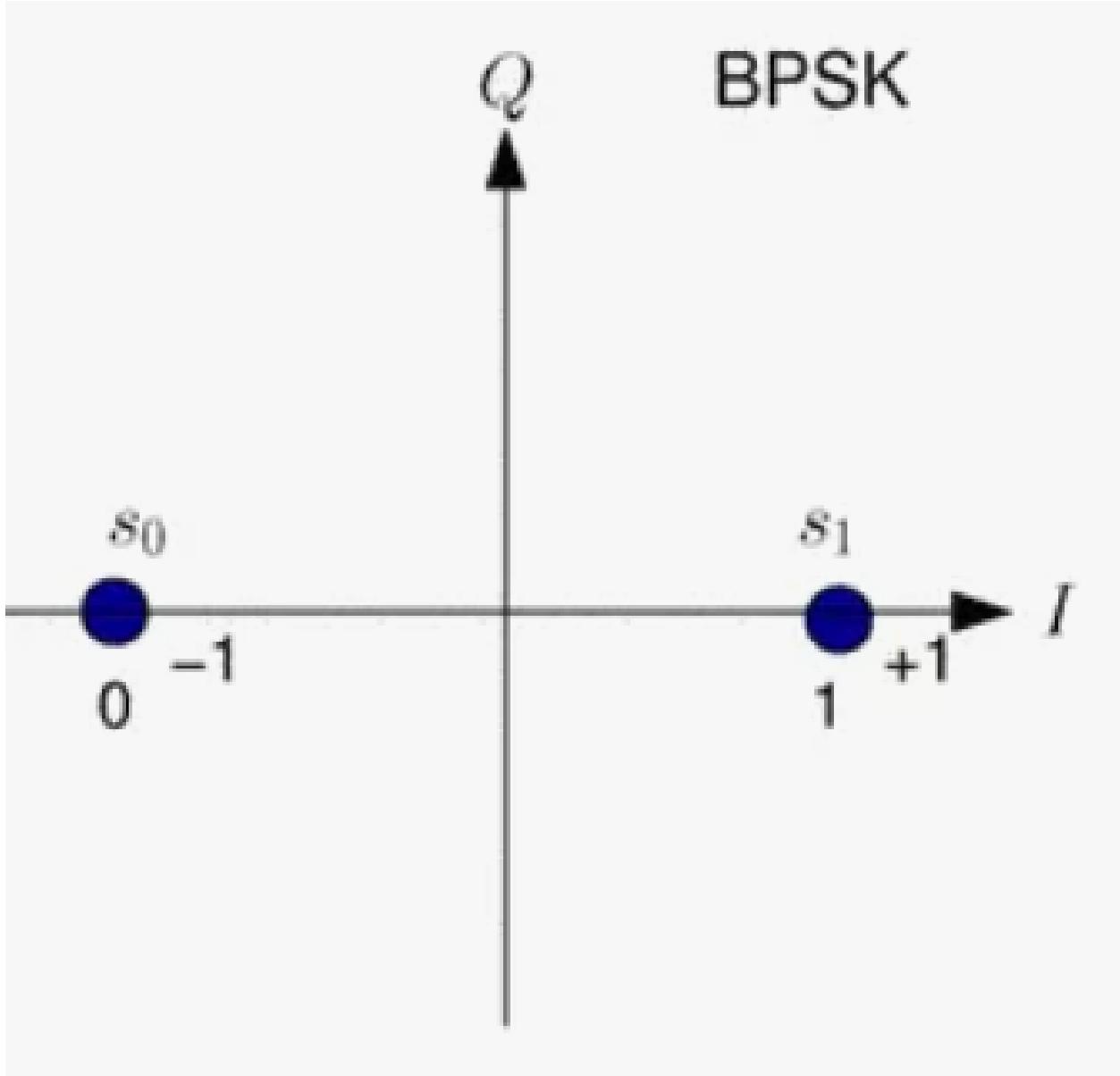


Рисунок 5 — Созвездие BPSK модуляции

Логика BPSK следующая: если бит имеет состояние 0, то он будет соответствовать сигналу с начальной фазой ϕ равной 0, т.е точке с координатой (1, 0), если бит находится в состоянии 1, то биту будет соответствовать сигнал с начальной фазой ϕ равной π , т.е точке с координатой (-1, 0). Можем заметить, что квадратурная составляющая всегда равна 0, это значит, что в сигнале будет только cos составляющая, т.к sin составляющая занулится (видно из уравнения сигнала $S_k(t) = I_k \cos(2\pi f_c t) - Q_k \sin(2\pi f_c t)$).

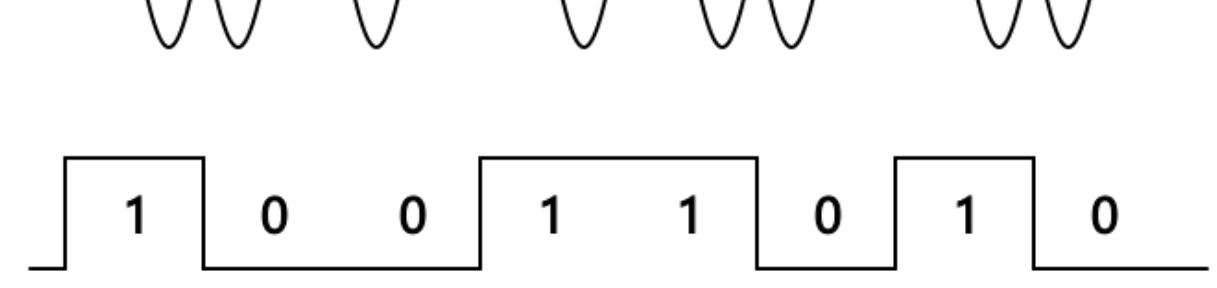


Рисунок 6 — Пример кодирования модуляцией BPSK

QPSK

QPSK (Quadrature Phase Shift Keying) - квадратурная фазовая манипуляция. 2 бита кодируются в одном символе. Используется 4 различных фазы для представления пар битов. Логика этой схемы модуляции такая же, как у BPSK, но уже добавляется квадратурные составляющие, т.е в сигнале будут как \cos составляющая, так и \sin составляющая. Точке $(0, 0)$ будет соответствовать фаза $\frac{\pi}{4}$, $(1, 0)$ будет соответствовать фаза $\frac{3\pi}{4}$, $(1, 1)$ будет соответствовать фаза $\frac{5\pi}{4}$, $(0, 1)$ будет соответствовать фаза $\frac{7\pi}{4}$,

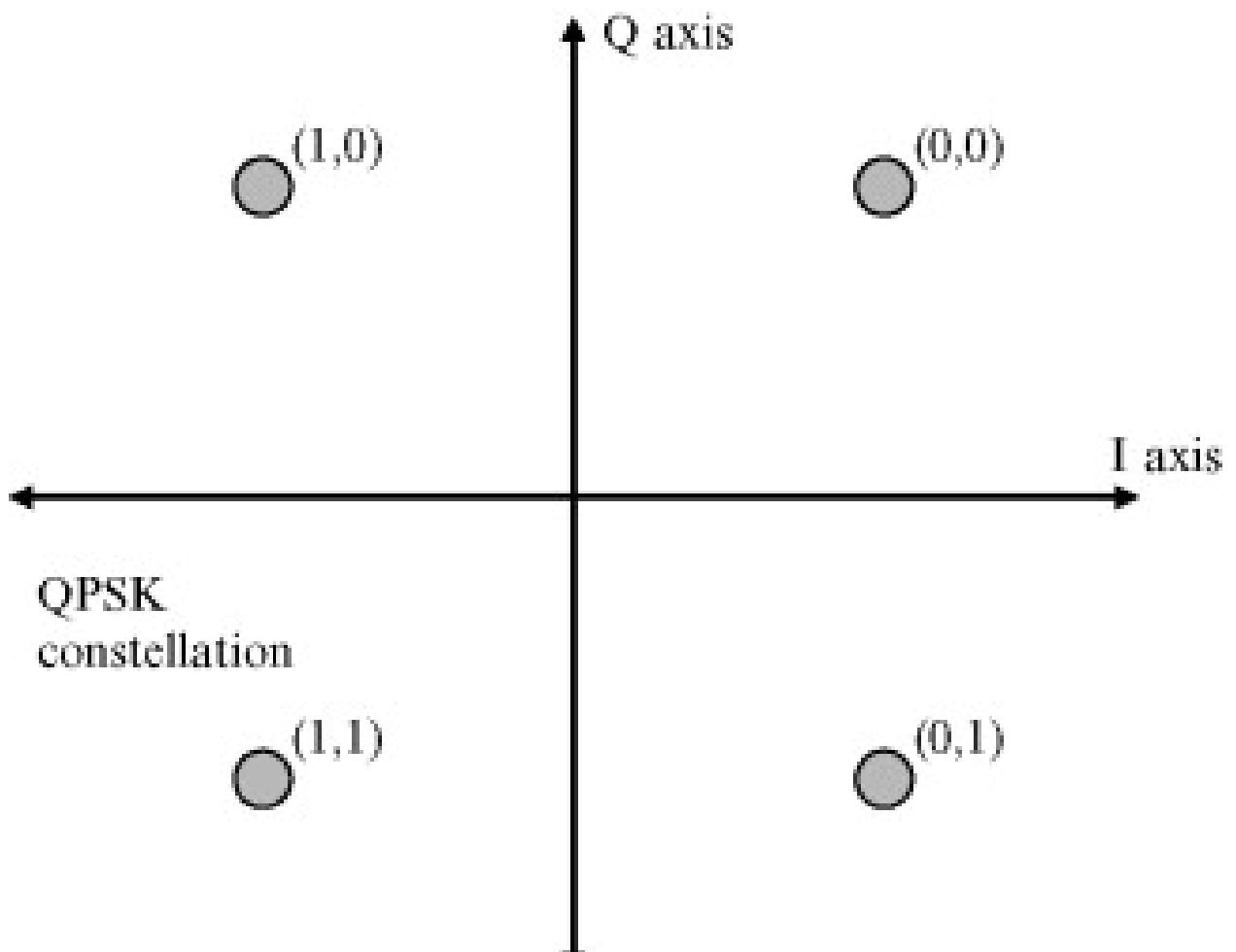


Рисунок 7 — QPSK созвездие

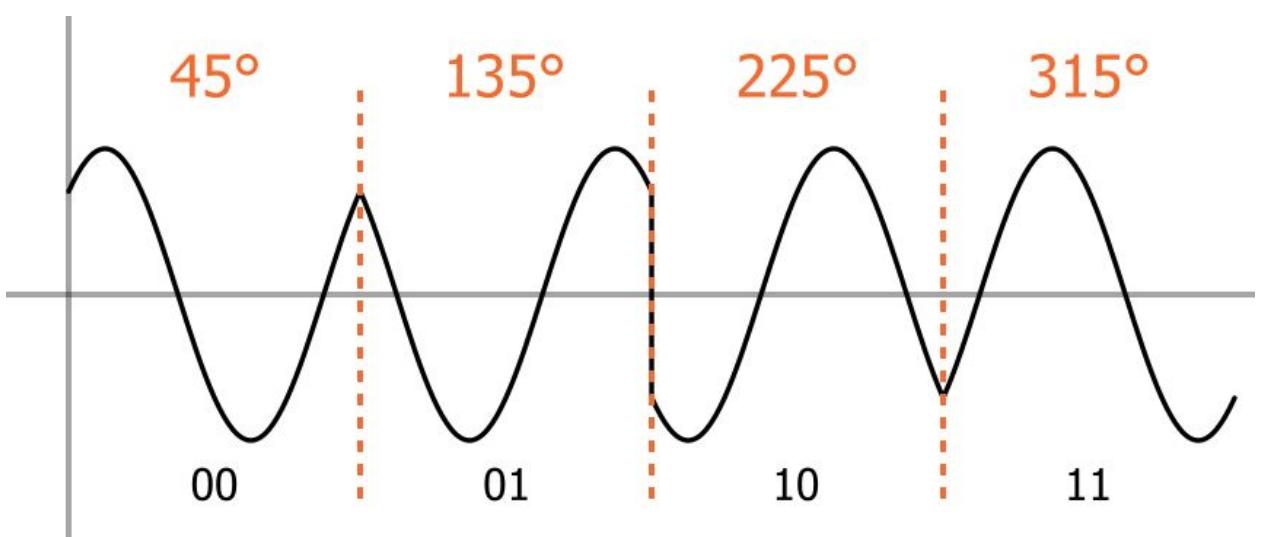


Рисунок 8 — Пример кодирования модуляцией QPSK

ПРАКТИЧЕСКАЯ ЧАСТЬ

Введение

На практике продолжаем реализовывать сигналы разной формы. На этом занятии реализую сигнал параболической формы и треугольный сигнал.

Треугольный сигнал

Я буду формировать такой сигнал по следующему принципу:

1. Если бит равен единице, то в момент длительности сигнала $\frac{\tau}{2}$ буду задавать максимальное значение, а в остальных случаях буду задавать нули. Таким образом в одной точке будет образовываться пик, который и будет придавать сигналу форму треугольного.
2. Если бит равен 0, то на всей длительности τ сигнал будет принимать 0.

Программная реализация

```
#define TAU 10
#define TAU_ON_ELEMENT 20

int16_t* bits_to_triangle_signal(uint8_t* bits, int
    bits_count, int tx_mtu){

    // allocate memory
    int16_t* tx_buff = (int16_t*)malloc(sizeof(int16_t) * tx_mtu
        * 2);

    // iterate on bits
    for (int i = 0; i < bits_count; ++i)
    {
        // fill tx_buff with samples
        for(int j = i*TAU_ON_ELEMENT; j < i*TAU_ON_ELEMENT + 20
            && j < tx_mtu*2; j+=2){
```

```

        if(bits[i] && j == i*TAU_ON_ELEMENT + 8){
            tx_buff[j] = 2047 << 4;      // I
            tx_buff[j+1] = -2047 << 4; // Q
        } else{
            tx_buff[j] = 0;           //I
            tx_buff[j+1] = 0;         //Q
        }
    }

    return tx_buff;
}

```

Код полностью идентичен коду, который использовался для создания прямоугольного сигнала. Единственное отличие заключается в условии

$j == i * \text{TAU_ON_ELEMENT} + 8$

Это условие как раз и позволяет выбрать одну конкретную точку, в которой задается максимальное значение.

Визуализируем сигнал:

Отправим сформированный сигнал и посмотрим на результат после приема.

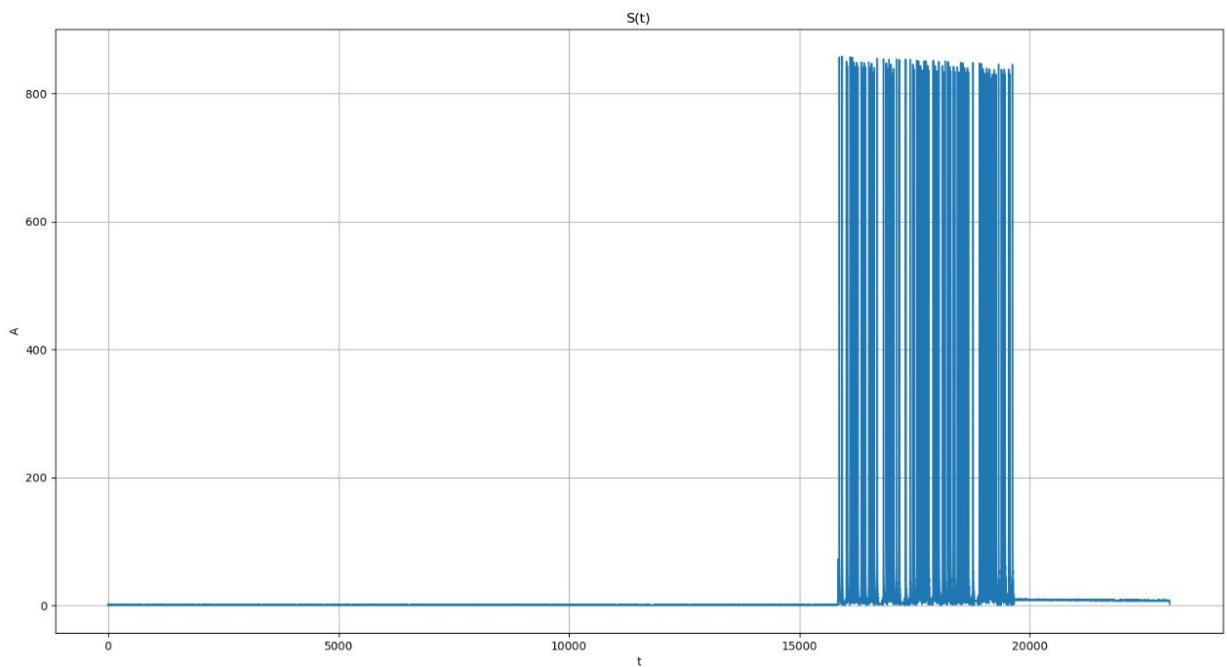


Рисунок 9 — Пример принятого треугольного сигнала

Параболический сигнал

Сформируем параболу, перебирая точки от $-\frac{tx_mtu}{10}$ до $-\frac{tx_mtu}{10} + 2\frac{tx_mtu}{10}$, где tx_mtu - кол-во семплов, передаваемых за раз. Деление на 10 используется для масштабирования, чтобы не возводить большие числа в квадрат.

Программная реализация

```
int16_t* parabola_signal(int tx_mtu){

    // allocate memory
    int16_t* tx_buff = (int16_t*)malloc(sizeof(int16_t) * tx_mtu
        * 2);

    float coef = -tx_mtu / 10;

    for (int i = 0; i < 2 * tx_mtu; i+=2)
    {
        tx_buff[i] = int16_t(coef * coef);    // I
        tx_buff[i+1] = int16_t(coef * coef); // Q

        coef += 0.1;
    }

    return tx_buff;
}
```

Визуализация

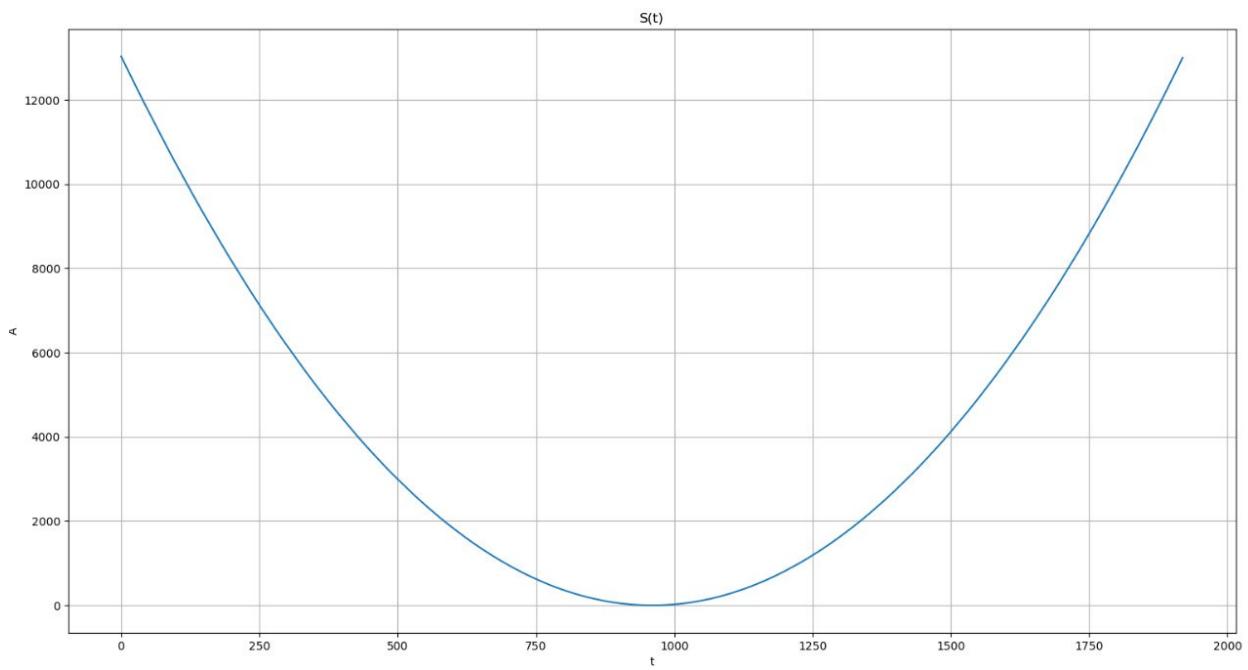


Рисунок 10 — Пример сигнала в виде параболы

Визуализация после приема

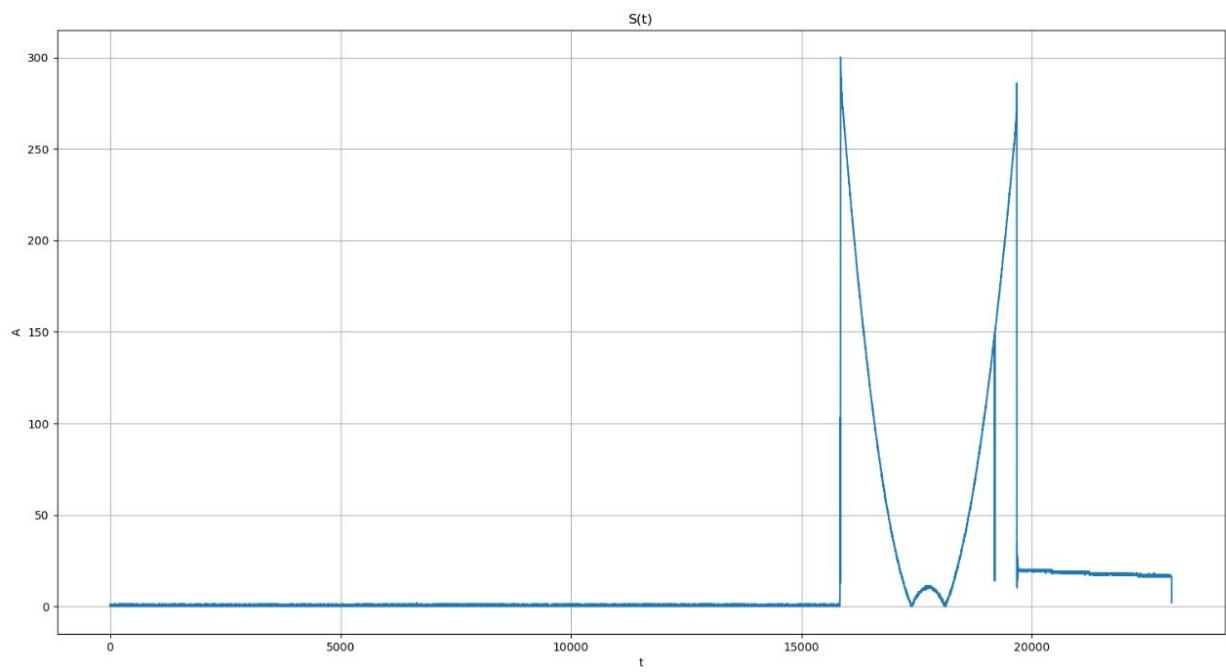


Рисунок 11 — Пример сигнала в виде параболы на приеме

Можем наблюдать параболу в нашем сигнале, по центру парабола отображена вверх, потому что я строил график модуля сигнала.

Реализация BPSK модуляции

Шаги по реализации:

1. Написать BPSK модуляцию (логика маппера)
2. На основе I и Q, полученных на прошлом шаге, сгенерировать прямоугольные импульсы (логика формирующего фильтра)
3. Перемножить прямоугольные импульсы на несущее колебание
4. Проанализировать полученные результаты и сделать вывод

BPSK модулятор

Реализация на языке C:

```
double* BPSK_modulation(int* bits, int bits_count){  
    //allocate memory  
    double* IQ_samples = (double*)malloc(sizeof(double) *  
        bits_count * 2);  
    //iterate on bits  
    for(int i,j = 0; i < bits_count * 2; i+=2){  
        if(bits[j]){  
            IQ_samples[i] = 1;      // I  
            IQ_samples[i + 1] = 0;  // Q  
        }else{  
            IQ_samples[i] = -1;    // I  
            IQ_samples[i + 1] = 0; // Q  
        }  
  
        ++j;  
    }  
  
    return IQ_samples;  
}
```

Сначала выделяем память под IQ семплы, т.к на каждый бит приходится 1 семпл, а на 1 семпл - 2 числа (I и Q), то размер массива с семплами должен быть вдвое больше кол-ва бит. Далее перебираем биты и в зависимости от его значения формируем семпл. В результирующем массиве I и Q идут в строгой последовательности $I_0, Q_0, I_1, Q_1, \dots, I_N, Q_N$. Массив в дальнейшем запишется в файл.

Формирующий фильтр

На языке Python парсим файл с семплами из прошлого шага. Далее формируем прямоугольные импульсы. Визуализация импульсов:

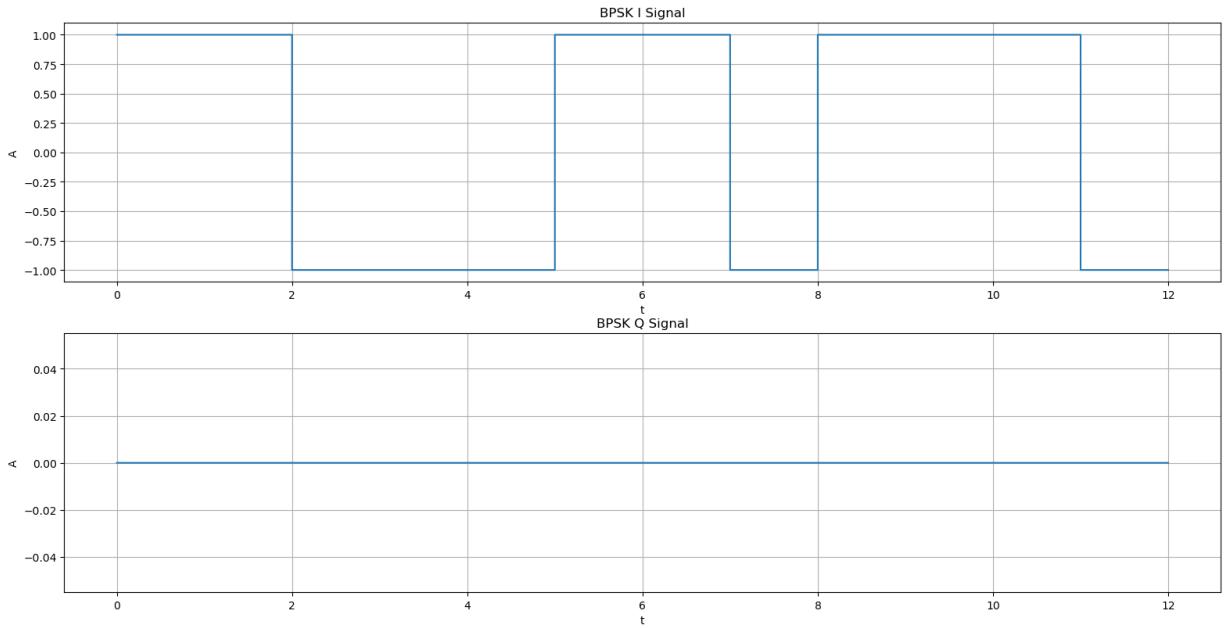


Рисунок 12 — Символы

Здесь явно видно, что в BPSK модуляции квадратурная составляющая всегда равна 0. Я передавал последовательность в 12 бит и это заняло 12 секунд, т.е на каждый бит приходился 1 символ, который длился 1 секунду.

Перемножение с несущим колебанием

Теперь перемножим символы с несущим колебанием. В роли несущего колебания я использовал $\cos(2\pi ft)$ и $-\sin(2\pi ft)$ с частотой $f = 1$. В реальности частоты несущих колебаний в разы больше, но для наглядности я взял маленькое значение. Этот процесс можно сравнить с наложением маски, где маской является прямоугольный сигнал. Если посмотреть на схему цифровой системы связи, которая была дана в разделе с теорией, то этот процесс происходит в цепи, которая следует после формирующего фильтра. Визуализация результата:

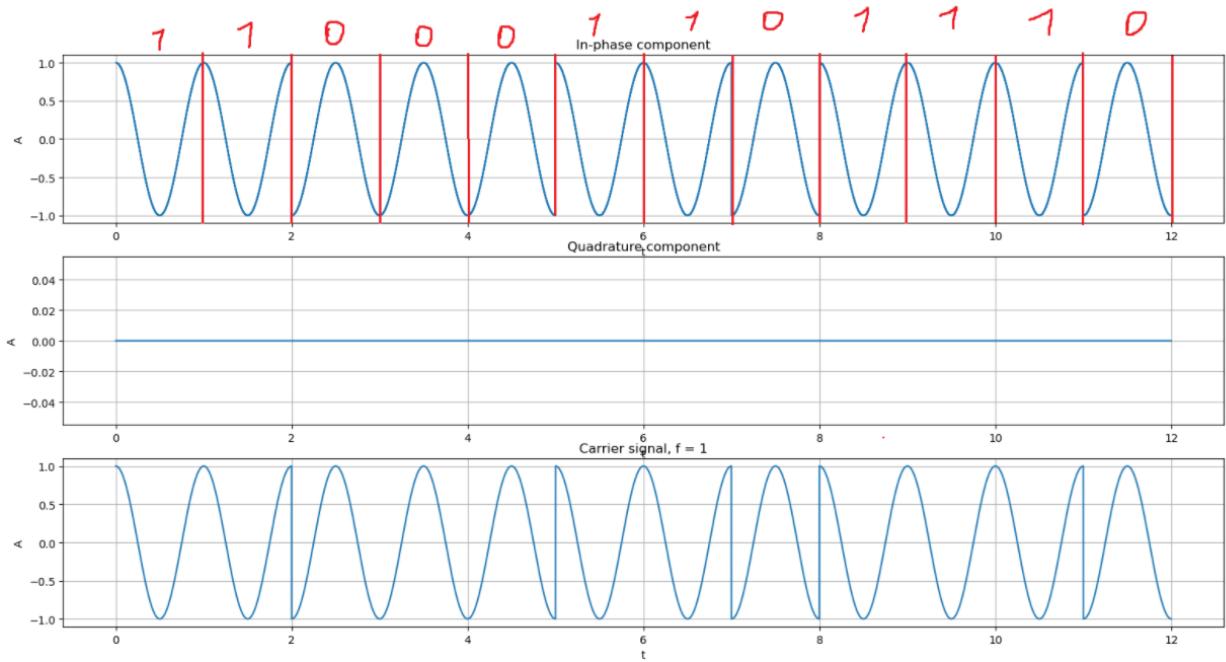


Рисунок 13 — Перемножение с несущим сигналом

Можем видеть, что квадратурная компонента по прежнему равна 0, т.е в сигнале присутствует только \cos . В синфазной компоненте можем видеть, как меняется фаза колебания, которая отражает значение бита. Если фаза равна 0, то передается 1, если фаза равна π , то передается 0. В местах, где меняется фаза, происходит смена значения бита. Таким образом можно восстановить передаваемую битовую последовательность. Результирующий сигнал будет иметь вид синфазной компоненты, т.к квадратурная всегда равна 0.

Реализация QPSK модуляции

Шаги по реализации будут в точности такие же, как у BPSK модуляции, но слегка будет отличаться реализация.

Реализация на языке C:

```
double* QPSK_modulation(int* bits, int bits_count){
    //allocate memory
    double* IQ_samples = (double*)malloc(sizeof(double) *
        bits_count);
    //iterate on bits
    for(int i = 0; i < bits_count; i+=2){
        if(bits[i]){
            IQ_samples[i] = -1;           //I
            IQ_samples[i+1] = 1;         //Q
        }
        else{
            IQ_samples[i] = 1;          //I
            IQ_samples[i+1] = -1;       //Q
        }
    }
    return IQ_samples;
}
```

```

        if(bits[i + 1]){
            IQ_samples[i + 1] = -1; //Q
        }else{
            IQ_samples[i + 1] = 1; // Q
        }
    }else{
        IQ_samples[i] = 1; //I
        if(bits[i + 1]){
            IQ_samples[i + 1] = 1; //Q
        }else{
            IQ_samples[i + 1] = -1; //Q
        }
    }
}

return IQ_samples;
}

```

Заметим, что в случае QPSK модуляции массив под семплы вдвое меньше, т.к в QPSK модуляции на 1 семпл приходится 2 бита.

Формирующий фильтр

Визуализация импульсов:

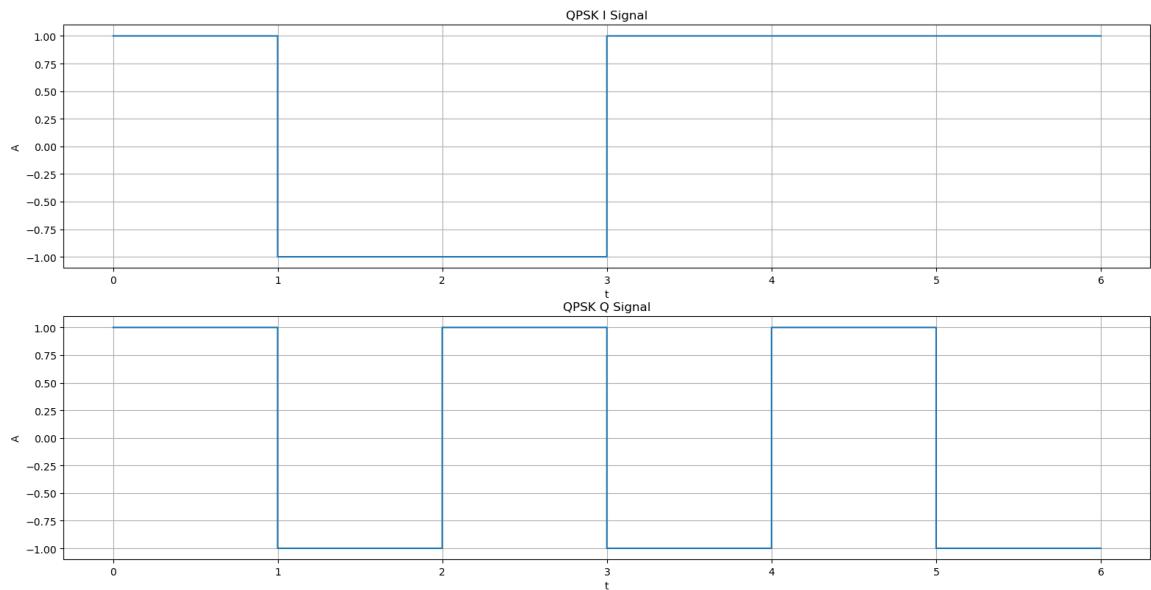


Рисунок 14 — Символы

Можем заметить, что появляется квадратурная часть. Также можно заметить, что та же последовательность из 12 бит передается уже за 6 секунд. Это связано с тем, что в QPSK на сэмпл приходится 2 бита. Можно сделать вывод о том, что чем больше точек в сигнальном созвездии, тем выше скорость передачи данных. 1 символ длится уже не 1 секунду, а 0.5 секунд (на графике чуть неверный масштаб)

Перемножение с несущим колебанием

Визуализация результата:

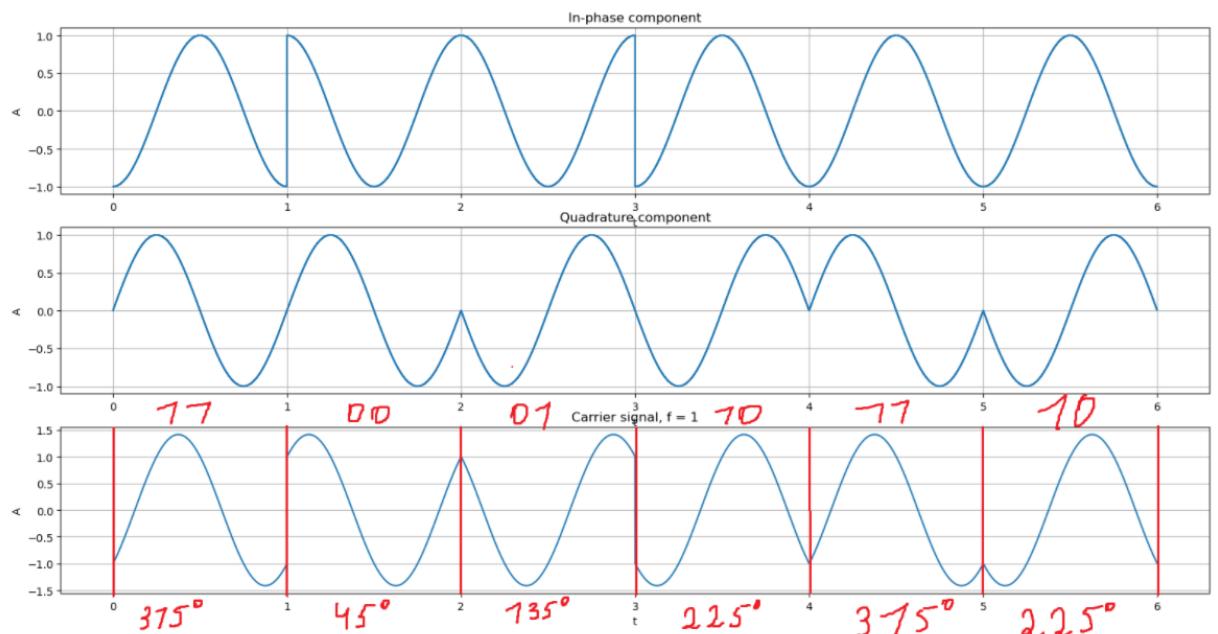


Рисунок 15 — Перемножение с несущим сигналом

Можем видеть, что квадратурная компонента теперь не равна 0, т.е в сигнале присутствует \cos и \sin . Еще можно заметить, что максимальная амплитуда в результирующем сигнале уже не равна 1, а равна $\sqrt{I^2 + Q^2} = \sqrt{1^2 + 1^2} = \sqrt{2}$. Результирующий сигнал будет иметь вид $I\cos(2\pi f_c t) - Q\sin(2\pi f_c t)$.

ВЫВОД

В ходе проделанной работы я реализовал сигналы разной формы, отправил их в радиоканал, потом принял и визуализировал форму сигнала. Также реализовал BPSK и QPSK модуляцию.

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»

Кафедра телекоммуникационных систем и вычислительных средств
(ТС и ВС)

Отчет по производственной практике
по дисциплине
SDR

по теме:
АРХИТЕКТУРА SDR-УСТРОЙСТВ. ПРИМЕРЫ ФОРМИРОВАНИЯ
I/Q-СЭМПЛОВ ПРОИЗВОЛЬНОЙ ФОРМЫ. РАБОТА С БУФЕРОМ
ПРИЕМА SDR

Студент:
Группа ИА-331

Я.А Гмыря

Предподаватели:
Лектор
Семинарист
Семинарист

Калачиков А.А
Ахпашев А.В
Попович И.А

Новосибирск 2025 г.

СОДЕРЖАНИЕ

1 ЦЕЛЬ И ЗАДАЧИ	3
2 ЛЕКЦИЯ	4
3 ПРАКТИЧЕСКАЯ ЧАСТЬ	9
4 ВЫВОД	17

ЦЕЛЬ И ЗАДАЧИ

Цель:

Более детально рассмотреть принцип работы формирующего фильтра. Программно реализовать логику формирующего фильтра, передать мелодию по радиоканалу.

Задачи:

1. Прослушать и законспектировать лекцию.
2. На основе полученных знаний выполнить программную реализацию формирующего фильтра.
3. Отправить сигнал по радиоканалу, потом принять его и попытаться воспроизвести.

ЛЕКЦИЯ

Введение

На прошлом занятии мы познакомились с двумя типами модуляции: BPSK и QPSK и программно реализовали их. При реализации логики PSF нужно было формировать прямоугольный импульс. Я сделал это самым простым методом: увеличивал кол-во значений I и Q. За счёт этого получали $I(t)$ и $Q(t)$, которые уже длились во времени. Данный подход рабочий и не является ошибкой, но на практике не используется, поскольку подходит только в случае, когда формирующий фильтр имеет прямоугольную импульсную характеристику. Если импульсная характеристика имеет форму приподнятого косинуса, то такой метод не сработает. На этом занятии изучим более грамотный подход.

Почему форма символов так важна?

Форма передаваемых символов $I_n(t), Q_n(t)$ определяет свойства спектра радиосигнала. Если форма символов прямоугольная, то форма спектра будет иметь вид функции $\frac{\sin x}{x}$ и будет занимать большую ширину спектра. В реальных системах чаще всего используется форма приподнятого косинуса.

Upsampling

Итак, мы хотим, чтобы I и Q были не просто числами, а имели длительность, т.е хотим получить $I(t)$ и $Q(t)$. Необходимо установить число семплов, которое будут длиться I и Q. Введем параметр L, который измеряется в $\frac{\text{sample}}{\text{symbol}}$ и будет отвечать за кол-во семплов, приходящихся на 1 символ, т.е за длительность символа. Если мы хотим сделать символы дляящихися, то необходимо поднять частоту дискретизации. Для этого существуют специальные блоки, которые называются Upsampling блоками. Их задача состоит в увеличении частоты дискретизации. Во сколько раз нужно увеличить частоту дискретизации? Если на каждый символ теперь приходится L семплов, то и частота дискретизации должна стать в L раз больше. За частоту дискретизации отве-

чает параметр f_{symb} (символьная скорость), который показывает скорость, с которой символы поступают из маппера. Соответственно после выхода из Upsampling блока получим $f_s = f_{symb} * L$, т.е кол-во семплов на секунду времени (sample_rate). Исходя из этого можно определить расстояние во времени между семплами $T_s = \frac{1}{f_s}$.

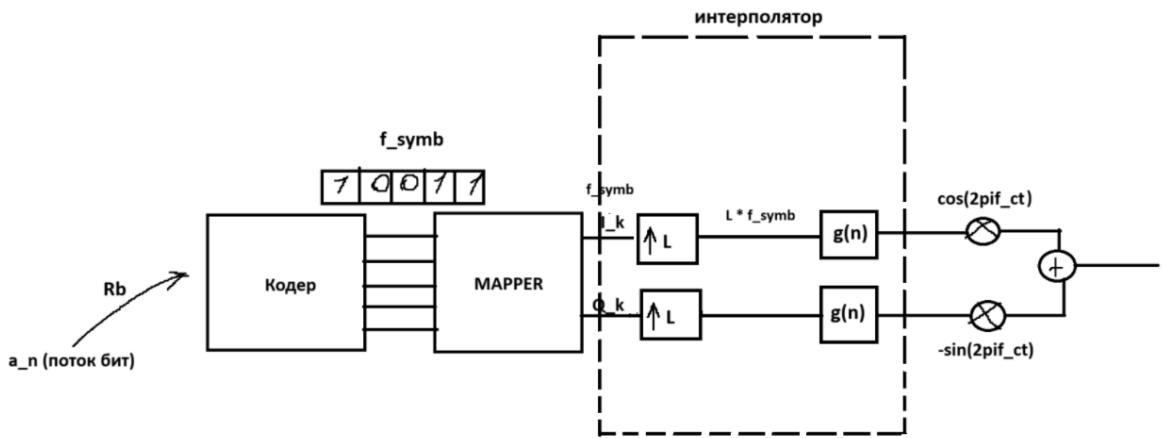


Рисунок 1 — Визуализация архитектуры передатчика

Техническая реализация Upsampling

Техническую реализацию проще будет показать на примере.

Пусть на выходе маппера мы получили $I = [1, -1, 1]$, и хотим, чтобы каждый символ длился $L = 4$ семпла

Сформируем новую последовательность $X(n)$, в которой между каждым I_n добавим $L-1$ нулей.

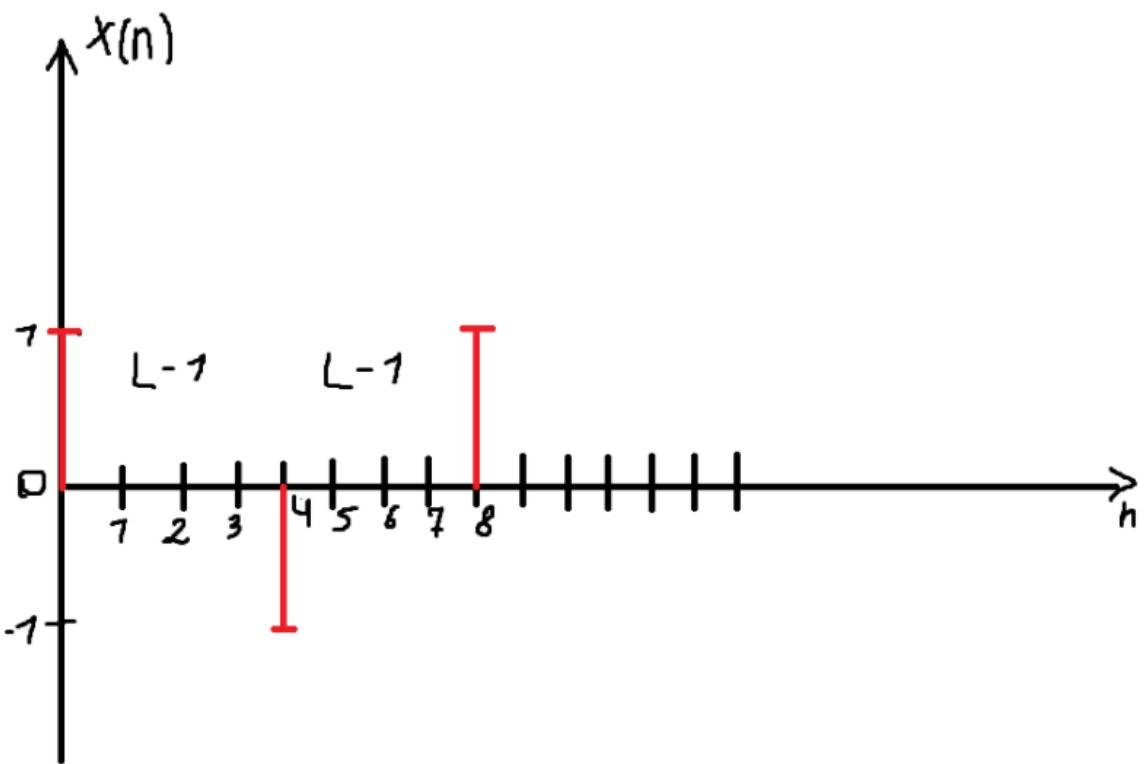


Рисунок 2 — Визуализация работы Upsampling

Получаем последовательность длиной 12 отсчетов (после последнего символа тоже идет 3 нуля).

Формирующий фильтр

На данный момент мы только "растянули" символы, но не придали им никакой формы. Эти действия выполняет формирующий фильтр (на схеме $g(n)$).

В блоке $g(n)$ происходят следующие расчеты: $S(n) = \sum_{m=0}^{L-1} X(m)g(n - m)$, где $X(n)$ - отсчеты, $g(n)$ - импульсная характеристика фильтра. Сама формула это дискретная свертка.

Импульсная характеристика фильтра имеет сложную форму, которая позволяет сделать сигнал любой формы.

Зададим форму импульсной характеристики. Для упрощения возьмем прямоугольную форму.

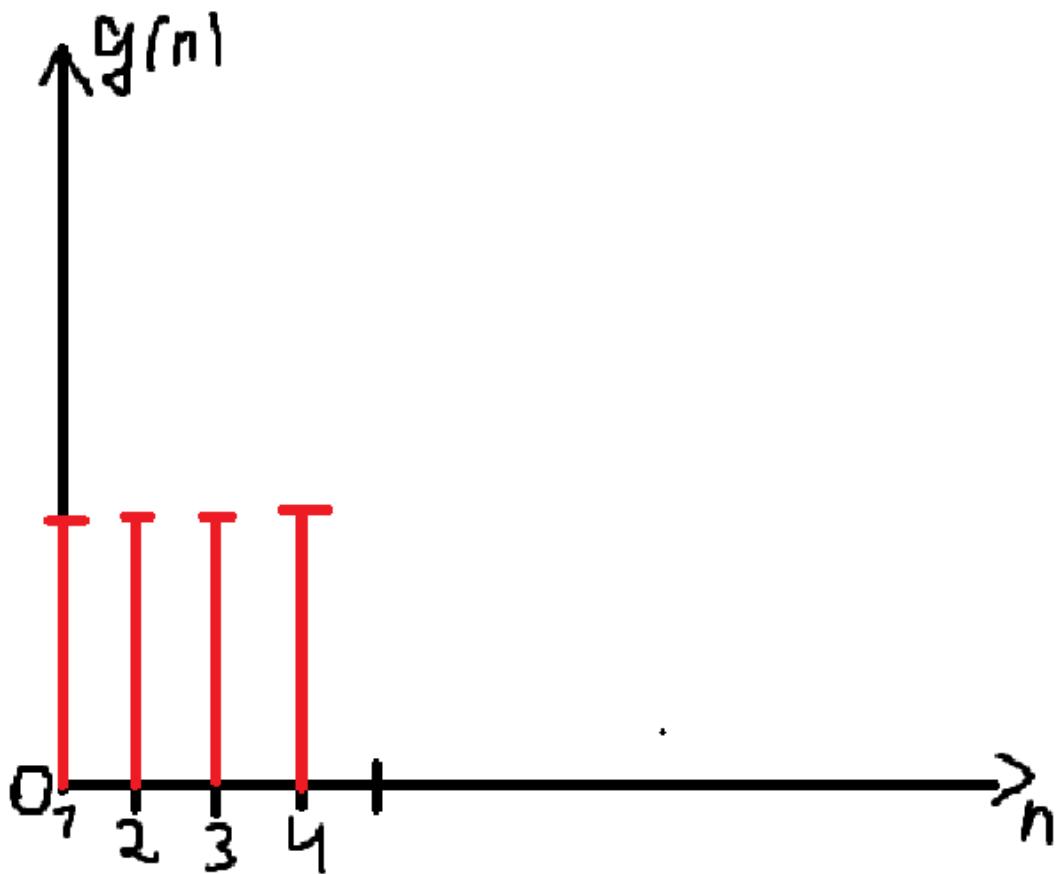


Рисунок 3 — Пример импульсной характеристики

Для иллюстрации работы фильтра произведем вычисления по формуле выше:

$$S(0) = X(0)g(0) = 1 * 1 = 1$$

$$S(1) = X(0)g(1) + X(1)g(0) = 1 * 1 + 0 * 1 = 1$$

$$S(2) = X(0)g(2) + X(1)g(1) + X(2)g(0) = 1 * 1 + 0 * 1 + 0 * 1 = 1$$

$$S(3) = X(0)g(3) + X(1)g(2) + X(2)g(1) + X(3)g(0) = 1 * 1 + 0 * 1 + 0 * 1 + 0 * 1 = 1$$

$$S(4) = X(0)g(4) + X(1)g(3) + X(2)g(2) + X(3)g(1) + X(4)g(0) = 1 * 0 + 0 * 1 + 0 * 1 + (-1 * 1) = -1$$

$$S(4) = X(0)g(5) + X(1)g(4) + X(2)g(3) + X(3)g(2) + X(4)g(1) + X(5)g(0) = \\ 1*0 + 0*0 + 0*1 + 0*1 + (-1 * 1) + 0 * 1 = -1$$

Визуализируем символы после выхода из фильтра:

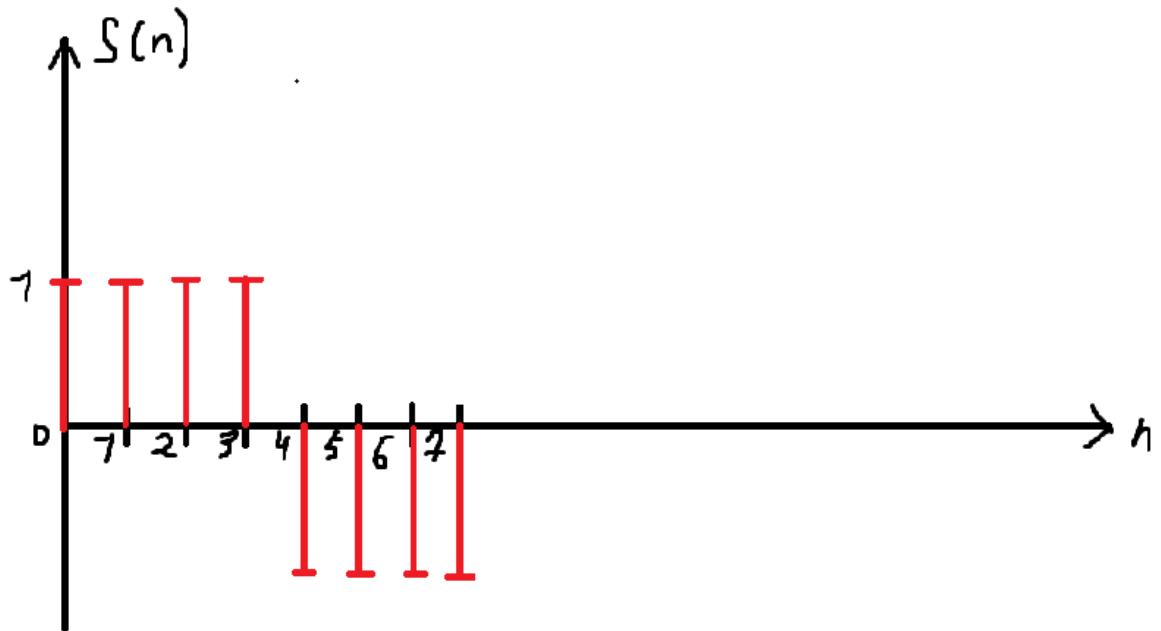


Рисунок 4 — Пример символов после выхода из фильтра

Получили растянутые во времени I и Q с прямоугольной формой. Таким образом можно задать сигналу любую форму.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Введение

На этом занятии отправим в радиоканал мелодию, а потом примем ее и попытаемся воспроизвести.

Конверторы

Для отправки мелодии необходимо конвертировать ее из .mp3 в .pcm формат (поток семплов), а потом после принятия конвертировать обратно из .pcm в .mp3. Для этого нам понадобятся конверторы.

Из .mp3 в .pcm

```
import numpy as np
import librosa
from pydub import AudioSegment

mp3_file = "audio_test.mp3"
pcm_file = "audio_bin.pcm"

# mp3 to pcm
y, sr = librosa.load(mp3_file, sr=44100, mono=True)

pcm_data = (y * 32767).astype(np.int16)

pcm_data.tofile(pcm_file)
```

В переменной `y` хранится массив отсчетов песни, где каждый отсчет принимает значения [-1;1]. `sr` - частота дискретизации. Файл .pcm хранит амплитуды как целые числа от -32768 до 32767, поэтому отмасштабируем значения, умножив их на 32767.

Из .pcm в .mp3

```
import numpy as np
```

```

import librosa
from pydub import AudioSegment

pcm_file = "audio_bin.pcm"
mp3_file = "audio_from_pcm.mp3"

pcm_data = np.fromfile(pcm_file, dtype=np.int16)

audio = AudioSegment(
    data=pcm_data.tobytes(),
    sample_width=2,      # 2      = 16
    frame_rate=44100,    #
    channels=1           #
)

audio.export(mp3_file, format="mp3", bitrate="192k")

```

Считываем из .pcm файла отсчеты, с помощью функции AudioSegment формируем аудиофайл, а потом сохраняем файл в текущей директории.

Отправка и прием мелодии

Чтение .pcm файла в C++

Чтобы отправить семплы из .pcm файла, нужно для начала считать файл. Для этого напишем функцию

```

int16_t *read_pcm(const char *filename, size_t *sample_count)
{
    FILE *file = fopen(filename, "rb");

    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    fseek(file, 0, SEEK_SET);
    printf("file_size = %ld\n", file_size);
    int16_t *samples = (int16_t *)malloc(file_size);

    *sample_count = file_size / sizeof(int16_t);

    size_t sf = fread(samples, sizeof(int16_t), *sample_count,
                      file);

```

```

if (sf == 0){
    printf("file %s empty!", filename);
}

fclose(file);

return samples;
}

```

Передаем в функцию имя .pcm файла и указатель на переменную, в которой потом вернется кол-во считанных семплов. Далее с помощью fseek(file, 0, SEEK_END) перемещаемся в конец файла, а с помощью ftell(file) узнаем текущую позицию в байтах, т.е фактически узнаем размер файла. Далее выделяем память под массив с семплами, вычисляем кол-во семплов и считываем их из файла.

Отправка и прием

```

size_t sample_count = 0;
int16_t *samples = read_pcm(PATH_TO_AUDIO, &sample_count);

for (size_t offset = 0; offset < sample_count; offset += 1920 *
    2)
{
    if(offset + 1920 * 2 >= sample_count)
        break;

    void *tx_buffs[] = {samples + offset};
    fwrite(samples + offset, 2 * rx_mtu * sizeof(int16_t), 1,
        tx_data);
    printf("offset: %d", offset);
    flags = SOAPY_SDR_HAS_TIME;
    int st = SoapySDRDevice_writeStream(sdr, txStream, (const
        void * const*)tx_buffs, tx_mtu, &flags, tx_time,
        timeoutUs);
    if ((size_t)st != tx_mtu)
    {
        printf("TX Failed: %in", st);
    }
}

```

}

Переменная `samples` содержит семплы, считанные из `.pcm` файла. Далее запускаем цикл, где будем итерироваться по сдвигам от 0 до `sample_count` с шагом $1920 * 2$ (потому I и Q занимают 2 байта). Сдвиг нужен потому, что за раз отправить мелодию мы не можем, поскольку размер отправляемого буфера должен составлять 1920 семплов. Далее сделаем проверку на выход за границы массива, чтобы не возникало ошибок (часть семплов срежется, но на качество это не влияет). Далее формируем `tx_buffs` как `samples + offset`, т.е каждый раз будем перемещаться по массиву и отправлять следующий блок данных. Прием данных остался неизменным. После выполнения программы получим семплы мелодии, принятой из радиоканала. Далее конвертируем `.pcm` файл в `.mp3` и наслаждаемся мелодией.

Эксперимент

SDR всех студентов работают на одной частоте, соответственно они создают помехи друг для друга. Возьмем разные мелодии, поставим SDR рядом и запустим отправку.

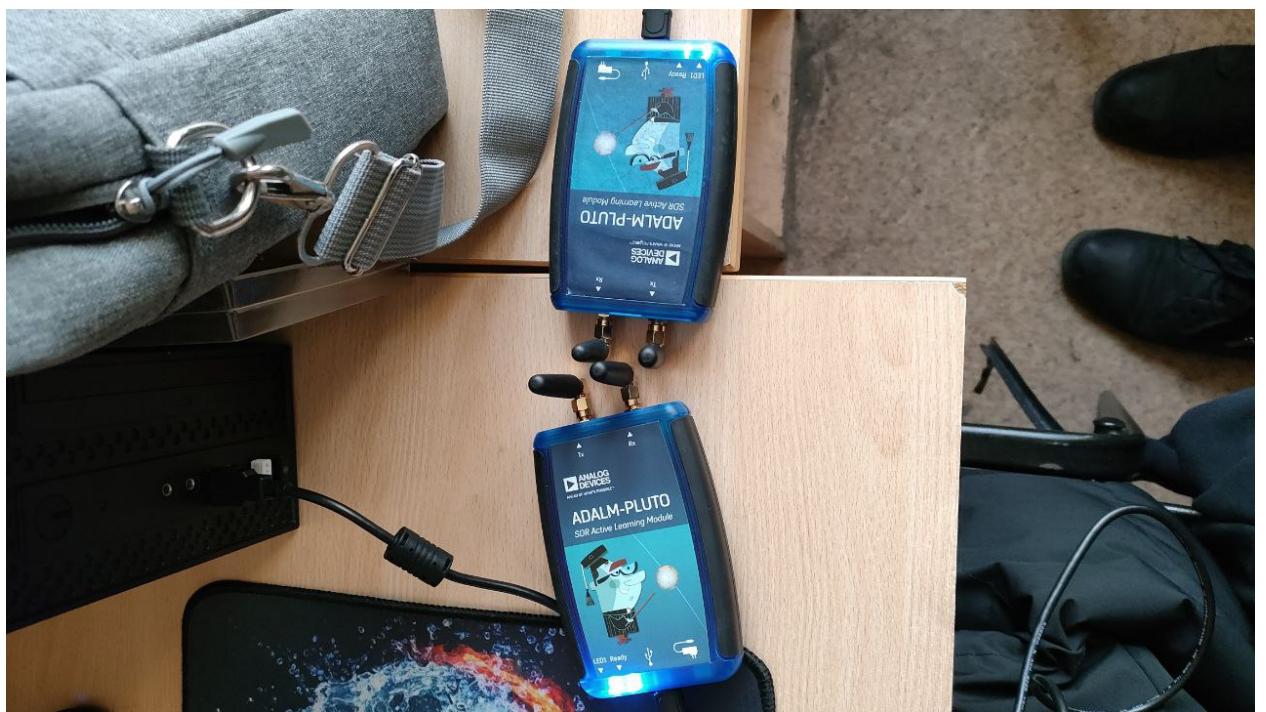


Рисунок 5 — Эксперимент

Итоговая мелодия одновременно похожа на две мелодии с некоторыми помехами. Этот пример иллюстрирует недостаток систем аналоговой связи - она подвержена интерференции от других сигналов.

Реализация логики формирующего фильтра

В теории был указан корректный метод для формирования дляящихся символов $I(t)$ и $Q(t)$ из символов I и Q . Реализуем эту логику на Python:

```
# samples on symbol
L = 1000

# samples
I_upsampling = []
Q_upsampling = []

# upsampling for In-phase component
for x in I_symbols:
    I_upsampling.append(x)
    I_upsampling.extend([0] * (L-1))

# upsampling for Quadrature component
for x in Q_symbols:
    Q_upsampling.append(x)
    Q_upsampling.extend([0] * (L-1))

# set impulse response (rect)
g = [1] * L

s_I = []
s_Q = []

# compute convolution
for n in range(len(I_upsampling)):
    tmp_I = 0
    tmp_Q = 0
    for m in range(L):
        if n - m >= 0:
            tmp_I += I_upsampling[n-m]*g[m]
            tmp_Q+= Q_upsampling[n-m]*g[m]
    s_I.append(tmp_I)
    s_Q.append(tmp_Q)
```

```
s_Q.append(tmp_Q)
```

Считываем и парсим из файла символы I и Q. После этого повышаем их частоту дискретизации путем формирования нового списка символов с добавлением L-1 нулей после каждого. Далее устанавливаем импульсную характеристику сигнала g. Далее вычисляем свертку. В переменных s_I и s_Q находятся символы I(t) и Q(t), растянутые во времени.

Для проверки корректности работы визуализируем полученные результаты. Если всё верно, то графики должны быть идентичны графикам из предыдущего занятия.

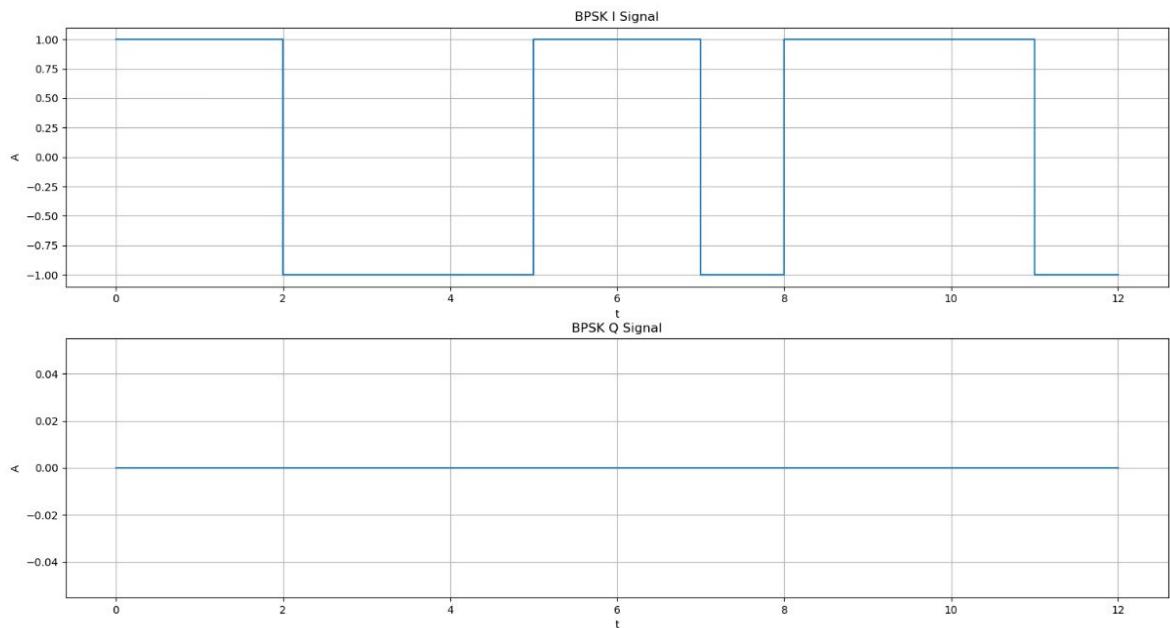


Рисунок 6 — Символы во времени (BPSK)

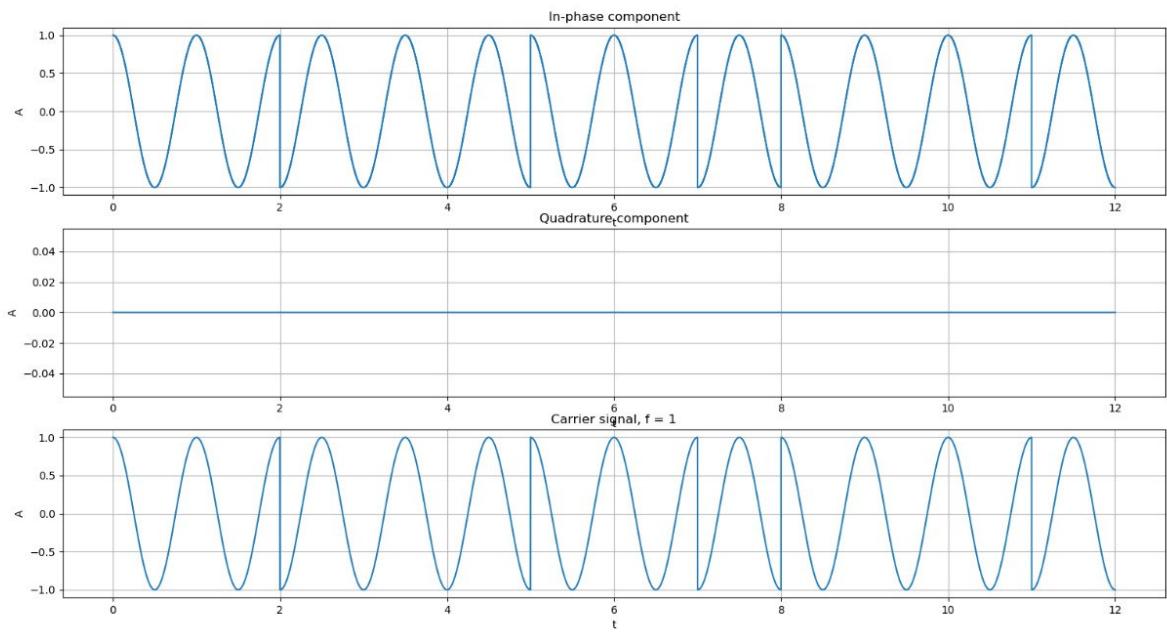


Рисунок 7 — Несущее колебание, перемноженное на символы (BPSK)

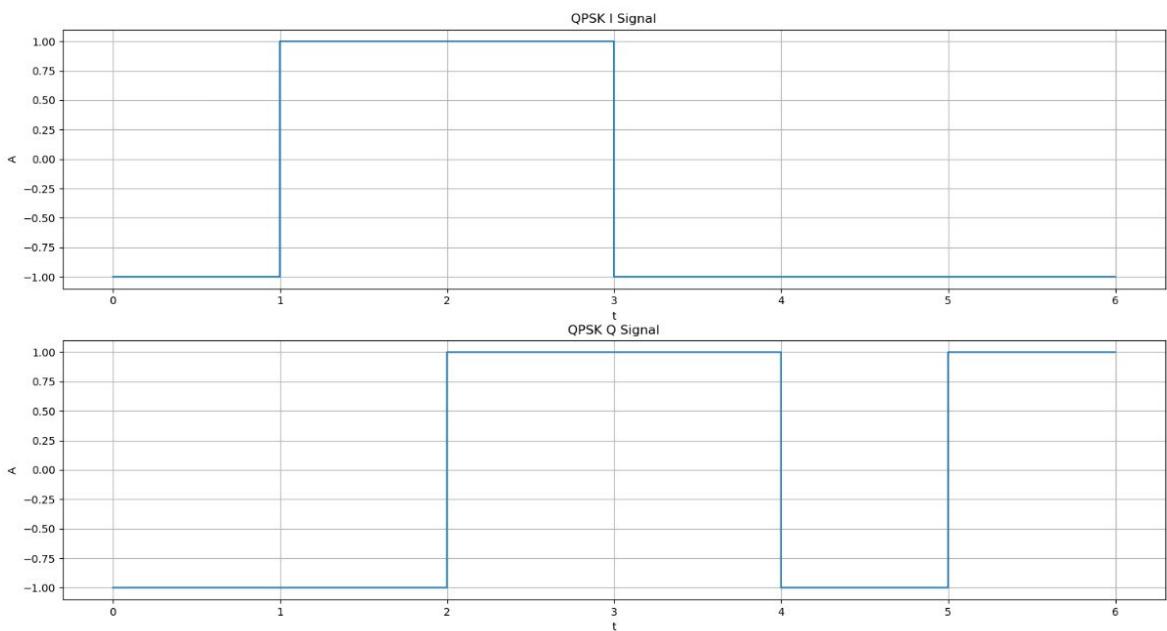


Рисунок 8 — Символы во времени (QPSK)

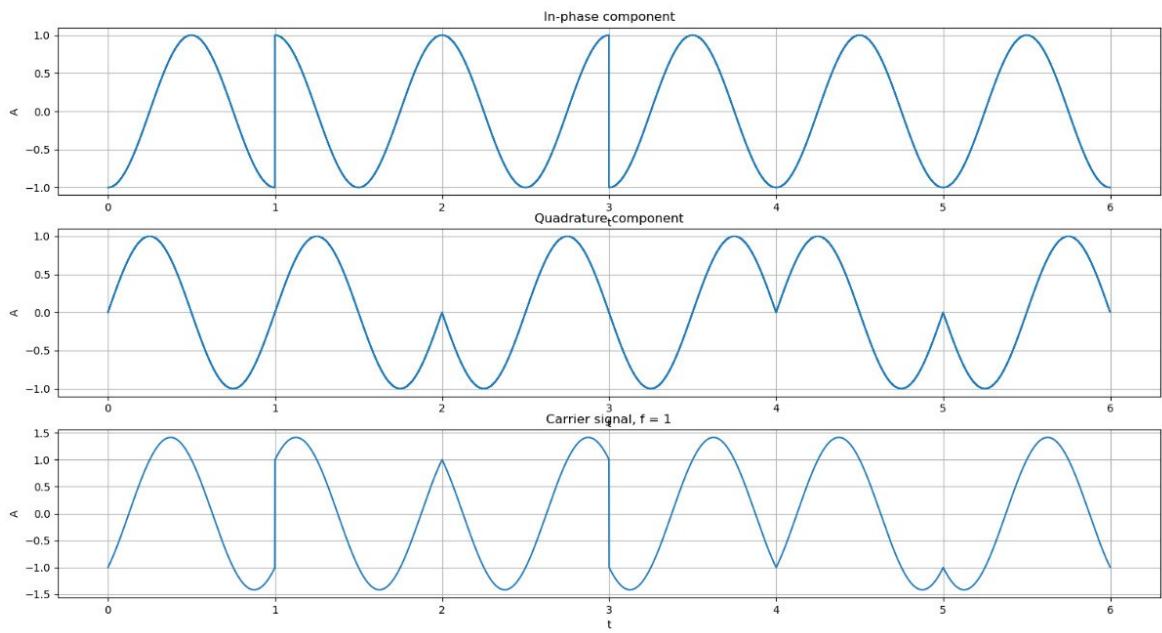


Рисунок 9 — Несущее колебание, перемноженное на символы (QPSK)

Графики идентичны графикам из прошлого занятия, значит, все работает верно. Таким образом теперь мы можем задать сигналу любую форму, изменения импульсную характеристику g .

ВЫВОД

В ходе проделанной работы я углубился в работу формирующего фильtra и программно реализовал его логику, а также отправил мелодию в радио-канал, принял ее и воспроизвел.