SDR OFDM Frame Generation according to IEEE 802.22

OFDMA modulation scheme for sub-GHz band

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Telecommunications

by

Bc. Marek Honek
Registration Number 12024729

to the Faculty of Electrical Engineering and Information Technology

at the TU Wien

Advisor: Univ.Prof. Dr.-Ing. Christoph Mecklenbräuker

Assistance: Dipl.-Ing. Bernhard Isemann

Vienna, 11th August, 2022

Marek Honek　　　　Christoph Mecklenbräuker

TECHNISCHE UNIVERSITÄT WIEN

institute of telecommunications
Technische Universität Wien
Gußhausstraße 25/E 389
1040 Wien, Austria

Meldung einer Masterarbeit

Studierender

Matrikelnummer:	12024729
Familienname:	Honek
Vorname:	Marek

Studium

Typ:	Master
Studium:	066507 Telecommunications

Arbeit

Titel DE:	Software-defined Radio OFDM Frame Generation according to IEEE 802.22
Titel EN:	Software-defined Radio OFDM Frame Generation according to IEEE 802.22
Institut:	E389 - Institute of Telecommunications

Betreuung

Hauptbetreuung:	Univ.Prof. Ing. Dipl.-Ing. Dr.-Ing. Christoph Mecklenbräuker

7.7.2021 Marek Honek
Datum, Unterschrift Studierender

7.7.2021 [illegible]
Datum, Unterschrift Betreuer

7.7.2021 [illegible]
Datum, Unterschrift Studiendekan_in

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Telecommunications**

Department of Radio Electronics

Student:	Bc. Marek Honek	***ID:***	191847
Year of study:	2	***Academic year:***	2021/22

TITLE OF THESIS:

Software-defined radio OFDM frame generation according to IEEE 802.22

INSTRUCTION:

The work shall contribute to the setup of a new digital transmission procedure in the frequency band 50-54 MHz (6m wavelength) by implementing an OFDM frame in software defined radio in C++ and the Liquid-Library. The radio frontend is LimeSDR with a low-power power amplifier. OFDM frame parameters are configurable in software and follow the IEEE 802.22 ("Wireless Regional Area Networks") specifications very closely with small required changes.

There are 14 different PHY modes in IEEE 802.22 and four different lengths of cyclic prefix. The thesis investigates bit error ratio performance of coded OFDM frames over AWGN channels and compares selected simulation results with experimental measurements.

RECOMMENDED LITERATURE:

[1] BANACIA, A. S., GELU, Q. P. , A simplified IEEE 802.22 PHY layer in Matlab-Simulink and SDR platform, International Conference on Electronics, Information, and Communications (ICEIC), 2016, pp. 1-4, doi: 10.1109/ELINFOCOM.2016.7562988.

[2] TOH, K. et al, A physical layer implementation of IEEE 802.22 prototype, 18th IEEE International Conference on Networks (ICON), 2012, pp. 304-308, doi: 10.1109/ICON.2012.6506574.

Date of project specification:	11.2.2022	***Deadline for submission:***	11.8.2022

Supervisor: prof. Ing. Roman Maršálek, Ph.D.
Consultant: Dr. Ing. Christoph Mecklenbräuker

prof. Dr. Ing. Zbyněk Raida
Chair of study program board

HONEK, Marek. Software-defined radio OFDM frame generation according to IEEE 802.22. Brno, 2022. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/142583. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky. Vedoucí práce Roman Maršálek.

Declaration of Authorship

Bc. Marek Honek

I hereby declare that I have written this Diploma Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 11th August, 2022

Marek Honek

Acknowledgements

I am very grateful to Christoph Mecklenbräuker for his willingness to help in writing this thesis and also to Bernhard Isemann for his persistent and intensive technical support.

I also thank Roman Maršálek a Michal Kubíček for help with administration and proofreading.

This work was supported by the Internal Grant Agency of Brno University of Technology, project no. FEKT-S-20-6325.

Abstrakt

Tato práce navrhuje nový radio komunikační standard Amateurradio Wireless Regional Area Network (AWRAN) určený pro přístup k radioamatérké síťi Highspeed Amateurradio Multimedia NETwork (HAMNET) která nevyžaduje volný line-of-sight (LOS) k přístupovému bodu. Dále se práce popisuje prototypovou implementaci zjednodušených rámců tohoto standardu na software-defined radio (SDR) a testovací aparaturou, která je používána pro tyto účely.

V první kapitole je představen standard IEEE 802.22 Wireless Regional Area Network (WRAN), ze kterého AWRAN vychází. Jsou popsány pojmy jako Orthogonal Frequency Division Multiplex (OFDM) a Orthogonal Frequency Division Multiple Access (OFDMA), výhody a způsoby jejich použití. Dále jsou zde popsány také modulační techniky phase-shift keyinq (PSK) a quadrature amplitude-shift keying (QAM), kódování pro detekci a opravu chyb přenosu a zdroje těchto chyb.

V druhé kapitole je zevrubný popis parametrů navrhovaného standardu AWRAN. Jsou popsány OFDM parametry, struktura rámců a superrámců. Je popsán způsob co-existence více buněk AWRAN a přesná podoba čtrnácti řídících zpráv, které umožňují efektivní rozdělení zdrojů přenosového pásma mezi až čtyři základnové stanice a až 63 uživatelů pripojených ke každé z těchto stanic.

Třetí kapitola popisuje strukturu zjednodušeního rámce, který byl implementován na testovací aparatuře RPX-100. Popisuje i aparaturu samotnou a její základní stavební hardwarové prvky i softwarovou knihovnu, která byla pro implementaci zjednodušeného rámce použita. V této kapitole je popsán i připravený testovací spoj a teoretický výpočet útlumu tohoto spoje.

Čtvrtá kapitola se zabývá programy, které byly v rámci této práce vytvořeny. Dva z nich sloužily pro první seznámení s hardwarem a softvarovými knihovnami. Další program slouží jako generátor dříve popsaných zjednodušených rámců, které následne odesílá za použití SDR. Ten stejný program slouží i jako příjmač těchto rámců. Poslední připravený program provádí simulaci bezdrátového přenosu pomocí matematického modelu kanálu. Dále počítá závislost bit error rate (BER) na signal-to-noise ratio (SNR) u takto simulovaného přenosu.

V poslední kapitole jsou vypsány funkce, které byly nadefinovány a použity v programech popsaných v předchozí kapitole.

Klíčová slova

WRAN; AWRAN; HAMNET; SDR; OFDM; komunikační schéma; vysílač

Kurzfassung

In dieser Arbeit wird ein neues Funkübertragungsverfahren Amateurradio Wireless Regional Area Network (AWRAN) für den Zugang zum Amateurfunknetz Highspeed Amateurradio Multimedia NETwork (HAMNET) vorgeschlagen, das keine direkte Sichtverbindung zum Zugangsknoten erfordert. Des Weiteren dokumentiert und diskutiert diese Arbeit eine prototypische Implementierung von vereinfachten Datenrahmen (engl: frames) dieses Übertragungsprotokolls mittels software-defined radio (SDR) auf den dafür verwendeten Testgeräten.

Im ersten Kapitel wird der Standard IEEE 802.22 Wireless Regional Area Network (WRAN) vorgestellt, auf dem AWRAN basiert. Begriffe wie OFDM und OFDMA werden beschrieben, ebenso wie ihre Vorteile und Anwendungen. Beschrieben werden auch die Modulationsverfahren phase-shift keyinq (PSK) und quadrature amplitude-shift keying (QAM), die Kodierung zur Erkennung und Korrektur von Übertragungsfehlern sowie die Fehlerquellen.

Das zweite Kapitel enthält eine detaillierte Beschreibung der Parameter des vorgeschlagenen AWRAN-Protokolls. OFDM-Parameter, Rahmenstruktur und Superframes werden beschrieben. Die Methode der Koexistenz mehrerer AWRAN-Zellen und die genaue Form der vierzehn Kontrollnachrichten, die eine effiziente Zuweisung von Bandbreitenressourcen zwischen bis zu vier Basisstationen und bis zu 63 mit jeder dieser Stationen verbundenen Benutzern ermöglichen, werden beschrieben.

Der dritte Abschnitt beschreibt die Struktur des Vereinfachungsrahmens, der auf dem RPX-100 Testbed implementiert wurde. Er beschreibt auch das Gerät selbst und seine grundlegenden Hardware-Bausteine sowie die Software-Bibliothek, die zur Implementierung des vereinfachten Rahmens verwendet wurde. Dieses Kapitel beschreibt auch die vorbereitete Teststrecke und die theoretische Berechnung der Dämpfung dieser Strecke.

Das vierte Kapitel befasst sich mit den Programmen, die im Rahmen dieser Arbeit entwickelt wurden. Zwei von ihnen wurden für eine erste Einführung in die Hardware- und Softwarebibliotheken verwendet. Ein weiteres Programm dient als Generator der zuvor beschriebenen vereinfachten Rahmen, die es dann unter Verwendung von SDR sendet. Das letzte vorbereitete Programm führt eine Simulation der drahtlosen Übertragung unter Verwendung eines mathematischen Modells des Kanals durch und berechnet darüber hinaus die Abhängigkeit von bit error rate (BER) von signal-to-noise ratio (SNR) für die so simulierte Übertragung.

Im letzten Abschnitt werden die Funktionen aufgeführt, die im vorherigen Abschnitt definiert und verwendet wurden.

Schlüsselwörter

WRAN; AWRAN; HAMNET; SDR; OFDM; Kommunikationsschema; Transceiver

Abstract

This thesis proposes a new radio transmission protocol Amateurradio Wireless Regional Area Network (AWRAN) for access to the amateur radio network Highspeed Amateurradio Multimedia NETwork (HAMNET) that does not require LOS to the access point. Furthermore, the thesis documents and discusses the prototypical implementation of simplified frames of this protocol on software-defined radio (SDR) and the test apparatus used for this purpose.

The first chapter introduces the IEEE 802.22 Wireless Regional Area Network (WRAN) standard on which AWRAN is based. Terms such as Orthogonal Frequency Division Multiplex (OFDM) and Orthogonal Frequency Division Multiple Access (OFDMA) are described, as well as their advantages and applications. There are described the modulation techniques phase-shift keyinq (PSK) and quadrature amplitude-shift keying (QAM), channel coding for error detection and correction, and the sources of these errors.

The second chapter gives a detailed description of the parameters of the proposed AWRAN protocol. OFDM parameters, frame and superframe structures are described. The method of co-existence of multiple AWRAN cells and the exact form of the fourteen control messages that allow efficient allocation of bandwidth resources among up to four base stations and up to 63 users connected to each of these stations are described.

The third section describes the structure of the simplified frame that was implemented on the RPX-100 testbed. It also describes the apparatus itself and its basic hardware building blocks as well as the software library that was used to implement the simplified frame. This chapter also describes the prepared test link and the theoretical calculation of the attenuation of this link.

The fourth chapter deals with the programs that were developed as part of this work. Two of them were done for a first introduction to the hardware and software libraries. Another program serves as a generator of the previously described simplified frames, which are sent using SDR. The same program could be used as well as the receiver of these frames. The last program prepared performs a simulation of the wireless transmission using a mathematical model of the channel. It calculates the dependence of bit error rate (BER) on signal-to-noise ratio (SNR) for the simulated transmission.

The last section lists the functions that were defined and used in programs from the previous section.

Keywords

WRAN; AWRAN; HAMNET; SDR; OFDM; communication scheme; transceiver

Contents

CHAPTER 1

Introduction

HAMNET is an intranet exclusively operated by and for radio amateurs [31] based on directive radio links via LOS propagation. It was initiated in Germany in 2009 and nowadays covers significant parts of Europe. Unfortunately, many radio amateurs are lacking a LOS connection to HAMNET and cannot currently connect. The ultimate goal of the work in this thesis is to design and prototypically implement a modulation and coding scheme that enables access to HAMNET in non line-of-sight (NLOS) conditions for up to fifty users simultaneously which are located at distances up to fifty kilometers from their access point. We have selected three amateur radio frequency bands (50-54 MHz, 144-146 MHz, and 430-440 MHz) for this purpose. The scope of this thesis covers the specification of the modulation and coding scheme for a novel AWRAN, its prototypical implementation for frame generation in the form of a C++ source code and compiled for the single board computer Raspberry Pi [19]. The modulation and demodulation are implemented in SDR developed by Lime Microsystems known as LimeSDR [15] for performance measurement of the frame transmission and reception.

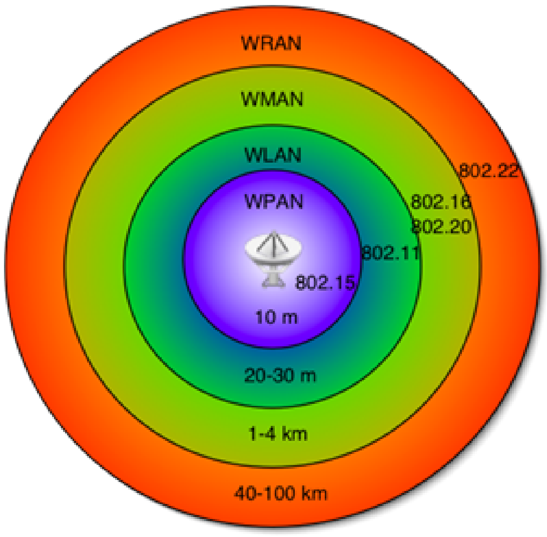


Figure 1.1: Types of networks [27]

1.1 Wireless Regional Area Network (WRAN)

Regional Area Network (RAN) is a communication network that covers an area larger than Metropolitan Area Network (MAN) and smaller than Wide Area Network (WAN), see Figure 1.1.

The standard defining WRAN, IEEE 802.22 [26], was first developed in 2004 to fulfill the requirements of internet access in rural areas with low population density, where it would not be economically feasible to roll out a cable network. The current version is IEEE 802.22-2019 [10].

The WRAN is designed to operate in unused television (TV) channels, so-called *white spaces*. Although spectrum usage by commercial TV broadcast stations is fairly static in the time domain, some changes do occur from time to time. It is not allowed for the WRAN to interfere with TV broadcasting, thus several techniques are specified to prevent this.

On one hand, all devices of the networks have to operate on a fixed geological location and the location must be known by the base station (BS). Each BS has access to a database of information describing the protected broadcast operation in the area.

On the other hand, all wireless devices have to observe the spectrum. If there is observed a presence of analog or digital TV broadcasting or licensed auxiliary device by a customer premises equipment (CPE), the fact is reported to the BS and CPE reduces its own equivalent isotropic radiated power (EIRP) by placing a limit on transmit power control (TPC). BS limits their EIRP also to reduce potential interference. If the reduced power does not allow the proper function of links with distant CPEs, the spectrum manager (SM) initiates a channel move procedure.

1.2 Other sub GHz standards

There are two other standards, namely IEEE 802.11af [4] and IEEE 802.11ah [1], that deal with wireless networks operating under 1 GHz. They enhance the coverage area of Wi-Fi taking advantage of less attenuation and better propagation characteristics of longer wavelengths in comparison to 2.4 and 5 GHz.

The IEEE 802.11af focuses on spectrum sharing of unused TV channels. It has complex architecture consisting of a Geolocation Database, Registered Location Secure Server, Geolocation-Database-Dependent enabling stations and dependent stations as well as complex communication control.

The other one, IEEE 80.11ah is designed to fulfill requirements that came up with Internet of Things (IoT). The operating frequency is considered to be 900 MHz, but re-use of TV white space is possible also.

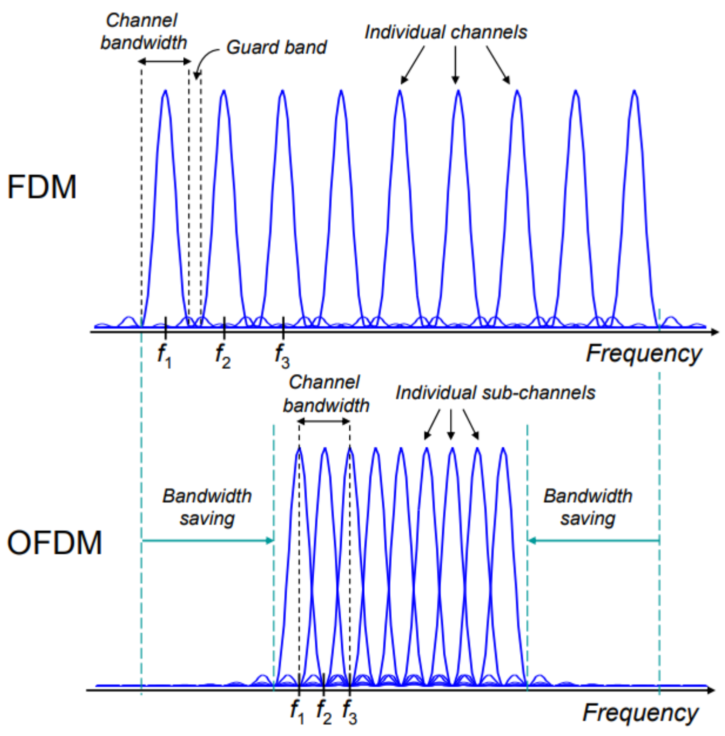


Figure 1.2: FDM vs OFDM [21]

1.3 Orthogonal Frequency Division Multiplex (OFDM)

OFDM is a multicarrier modulation technique that transmits data over several orthogonal carriers simultaneously. Conventional techniques, on the other hand, transmits data using only a single carrier. OFDM is a special case of classical frequency division multiplexing (FDM). The difference is, that OFDM takes advantage of the orthogonality of precisely placed subcarriers. If the difference between each neighboring subcarriers is $\frac{1}{T_{symbol}}$, the subcarriers do not interfere with each other, thus do not require a guard band between them, and the spacing is much lower than in FDM case as you can see on figure 1.2.

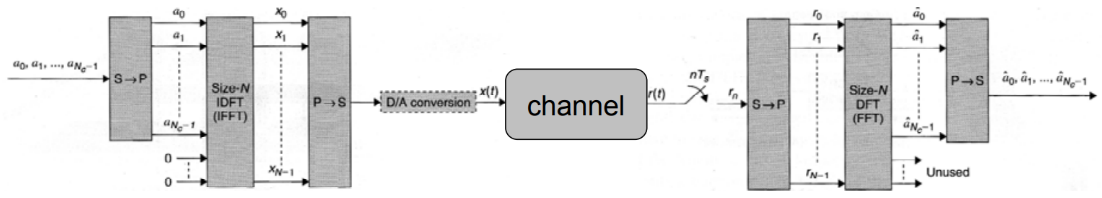


Figure 1.3: OFDM modulation/demodulation using IDFT/DFT [21]

1.3.1 Fourier transform (FT)

Although it may seem that the high number of subcarriers requires the same number of modulators and mixers, there is a simpler and more convenient solution in the usage of inverse fast Fourier transform (IFFT) in the modulator and fast Fourier transform (FFT) in the demodulator as shown on figure 1.3.

A FT is a mathematical function (see eq. 1.1) that transforms other function from it's time domain into a frequency domain, often written as $F(f(t)) = F(\omega)$ where F is a FT operator.

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \tag{1.1}$$

Because the digital systems deal with a discrete values rather than continuous functions, a special cases of FT called discrete Fourier transform (DFT) (see eq. 1.2) is used.

$$A_k = \sum_{n=0}^{N-1} e^{-i\frac{2\pi}{N}kn} a_n \tag{1.2}$$

Computing N-point sequence using a DFT algorithm takes N^2 complex multiplications and additions, however if the number N is power of 2, faster algorithm for calculating DFT could be used. Such algorithm is called FFT and reduces the number of calculations to N*log(N), thus drastically reducing the computational time, especially for large N.

An inverse Fourier transform, as the name suggests, does the inverse operation and OFDM modulator takes advantage of that. For each subcarrier of the currently assembles symbol is defined its amplitude and phase (according to the used modulation) and then the whole symbol is transformed into a time domain then usually mixed with a local oscillator frequency and transmitted. [7]

1.3.2 Cyclic prefix (CP)

OFDM is very useful for reducing trouble with ISI caused by multi-path propagation. If n is the number of subcarriers, each OFDM symbol is n times longer in the time domain than it would be in the single carrier technique. This longer time duration of each symbol allows signals propagating by different paths to arrive at the receiver with a relatively smaller time difference compared to the symbol duration, but ISI is still present as you can see in figure 1.4.

For complete avoidance of ISI, a prefix is added before each symbol. If the prefix is longer than the impulse response of the channel, it acts as a guard interval during which all delayed signals of the previous symbol reach the receiver. There is a need to transmit the symbol even during the prefix to maintain the orthogonality of subcarriers. As shown in figure 1.4, it is done such, that corresponding last part of the symbol is copied before the symbol itself.

1.3.3 Pilot signals

Time and frequency properties need to be recovered in the receiver, for this purpose reference symbols - pilot signals are introduced. Pilots are subcarriers that contain known information and are spread over the OFDM symbol. Pilots often change their location after each OFDM symbol to be present on every subcarrier during several symbols.

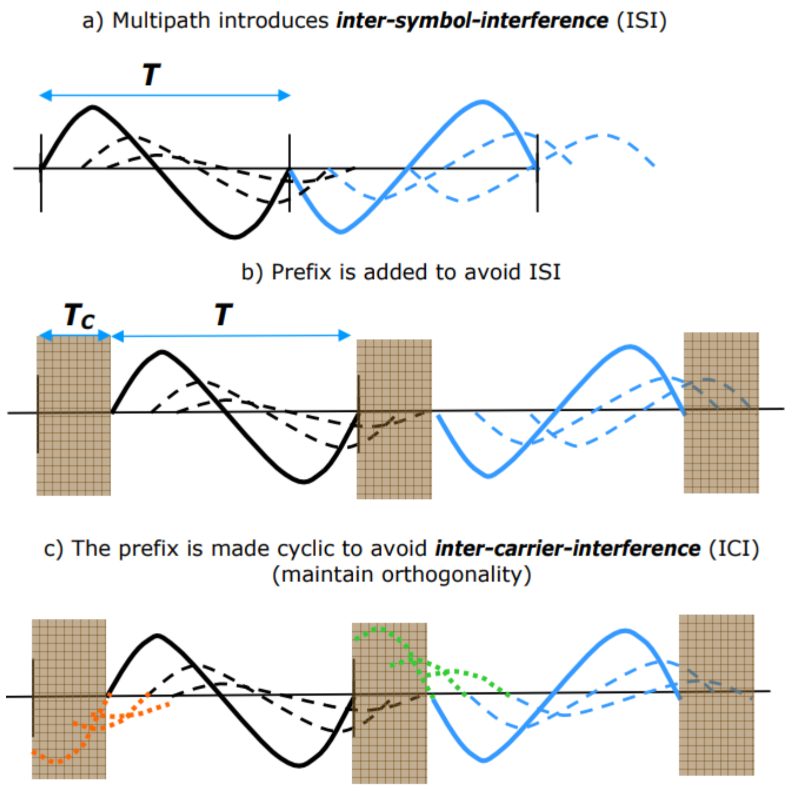

a) Multipath introduces ***inter-symbol-interference*** (ISI)
T
b) Prefix is added to avoid ISI
T_C
T
c) The prefix is made cyclic to avoid ***inter-carrier-interference*** (ICI)
(maintain orthogonality)

Figure 1.4: Multi-path propagation, guard interval and cyclic prefix [21]

1.3.4 Peak to Average Power Ratio

High peak to average power ratio (PAPR) (see fig.1.3) is the major drawback of OFDM signals. Due to the finite output power of amplifiers, the average power (without cutting off the peaks due to hard non-linearity) is always reduced at least by the highest (expected) PAPR of the signal.

Because OFDM systems are sensitive to linearity, class A amplifiers are often used. The output power is reduced even more to not distort the signal by soft non-linearity by operating the amplifiers close to their limits. This results in poor efficiency which is another related problem besides the lowered output power. That affects especially power amplifiers at the transmitter side.

The high PAPR also makes a demand on a resolution of digital to analog converters

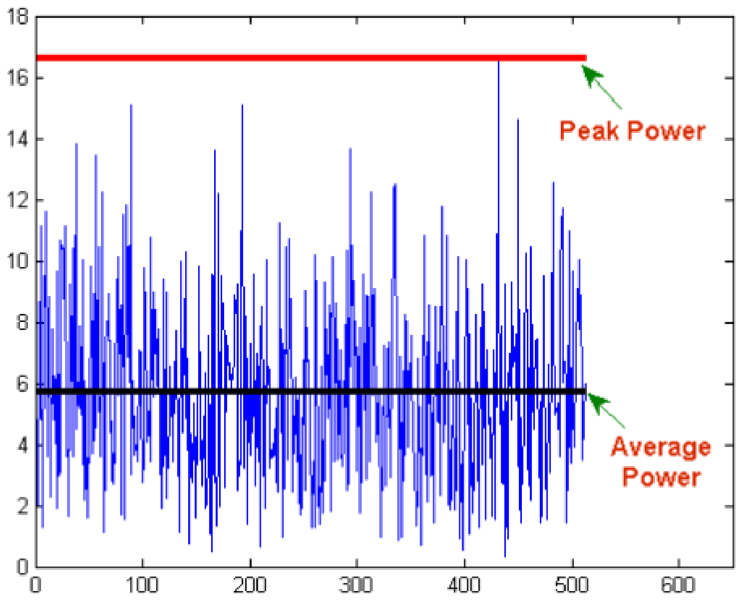


Figure 1.5: peak to average power ratio [24]

(DACs) in the transmitter and analog to digital converters (ADCs) in the receiver. If the resolution were not sufficient, the quantization noise would cause losing information on the non-peak parts of the waveform.

The high peak on the OFDM symbol occurs while suddenly a high number of individual subcarriers constructively interfere. The maximum achievable PAPR is given by formula 1.3.

$$PAPR_{dB} = 10log_{10}(n) + PAPR_{c,dB} \tag{1.3}$$

where n is the number of subcarriers and $PAPR_{c,dB}$ is a the PAPR of each subcarrier, which is 3.01 dB for sine signal.[37]

Several PAPR reducing techniques exist to overcome these problems, eg: clipping ("giving up" the peaks and optimizing for the non-peak parts of the signal), coding schemes, phase optimization, nonlinear companding transforms, tone reservation, tone injection, constellation shaping, partial transmission sequence, selective mapping. [12] [20]

1.4 Orthogonal Frequency Division Multiple Access

Multi-user applications require any multiple access technique. OFDM can be combined with conventional techniques like time division multiple access (TDMA) or frequency division multiple access (FDMA) but the OFDM properties can also be used for multiple access. OFDMA allows to transmit information to different receivers on many orthogonal subcarriers and also receive orthogonal signals from multiple transmitters simultaneously.

The set of orthogonal frequencies can be divided into several blocks, then we talk about consecutive channel multiplexing, or can be interleaved (e. g. first user has assigned odd subcarriers and the second user has assigned even subcarriers), then we talk about distributed channel multiplexing.

Consecutive channel multiplexing has not had such high demands on frequency synchronization as distributed channel multiplexing does, but is much more sensitive to frequency selective fading. [21]

1.5 Code division multiple access (CDMA)

CDMA is a technique that allows the combining of data streams in the same time and frequency band. The advantage over FDMA and TDMA, which distinguish the data streams by frequency or time respectively, is significant especially for mobile applications. The mobile systems introduce variable time delays and Doppler shifts in frequency due to changes in distance between transmitter and receiver.

Standard CDMA works such, that each transmitted symbol is modulated by a chip sequence of +1 and -1 which effectively increase the data rate and thus bandwidth (BW) of the transmitted signal.

Whenever there are more data streams, each modulated by known mutually orthogonal sequences, the receiver can perform a correlation of the received signal with each sequence and thus separate these streams.

These mutually orthogonal sequences are taken from a Hadamard matrix.

1.5.1 Hadamard matrix

The Hadamard matrix is a matrix of orthogonal vectors, or in other words, each line of the matrix has an equal number of matching values and non-matching values (in corresponding columns) with each other line. The matrix could be easily constructed and scaled up using Sylvester's recursive construction algorithm:

When **H** is a Hadamard matrics of order **n**, then

$$\begin{bmatrix} H & H \\ H & -H \end{bmatrix}$$

is a Hadamard matrix of order **2n**. [41]

Thus first three orders are following: $H_1 = \begin{bmatrix} 1 \end{bmatrix}$; $H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$; $H_3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$

1.5.2 Multi Carrier (MC)-CDMA

Multi-carrier version of CDMA is a combination of CDMA with OFDM technique. There is no transmission of the chipped data stream sequentially on a single carrier, but each subcarrier of the OFDM symbol belongs to one symbol of the chip sequence. All the chips are transmitted simultaneously on orthogonal subcarriers such, that a phase shift of 180° is introduced on carriers belonging to -1 of the chip sequence.

All the transmitters then transmit all the data on all the subcarriers using IFFT modulator. The receiver converts the received signal to the frequency domain using FFT and performs correlation with individual chip sequences which separates individual data streams. [42]

1.6 Modulations

Modulation in communication theory is a technique for expressing information by changes of electromagnetic (EM) field. There are baseband and carrier frequency modulations.

When the pulse shape or position carries information, we are talking about baseband modulation. Examples of such modulations could be pulse width modulation (PWM), pulse position modulation (PPM) or pulse code modulation (PCM). Sometimes baseband codes are referred to as modulation, so return to zero (RZ), non-return to zero (NRZ), alternate mark inversion (AMI) or Manchester codes would belong under baseband modulations also.

But we are interested in carrier frequency modulation which is a process of changing a property or multiple properties of a high-frequency sinusoidal signal. The high-frequency signal is called a carrier because enables the possibility of wireless transmission. Its properties are changed by a modulation signal that carries the information.

Examples could be amplitude modulation (AM), frequency modulation (FM), or phase modulation (PM) that uses a continuous (analog) modulation signal, typically

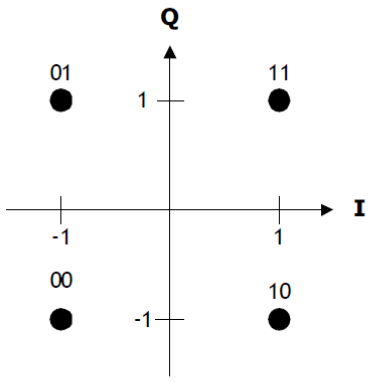


Figure 1.6: QPSK Gray mapping [10]

an output of some transducer (eg. microphone).

Digital carrier frequency modulations work in a similar manner, but there are discrete values of the carrier signal properties. That is why these modulations often (but not always) use "shift-keying" in their name.

1.6.1 Phase-shift keyinq (PSK)

PSK is a modulation technique that represents information in changing phase of the fixed frequency carrier signal. The simplest form is binary phase-shift keying (BPSK) which uses only two discrete phases and hence carries a single bit per symbol. quadrature phase-shift keying (QPSK) (sometimes called 4-PSK) works with four different phases and hence two bits per symbol. To minimize the number of erroneous bits in case wrong reading of the phase Gray mapping of the constellation is used, see fig. 1.6

There are also higher-order PSK modulation like 8-PSK or more complex types such as $\pi/4$-QPSK or differential phase-shift keying (DPSK). [43]

1.6.2 Quadrature amplitude-shift keying (QAM)

This modulation uses two carriers with a fixed matching frequency that have 90° phase difference. The two carriers are referred to as in-phase (I) and quadrature (Q). Each of them is amplitude-shift keying (ASK) modulated and the resultant signal is their combination. Because the amplitude of each carrier is changing (the amplitude could be also negative - or the signal has 180° phase shift in other words), the final signal changes not only its amplitude but also the phase.

This technique allows very high modulation depths. Up to 4096-QAM carrying 12 bits per symbol is currently used. However such high modulation depth requires excellent SNR. Another disadvantage of this technique is high PAPR (see chapter 1.3.4). [17]

To minimize the number of erroneous bits in case wrong reading of the received signal Gray mapping of the constellation is used as well as in the PSK case. Constellation diagram of 64-QAM is on fig. 1.7.

1.7 Signal quality measures

A typical measure of received signal quality is SNR (eq. 1.4), commonly stated in decibels (eq. 1.5).

$$SNR = \frac{P_{signal}}{P_{noise}} \tag{1.4}$$

$$SNR_{dB} = P_{signal,dB} - P_{noise,dB} \tag{1.5}$$

The SNR was originally used for an evaluation of baseband analog signals after demodulation and filtering.

Technically the same equations could be used for the received RF signal on its carrier frequency. Such measure is called the carrier to noise ratio but the term SNR is often used interchangeably. [33]

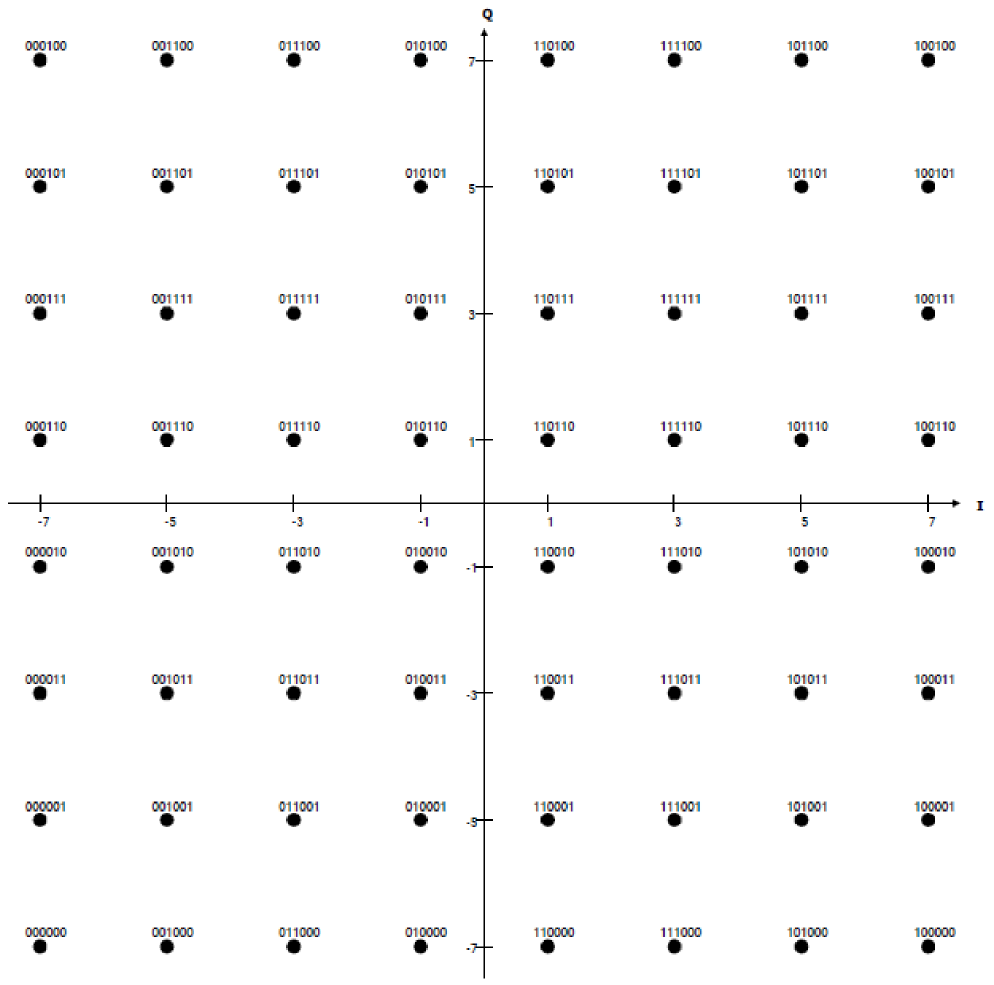


Figure 1.7: 64-QAM Gray mapping [10]

The most useful measure for a digital system is BER.

1.7.1 Bit error rate (BER)

BER is defined as the ratio of received bits with error to the number of all received bits. A similar measure is a frame-error ratio where the same applies to frames with an error and number of all frames.

Theoretical curves for BER are usually plotted on a logarithmic scale versus energy per bit to noise power spectral density ratio (E_b / N_0) (dB), sometimes referred to as normalized SNR or SNR per bit. Because we can only measure SNR directly a conversion that depends on the transmission format such eq. 1.6 is needed.

$$E_b/N_0 = SNR \times c = SNR \times \frac{N_{bits}}{N_{TXsamples}} \tag{1.6}$$

Where the constant c depends on preambles, number of pilots, constellation size (modulation depth), code rate, and duration of the cyclic prefix. Basically, it is a ratio of the useful bit rate to the transmitter's sample rate.

1.8 Forward error correction (FEC)

For the detection of digital data corruption and its correction (whether it happens during transmission over a noisy channel or storing on some unreliable medium), some sort of redundancy - error detection codes (EDC) or error correction codes (ECC) - needs to be added. While the EDC can only bring the information that the received data were corrupted, the ECC can even correct a limited amount of corrupted bits. The use of ECC is mostly referred to as FEC in the data transmission.

Due to FEC the receiver can easily recover corrupted data without the need to send a request for retransmission. That is handy, particularly for simplex links where the request can not be sent at all. Many algorithms exist. They are typically distinguished as block codes and convolutional codes.

FEC enhances the final BER of the radio link for the price of higher BW due to the redundancy. How well the protection works and how much the BW is increased depends on the coding rate. The coding rate is is noted as fraction *k/n* where *k* stands for uncoded message length and n for the coded message length (hence *n-k*

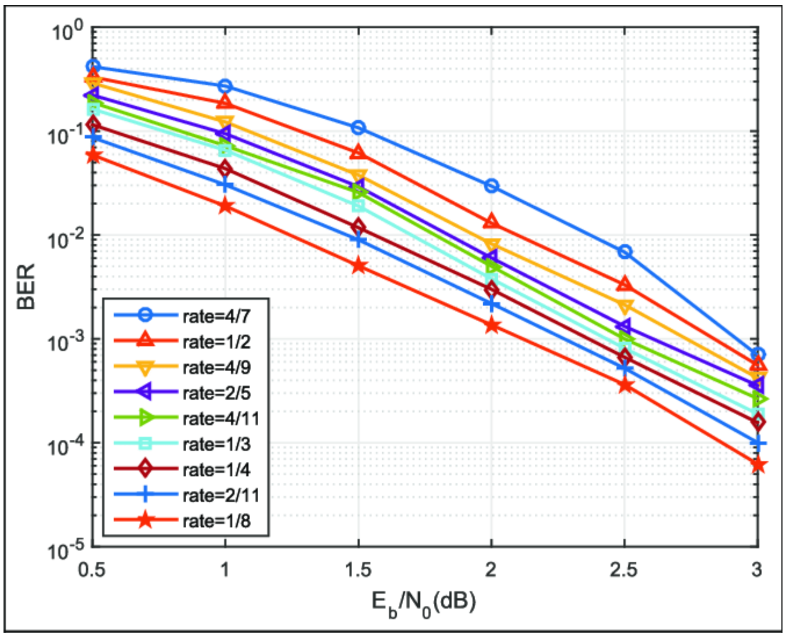


Figure 1.8: FEC impact on BER [3]

is the redundancy). Plotting BER against E_b / N_0 (see fig. 1.8) for different code rates gives a good comparison of their performance. [38]

1.8.1 Block codes

Block codes break the entire data stream into fixed-size messages and handle them independently. Based on the message, the block codes generate the parity bits and create a block with the message and parity combined. For decoding the block codes a hard decision algorithm is typically used.

Richard W. Hamming comes with the idea of block codes back in 1950 to overcome an issue of wrong readings of puncture cards. He proposed a general idea but was specifically focused on Hamming(7,4). This code adds three redundant bits to four message bits allowing single error correction and double error detection. Such code can correct all single-bit errors and detect all double-bit errors. [32]

1.8.2 Convolutional codes

Unlike block codes, convolutional codes slide over the full length of the data stream. The produced parity is also dependent on already encoded data. Convolutional codes are characterized by the coding rate k/n as well as by a memory depth K of the encoder. The memory length describes how many bits are taken into account while the parity is computed.

The best advantage of convolutional codes is maximum-likelihood soft-decision decoding which increases the performance of the coding for correcting burst errors. [36]

1.8.3 Code puncturing

Puncturing is an easy method for increasing the code rate by dropping some of the parity bits. It is convenient to use puncturing while a high code rate is not required because the same decoder can be used for both - punctured or not punctured code. This increases the flexibility of the system without increasing its complexity. [44]

1.9 Wireless Channel

A communication channel could refer to a physical or logical medium used for the transmission of information. The physical case is when the information is represented by changes in a physical quantity. For radio links that transmit over the wireless channel, the physical quantity is a EM field.

In fact EM field is used for carrying data also in wires. Even light is an alternating electromagnetic field so both, fiber and free-space optical links, represent the information using the same physical property as radio links. It is obvious that the basic physical principles must be the same, but each of the named cases faces different challenges.

Although the physical quantity is always continuous in time and magnitude, the transmitted information may be continuous (analog, eg. FM radio broadcast) or discrete (digital). The channel is characterized by its BW in Hertz or by its capacity in bits per second in the case of a digital channel. [28]

The relation between channel BW, SNR and its maximum data rate is given by the Shannon-Hartley theorem, eq. 1.7

$$C = BW \log_2 (1 + SNR) \tag{1.7}$$

where C stands for capacity (data rate) in b/s, BW in Hz, and SNR is in linear form (not in dB).

1.9.1 Attenuation

Attenuation of the signal power is common for all kinds of physical channels. It is rate between received and transmitted power (see eq. 1.8), or their difference on dB scale (see eq. 1.9).

$$A = \frac{P_T}{P_R} \tag{1.8}$$

$$A_{dB} = 10 \log \left(\frac{P_T}{P_R} \right) = P_{T,dB} - P_{R,dB} \tag{1.9}$$

In radio links usually, the main contributor to attenuation is the spreading of the signal power with the square of distance called free space attenuation (FSA) according to eq. 1.10 or 1.11 respectively.

$$A_0 = \left(\frac{4\pi d}{\lambda}\right)^2 \tag{1.10}$$

$$A_{0,dB} = 20\log\left(\frac{4\pi d}{\lambda}\right) \tag{1.11}$$

where d stands for length of the LOS and λ stands for wavelength, both in meters. [40]

The signal experiences attenuation in a wire and antenna itself even before being actually transmitted. The same applies on the receiver side where the power captured by an antenna is attenuated before reaches the receiver's circuitry.

The next kind of attenuation is introduced by obstacles that could be still (such as buildings, trees, or hills) or time-varying. While the still obstacles are usually considered in overall attenuation, the time-varying attenuation is referred to as shadow fading.

In fixed application is convenient to use directive antennas that concentrate the radiated power in one direction (effectively increasing EIRP) or collect more power on receiving side. The directivity D of a non-isotropic antenna is defined as the ratio of its radiation in a given direction over that of an isotropic antenna would have. [22]

Although it does not decrease the attenuation of the link, the result is higher received power without the need to increase transmit power (in comparison to isotropic - omnidirectional antennas). The average received power would be described in equation 1.12.

$$P_{R,dB} = P_{T,dB} - A_{dB} + D_{T,dB} + D_{R,dB} \tag{1.12}$$

where $P_{R,dB}$ and $P_{T,dB}$ are received and transmitted power in dB, A_{dB} is attenuation of all kinds (including attenuation in cables and antennas itself) and $D_{T,dB}$ and $D_{R,dB}$ are directivities of both antennas. [35]

1.9.2 Fading

Fading generally refers to attenuation variation over time or frequency. When the variations occur in time, we talk about fast fading and slow fading - relative to the symbol period. When the variations occur over frequency it is referred to as selective fading, otherwise, it is a flat fading.

The slow fading is commonly caused by shadowing by temporary obstacles. Because slow fading changes attenuation slowly, the channel transfer function could be estimated out of preambles or pilot signals in the receiver.

The fast fading is usually caused by multi-path propagation. Interference of multiple signal images (propagated thru multiple paths) creates places where constructive or destructive interference occurs. Such places are denser the frequency is higher. Receiver propagating thru such an environment experiences rapid (proportionally to movement velocity) changes in received signal power. A low data-rate (narrow band) as well as OFDM systems experience fast fading more likely (compared to high data rate - single carrier systems) due to their long symbol duration.

The selective fading is being caused also by multi-path propagation. The time difference between two signal images causes interference. For frequency f according to equation 1.13 is the interference destructive when n is an odd integer and constructive when n is an even integer.

$$f = \frac{n}{2\Delta t} \tag{1.13}$$

where Δt is the time difference between signal images arrival. [39] [9] [8]

1.9.3 Inter symbol interference (ISI)

ISI occurs when a significant part of symbol energy spreads over time and affects another symbol. Such a thing could happen either by tight filtering or a channel distortion such as multi-path propagation. Time difference between individual paths could be lowered relatively to symbol duration by using the OFDM. For complete avoidance of ISI, guard intervals between symbols are added.

1.9.4 Doppler shift

Doppler effect is a change of frequency induced by the relative motion of receiver and transmitter according to general equation 1.14.

$$f = f_0 \frac{c \pm v_r}{c \pm v_s} \tag{1.14}$$

where f is the observed frequency, f_0 is the original frequency, c is a speed of wave propagation (a speed of light in case of EM waves), v_r and v_s are the velocity of receiver or source respectively.

When the distance between receiver and source is decreasing, the observed frequency is higher than the original and vice versa.

1.9.5 Noise

Noise refers to unwanted energy that modifies the desired signal. There are many types and sources of noise. They can be differentiated into external and internal noises.

Examples of external noises could be atmospheric noise mainly from thunderstorms or man-made noise from wireless transmissions or other electronics.

Examples of internal noise are thermal noise, shot noise, and flicker noise. Quantization noise as a result of analog to digital conversion. The nonlinearity of the signal processing path could be considered a noise source also.[45]

Another differentiation could be done according to the frequency spectrum of the noise. There are narrow band noises such as powerline noise at 50 Hz which is familiar to nearly everybody from audio. Noise with flat spectral characteristics is called white noise (according to white light that contains all frequencies of the visible light, although it does not have a flat characteristic). A lot of broadband noises can be considered as white noise across a finite BW. For those who can not, other color noise models exist. The one whose power decreases with a frequency of 10 dB per decade is called 1/f or pink noise. When it is 20 dB per decade it is 1/f^2 or red noise. On the other hand, noise whose power increases with a frequency of 10 dB per decade is called blue noise. When it is 20 dB per decade it is a violet noise. [34]

A typical measure of broadband noise is a power spectral density N_0 for continuous noise or energy spectral density for impulse noise concentrated in a narrow time window (typically narrower than a symbol duration).[47]

Channels are often modeled with so-called additive white Gaussian noise (AWGN) which approximates various natural sources of noise (distant thunderstorms, thermal noise, shot noise). This noise is added to the signal, has flat spectral characteristics and its magnitude follows a normal distribution with a mean time value at zero. [30]

CHAPTER 2

System model

The new OFDMA communication scheme for accessing HAMNET - Amateurradio Wireless Regional Area Network (AWRAN) is introduced in this chapter.

The new AWRAN scheme is tailored to be used in the following three amateur-radio frequency bands:

- 50–54 MHz (6m band)
- 144–146 MHz (2m band)
- 430–440 MHz (70cm band)

WRAN is designed to operate in the unused portions of the TV broadcast bands: the so-called TV white space which can be anywhere in the VHF/UHF frequency range, 54 MHz and 862 MHz[10]. One frequency band for AWRAN is just below this frequency range. Propagation phenomena and interference scenarios are expected to be practically identical in these cases.

The IEEE 802.22 WRAN was chosen as a baseline for the novel scheme AWRAN. Please, do not confuse with the IEEE 802.22b variant of WRAN called advanced WRAN (A-WRAN).

On the lines below are proposed properties of the new communication scheme Amateurradio Wireless Regional Area Network.

2.1 OFDM parameters

The channel access technique is a OFDMA, so we need to discuss the fundamental properties of OFDM first. The selected parameters are inspired by WRAN and modified to fit the available amateur radio regulations, most importantly the BWs.

The frame duration is fixed at 10 ms. The number of OFDM symbols in one frame varies between 26 and 31 for different lengths of CP. The allowed lengths of CP are $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ and $\frac{1}{32}$ of the FFT length. The cyclic prefix is specified by BS in superframe control header (SCH).

2.1.1 Bandwidth

As stated above, the OFDM properties of WRAN were adjusted to the available BW. Unlike the TV channels, which are 6 (somewhere 7 or 8) MHz wide, the dedicated bands for AWRAN are 2, 4, and 10 MHz wide.

The different BW of AWRAN compared to the original WRAN is achieved by changing the number of subcarriers from 1680 to 560, 1120, or 2800 respectively while the symbol duration remains the same.

Generally is necessary to set real BW of wireless transmission approximately 10 % [21] smaller compared to the dedicated BW to avoid interference with neighboring services.

CP length	number symbols per frame	2 MHz channel BW (MHz)	4 MHz channel BW (MHz)	20 MHz channel BW (MHz)
$\frac{1}{4}$	26	1,820	3,640	9,100
$\frac{1}{8}$	28	1,764	3,528	8,820
$\frac{1}{16}$	30	1,785	3,570	8,925
$\frac{1}{32}$	31	1,780	3,560	8,900

Table 2.1: Supported CPs and corresponding number of symbols in frame and BW [26]

In order to maintain the frame duration of 10 ms with, the number of OFDM symbols roughly compensates for changing CP duration. The fine compensation is achieved by slight BW adjustment for each length of CP.

The equation 2.1 shows calculation of BW really occupied by OFDM with 560 subcarriers, 26 symbols per frame and $\frac{1}{4}$ cyclic prefix.

$$BW_{OFDM} = \frac{n_{subcarriers} \times n_{symbols} \times (1 + CP)}{t_{frame}} = \\ = \frac{560 \times 26 \times (1 + \frac{1}{4})}{10 \text{ ms}} = 1.82 \text{ MHz} \quad (2.1)$$

Since the calculation is for 2 MHz BW, the result is 9 % smaller which is a good starting point. Calculation of BW for all combinations of CP length, BW and corresponding number of symbols are listed in table 2.1.

2.1.2 Pilot signals

The AWRAN scheme consists of 560, 1120, or 2800 subcarriers out of which every seventh is a pilot used for channel estimation, frequency offset estimation, and phase noise estimation.

For better performance, all the pilots move with each OFDM symbol. Their offset follows pattern 0, 3, 5, 1, 4, 6, and 2 for OFDMA symbol indexes 0, 1, ... 6. Hence

PHY mode	modulation	coding rate
1	BPSK	Uncoded
2	QPSK	1/2 and repeat: 3
3	QPSK	1/2
4	QPSK	2/3
5	QPSK	3/4
6	QPSK	5/6
7	16-QAM	1/2
8	16-QAM	2/3
9	16-QAM	3/4
10	16-QAM	5/6
11	64-QAM	1/2
12	64-QAM	2/3
13	64-QAM	3/4
14	64-QAM	5/6

Table 2.2: Supported modulations and coding rates

in seven OFDM symbols every single subcarrier is a pilot once and carries data six times. The OFDMA symbol index is reset to 0 at beginning of each subframe - downstream (DS) and upstream (US).

These pilots carry BPSK modulated pseudo-random binary sequence (PRBS) generated out of fixed seed value. For more details see the detailed description of generator and pilots in WRAN in chapter 9.6.1 of [10].

2.2 Adaptive modulation and coding

AWRAN support several combinations of modulations and coding rates described in table 2.2. These combinations are called physical layer (PHY) modes. PHY modes 3 to 14 can be flexibly chosen for data communication to achieve desire trade-off between data rate and robustness of the system. The PHY mode 1 is used for multi carrier code division multiple access (MC-CDMA) transmission in OW. The PHY mode 2 is used for SCH, FCH and for transmission in SCW.

The PHY mode for data transmission is selected by BS for each CPE individually based on the previous transmission reliability and channel estimation. The selected PHY mode is announced in DS or US map respectively.

2.3 Superframe (SF) structure

SF is 160 ms long and consists of sixteen 10 ms frames. At the beginning of the first frame is a SF preamble followed by the first frame preamble, SCH then FCH and the rest of the first frame. The whole SF and frame structure is shown on figure 2.1.

2.3.1 SF preamble

The first frame of SF starts with SF preamble which contains four repetition of short training sequence (STS). It serves time and frequency synchronization between BS and CPEs. The SF preamble is always sent with CP length $\frac{1}{4}$ of the FFT length.

The STS has length of $\frac{1}{4}$ of the FFT. Four repetition results in one symbol period. The CP effectively adds another full repetition of the STS (a copy of the last one). See fig. 2.2

The OFDM symbol is constructed using IFFT with its standard length depending on the number of subcarriers. The short time duration of the training sequence and its repetitions in the time domain over the symbol duration is achieved by using only one of four subcarriers. It means that between two non-zero subcarriers are three subcarriers with zero amplitude. On the non-zero subcarriers is a BPSK modulated PRBS.

The series is generated by linear feedback shift register (LFSR) with an aim to achieve low PAPR. The generation process and seed sequence for WRAN is

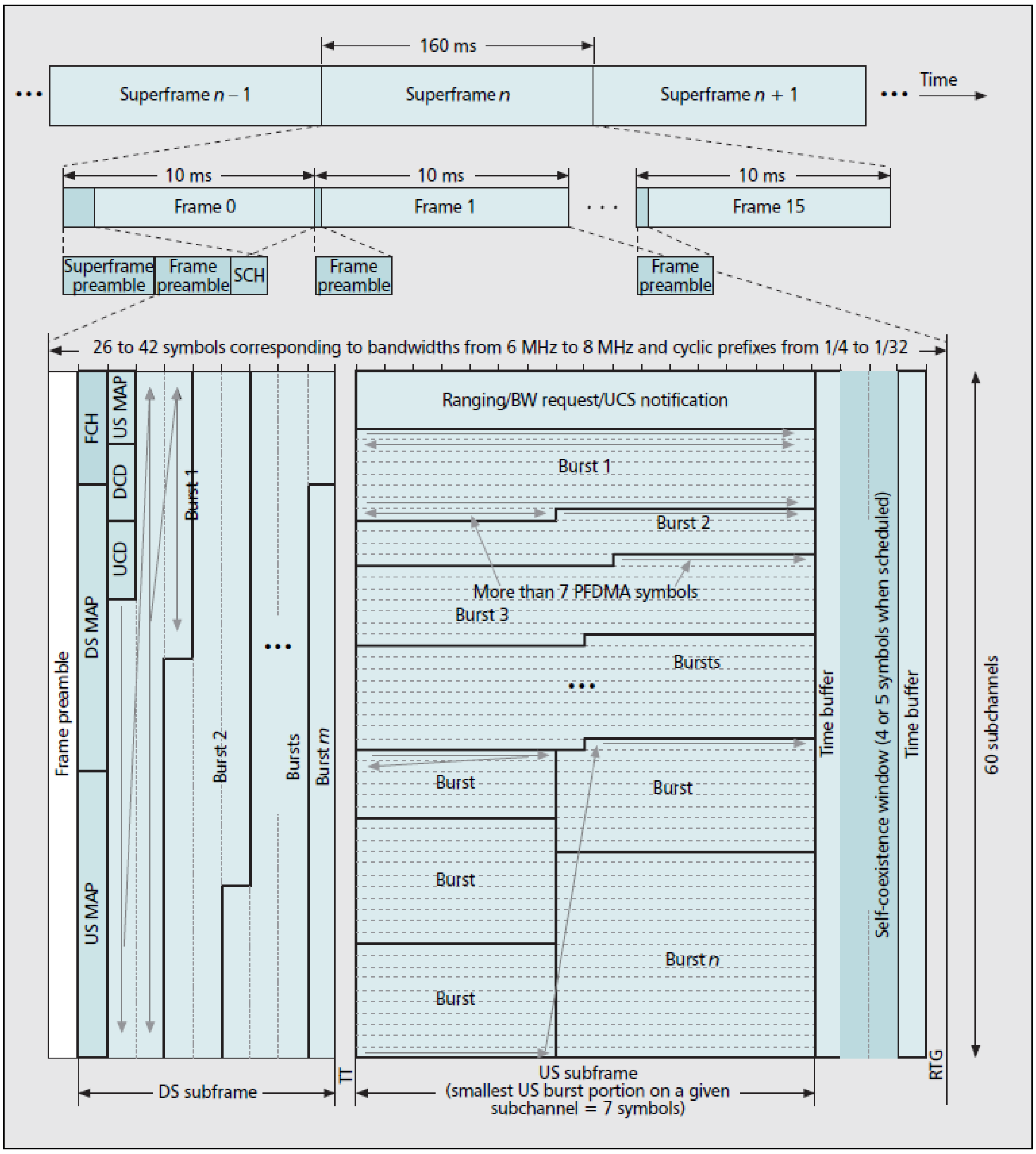


Figure 2.1: SF and frame structure [26]

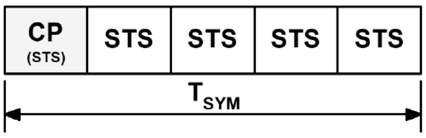


Figure 2.2: SF preamble using four STS and $\frac{1}{4}$ CP length[10]

described in chapter 9.4.1.1.1 of [10].

2.3.2 SF control header

The SCH is always a whole OFDM symbol which is transmitted in the first frame of SF immediately after the frame preamble. Especially in the case of transmission in a 2 MHz bandwidth, one OFDM symbol can not be sufficient. In such cases, the SCH can occupy more OFDM symbols.

The SCH is always modulated and coded according to PHY mode 2 (see Table 2.2) and transmitted with CP length of $\frac{1}{4}$. The exact form of SCH is documented in Table 2.3.

Data	size
BS MAC	48 bits
SF number	8 bits
FA map CPL	2 bits
DCD	(see DCD)
UCD	(see UCD)
SCW-FB	21 bits

Table 2.3: SCH form

2.4 Frame structure

The frame forms the basic transmission unit of this OFDM system. Its time duration is defined to be 10 ms. A group of sixteen frames is called SF and is discussed above.

Only the first frame of a SF starts with a SF preamble and continues with a frame preamble. All other frames start with the frame preamble which consists of two repetitions of long training sequence (LTS) as discussed below.

The remainder of the frame is divided into two parts, which are called the DS and US subframes. These are divided into subchannels in the frequency domain, see details further below. The first symbol (or several symbols) of a DS subframe is the SCH, in the case of the first frame of SF. The rest of the frames start their DS subframes with a FCH followed by FMW and the user DS data.

The US subframe contains users US data. An OW can be scheduled in the US subframe. It is used by CPE for transmitting requests to BS using MC-CDMA management messages. At the end of some DS subframes, can be located SCW which allows other BSs contact the transmitting BS with coexistence request.

2.4.1 Subchannels

The US and DS subframes feature subchannels, which are groups of neighboring subcarriers. The entire OFDM symbol is divided into several subchannels depending on BW of the transmission. Each subchannel is 28 subcarriers wide. Thus the 2 MHz wide channel, which uses 560 subcarriers, is divided into 20 subchannels, in the 4 MHz wide are 40 subchannels and the 10 MHz wide has 100 subchannels.

Each subchannel contains 24 data carriers and 4 pilots. The pilots change their location from symbol to symbol as described above. Subchannels serve as a measure for resource allocation e.g. in downstream channel descriptor (DCD) and upstream channel descriptor (UCD) messages.

2.4.2 Frame preamble

Each frame has its own preamble consisting of two repetitions of LTS used for synchronization, channel estimation, frequency offset estimation, and received power estimation.

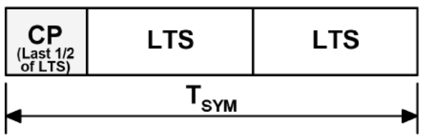


Figure 2.3: frame preamble using two LTS and $\frac{1}{4}$ CP length[10]

The LTS is twice as long as the STS discussed above, thus it is one-half of FFT length. The frame preamble is transmitted as well as the SF preamble BPSK modulated binary sequence with CP length of $\frac{1}{4}$. Thus the OFDM symbol with CP contains two and half LTS. This is illustrated in Fig. 2.3.

The generation with the aim to create low PAPR signal, and LFSR with seed sequence used in WRAN is described in Chapter 9.4.1.3 of [10].

2.4.3 DS subframe

The most significant parts of frames are the DS subframe and US subframe. Their sizes are modified adaptively, to fulfill current needs. The DS subframe starts right after the frame preamble. First OFDM symbol is a SCH in case of first frame of SF, otherwise the first symbol starts with the FCH and continues FMW.

The DS subframe divided into data bursts defined by DCD message. This message is being transmitted by BS in SCH. Or, when the data bursts allocation is being changed during SF, in FMW.

After the last OFDM symbol of DS subframe, the US subframe does not start immediately. A transmit-receive turnaround gap (TTG) with a length of one OFDM symbol is introduced to allow CPE to transmit with proper timing alignment to overcome a problem with the propagation delay of the signal.

Frame control header (FCH)

The FCH is a first part of DS subframe (except the case when SCH is transmitted). It contains frame number, length of the FMW and its PHY mode.

The header is modulated and coded according to PHY mode 2 (see table 2.2).

Data	size	note
frame number	4 bit	
FMW length	8 bit	subcarrier × OFDM symbol
FMW PHY mode	4 bit	the most robust PHY mode used for data in current frame shall be used

Table 2.4: FCH form

Frame management window (FMW)

The FMW is present after each FCH. It contains all the management messages the BS transmits and which are not part of SCH or FCH. Some of these messages (depending on the type) can be addressed to all CPEs (broadcast). Others are addressed to particular CPE which identification (ID) is noted in the first part of the message.

The number of transmitted management messages varies frame to frame, so the length of FMW is noted in FCH. The FMW is placed over the data bursts. If the FMW occupies the whole data burst, the data burst is not assigned to any CPE in the DS map (CPE ID for the occupied data burst is zero). If the FMW occupies only part of a data burst, the data bust is assigned to a CPE and the actual data starts right after the end of the FMW.

DS data bursts

Data bursts are two-dimensional containers for location data in the frame. One dimension is defined by subchannels (groups of neighboring subcarriers) and the other by OFDM symbols.

In DS subframe are data bursts allocated vertically such, that typically all bursts are spread across the whole BW (all subchannels) and each burst occupies just a few OFDM symbols. US bursts are on the other hand allocated horizontally, more about that below.

Vertical allocation of bursts has an advantage, especially in reduced demodulation effort in CPEs. They are not interested in all the OFDM symbols, but only in those which contain preambles, headers, FMW and to them associated burst. Other OFDM symbols can be scratched.

Although after the US subframe is a TTG to absorb a propagation delay, there is another advantage of vertical allocation for more distant CPEs. The earlier sent bursts are associated to them, so they have an additional time buffer for propagation delay.

2.4.4 US subframe

The US subframe comes after DS subframe. There is one symbol long TTG between them which serves as a propagation delay buffer and enables the CPEs to transmit with proper time alignment.

US data bursts

The US data bursts are described by UCD. They are allocated horizontally on the US subframe, unlike the DS bursts which are allocated vertically. It means, that each burst occupies typically the whole length of US subframe and only a few subchannels.

There is a limitation for burst boundary in a subchannel. The minimum number of OFDM symbols occupied at the subchannel can not be smaller than seven symbols for each burst (see figure 2.1). This is because seven symbols are a period of the pilot pattern. So all of the subcarriers in the part of burst are pilot at least once.

The horizontal allocation has a particular advantage for CPEs. Thanks to longer time duration and reduced number of subcarriers of the bursts, the radiated power per subcarrier could be higher with lower EIRP compared to short and wide bursts.

Opportunistic window (OW)

The first twelve subchannels serves as a OW for CPEs to contact the BS. The OWs are scheduled by BS in a US map transmitted during FMW. The length of DS subframe (without SCW) must be at least twelve OFDM symbols when the OW is scheduled.

The CPEs transmits MC-CDMA management messages such BW requests or ranging requests all at once during single OW. The code for data scrambling derived from CPE ID which BS assigns to each CPE during registration process. Subcarriers that would serve as pilot signals (when the OW would not be scheduled) shall remain unused by all CPEs.

Self-coexistence window (SCW)

A SCW could be located at the end of US subframe. It is announced by BS in a SCW frame bitmap transmitted in SCH. The SCW is four OFDM symbols long and needs one symbol before itself and one symbol after itself as a guard interval.

The SCW is dedicated for a BSs of a different AWRANs to transmit coexistence request. The request is modulated according to PHY mode 2.

2.5 Self-coexistence

When there are multiple overlapping AWRANs, some kind of multiplexing needs to be done. When each of them operates in the different frequency band, there is a natural FDM and nothing has to be solved. If they share the same transmission band, a time division multiplexing (TDM) takes the place.

When a BS recognizes another AWRAN transmission on the same band it is about to operate, it shall decode SCH and read SCW frame bitmap management message. This message contains SCW allocation across frames of the SF.

Once a frame with SCW is on schedule, the BS transmits a coexistence request informing the already operating BS about the need for coexistence. This BS responds with a coexistence response message which is sent during SCH. This message contains a BS ID which was associated to the requesting BS. Another message called FA map shall be found in the SCH. This message associates individual frames of the SF to individual AWRANs.

There can be up to four individual AWRANs operating in the same frequency band using this TDM coexistence.

2.6 Management messages

As was mentioned previously, there are so-called management messages in the AWRAN. They are transmitted either by BS to CPEs as a part of a SCH and FMW or by CPE to BS during OW. Coexistence requests are sent by BS that observes the current transmission (in the table 2.5 is referred to as "other BS") to the transmitting BS during SCW. The coexistence response message is sent during the first SCH.

The table 2.5 lists all management messages:

type	message	description	transmit by	transmit to	transmit during
0	DCD	DS channel descriptor	BS	all CPEs	SCH
1	DS-MAP	DS map	BS	all CPEs	FMW
2	UCD	US channel descriptor	BS	all CPEs	SCH
3	US-MAP	US map	BS	all CPEs	FMW
4	RNG-REQ	Ranging request	CPE	BS	OW
5	RNG-CMD	Ranging command	BS	one CPE	FMW
6	REG-REQ	Registration request	CPE	BS	OW
7	REG-RSP	Registration response	BS	one CPE	FMW
8	BW-REQ	BW request	CPE	BS	OW
9	RET-REQ	Retransmission request	BS oe CPE	one CPE or BS	OW or FMW
10	SCW-FB	SCW frame bitmap	BS	all CPEs	SCH
11	CPL	CP length	BS	all CPEs	SCH
12	FA map	Frame allocation map	BS	all CPEs and BSs	SCH
13	CO-REQ	Coexistence request	other BS	BS	SCW
14	CO-RSP	Coexistence	BS	other BS	SCH

Table 2.5: List of management messages

2.6.1 Downstream map

The DS map is a management message sent by BS during FMW as a broadcast to all CPEs. It contains association of data bursts (predefined by latest DCD) to individual CPEs and information about the chosen PHY mode, see table 2.6.

At least a part of the first data burst is always occupied by FCH and FMW. So, if some space remains unoccupied, the burst is assigned to CPE and the user data fill the unoccupied part of the burst modulated and coded as described in the DS map. If no space is left, the burst is not assigned to any CPE.

Data	size
type = 1	5 bit
NDB repetitions of assignment:	
CPE ID	6 bit
PHY mode	4 bit

Table 2.6: DS-MAP form

2.6.2 Upstream map

The US map is a management message sent by BS during FMW as a broadcast to all CPEs. In the case of a frame without US subframe, the US map is not transmitted.

The US map contains information if the OW is on schedule during the frame. It further associates data bursts (predefined by latest UCD) to individual CPEs and informs which PHY mode and transmit power per subcarrier should be used by each CPE, see table 2.7. When OW or SCW occupies part of the data burst, only the unoccupied part of the burst is used for user data. When any data burst is covered as a whole by such windows, the burst is not assigned to any CPE.

2.6.3 Channel descriptors

DS and US channel descriptors are a management messages broadcast by BS to all CPEs. These messages are sent during SCH or FMW.

Data	size
type = 3	5 bit
OW bit	1 bit
NUB repetitions of assignment:	
CPE ID	6 bits
PHY mode	4 bits
power per subcarrier	8 bits

Table 2.7: US-MAP form

These messages defines sizes of data bursts (see figure 2.1). These bursts are associated in US and DS maps to individual CPEs for each frame individually.

Channel descriptors are always a part of SCH. They must be stored in each CPE and used for all following frames of the SF. If new channel descriptors are transmitted as part of any frame, the stored descriptors shall be rewritten and applied immediately to the frame they were part of.

The DS bursts are allocated vertically such, that each OFDM symbol is filled before the next symbol starts to be filled. First DS burst is counted from the beginning of DS subframe, thus at least part of it is always occupied by FCH and FMW. (In the case of the first frame of SF when two preambles are sent, the first burst is one symbol shorter.) The size is expressed in subchannels × OFDM symbols (e.g. size of 100 mean 5 OFDM symbols in 2 MHz system, 2.5 symbols in 4 MHz system and 1 symbol in 10 MHz system). The Sum of sizes of all data bursts divided by the number of subchannels is the length of DS subframe. The form of DCD is in table 2.8.

Data	size	notes
type = 0	5 bit	
number of DS bursts (NDB)	8 bit	
length of each burst	NDB × 8 bit	symbol × subchannel

Table 2.8: DCD form

The US bursts are allocated horizontally analogically to DS burst allocated vertically. So, each subchannel is filled across all OFDM symbols of US subframe before another subchannel is being filled. Sum of all burst lengths divided by number of OFDM symbols in US subframe must be equal to number of subchannels. The form of DCD is in table 2.9.

Data	size	notes
type = 2	5 bit	
number of US bursts (NUB)	8 bit	
length of each burst	NDB × 8 bit	symbol × subchannel

Table 2.9: UCD form

2.6.4 Ranging messages

There are two types of ranging messages. Ranging request is sent by CPE to BS when the received signal has poor or unnecessarily high quality. The BS alters transmit power, modulation depth or code rate of the transmitted signal in iterative manner. The form of ranging request is in table 2.10. The BS tells desired PHY mode and transmit power in each US map, so it does not need to sent such requests.

Data	size	notes
type = 4	5 bit	
action	1 bit	1 - increase robustness;
		0 - increase throughput

Table 2.10: RNG-REQ form

The ranging command is sent by BS to CPE. The aim of this command is to synchronize the arrival time of the signal from all CPEs. The CPE must start the next transmission sooner or later according to the correction time in the command.

Data	size	notes
type = 5	5 bits	
CPE ID	6 bits	
time shift	16 bits	tens of ns; signed int

Table 2.11: RNG-CMD form

2.6.5 Registration messages

Registration messages are used for registration of CPEs to the AWRAN. The CPE observes ongoing communication and reads US maps to see if the OW is on schedule.

The, so far non registered, CPE does not have an ID nor propagation delay estimation. So it waits for SCW and sends MC-CDMA registration request (see table 2.12) coded according to first row of 64×64 Hadamard matrix constructed by Sylvesters algorithm - thus 64 ones. The transmission is repeated during each OW with a $50\mu s$ time shift trying to guess the propagation delay. When the BS successfully reads the message, answers with a registration response.

Data	size
type = 6	5 bits
CPE MAC	48 bits

Table 2.12: REG-REQ form

The registration response contains CPE MAC address and ID assigned to the CPE. When the registration is declined, the ID field contains zero. The form of registration response is in table 2.13.

Data	size	notes
type = 7	5 bits	
CPE MAC	48 bits	
CPE ID	6 bits	0 - registration declined

Table 2.13: REG-RSP form

2.6.6 Bandwidth Request

The BW request is sent by CPE in OW. The first bit means a type of the BW request (incremental or aggregate). The rest of the message is a number of US bytes the CPE wants to transmit, see table 2.14.

Data	size	notes
type = 8	5 bits	
BW request type	1 bit	1 - incremental; 0 - aggregate
number of bytes	10 bits	

Table 2.14: BW-REQ form

2.6.7 Retransmission request

Either BS or CPE can ask for retransmission of user data that were not properly received. The retransmission request contains SF and frame number of the data to be retransmitted. The request transmitted by BS during FMW contains ID of the CPE it belongs to. Request sent by CPE during OW does not contain its ID, because the BS knows origin of each message from CDMA code. The form of the retransmission request is in table 2.15.

Data	size	notes
type = 9	5 bits	
CPE ID	6 bits	only if transmitted by BS to CPE
SF number	8 bits	
frame number	4 bits	

Table 2.15: RET-REQ form

2.6.8 SCW frame bitmap

The SCW frame bitmap is a message located in SCH which defines frames of SF that end with a SCW. The bitmap consists of sixteen bits, each bit corresponds to one frame of the SF. The form of SCW frame bitmap message is in table 2.16.

Data	size
type = 10	5 bits
bitmap	16 bits

Table 2.16: SCW-FB form

2.6.9 CP length

CP length message is transmitted during SCH and defines CP for the whole SF (except preambles). The form of this message is in table 2.17.

Data	size	notes
type = 11	5 bits	
time shift	2 bits	00 - $\frac{1}{4}$; 01 - $\frac{1}{8}$; 10 - $\frac{1}{16}$; 11 - $\frac{1}{32}$

Table 2.17: CPL form

2.6.10 Frame allocation (FA) map

This message is sent in SCH only when the AWRAN works in a coexistence mode. The bitmap features 32 bits - hence two bits belong to one frame. These two bits carry BS ID and tell which AWRAN cell will transmit during the frame. The first two bits are always 00 - meaning the first frame belongs to AWRAN cell of the transmitting BS. The form of the FA map message is in table 2.18.

Data	size
type = 12	5 bits
bitmap	32 bits

Table 2.18: FA-MAP form

2.6.11 Coexistence messages

Request for coexistence and coexistence response are two messages transmitted between BS negotiating on self-coexistence of its transmission.

When any BS spots another AWRAN transmission, it shall wait for SCW and transmit coexistence request during the window according to table 2.19.

Data	size
type = 13	5 bits
asking BS MAC	48 bits

Table 2.19: CO-REQ form

The BS which receives request for coexistence transmits coexistence response during first SCH assigning BS ID to the requesting BS. It must include also FA map message to the SCH and assign a proportional part of its resources to the other AWRAN.

There can be up to four coexisting AWRAN cells. If a fifth one asks for coexistence, the master BS will respond with ID 00 which means declination of the request. The form of coexistence response is in table 2.20.

Data	size	notes
type = 14	5 bits	
other BS MAC	48 bits	
other BS ID	2 bits	00 - declined

Table 2.20: CO-RSP form

CHAPTER 3

Prototype

The Vienna University of Technology has permission for test operation at frequencies 52 to 54 MHz. The frame structure with OFDM properties of the 2 MHz variant of the proposed communication scheme was simplified and implemented as a C++ program. It is running on prepared transceiver boxes RPX-100 [11]. These boxes contain Raspberry Pi (RPi) compute module 4 and the LimeSDR followed by a custom RF front-end. The output of the RPX-100 is connected to the HB9CV antenna. See the diagram in fig. 3.1.

The C++ program runs on the RPi. Two libraries are essential for this program. The first one is LimeSuite ensuring communication between the RPi and the LimeSDR. The second one is liquid-dsp which handles the OFDM frame generation and synchronization.

3.1 Liquid DSP library

Time samples of the implemented OFDM frame (described in chapter 3.2 are created using liquid digital signal processing library. It is written in C as an open

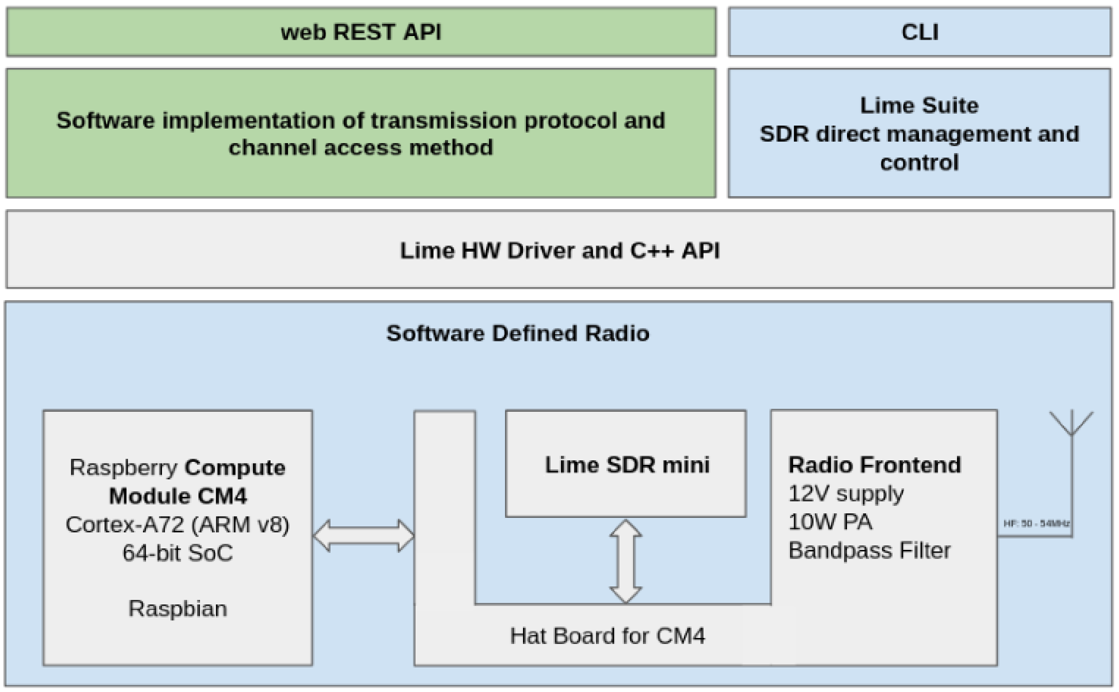

web REST API
CLI
Software implementation of transmission protocol and channel access method
Lime Suite SDR direct management and control
Lime HW Driver and C++ API
Software Defined Radio
Raspberry Compute Module CM4 Cortex-A72 (ARM v8) 64-bit SoC
Raspbian
Lime SDR mini
Radio Frontend 12V supply 10W PA Bandpass Filter
HF: 50 - 54MHz
Hat Board for CM4

Figure 3.1: RPX-100 block diagram [11]

source library for software-defined radios. [6]

This library has a number of modules that focuses on various signal processing tasks and other wireless communication-related problems. For example:

- FIR and IIR filters
- FFT
- auto-correlation
- automatic gain control
- channel emulation
- modulators
  - AM

- CPFSK
- FM
- GMSK
- OFDM

3.1.1 Flexible framing structure for OFDM

The essential module of liquid-dsp library for this thesis is called *ofdmflexframe*. It brings objects OFDM frame generator, which generates time samples for SDR to transmit, and OFDM frame synchronizer, which finds the OFDM frame in samples sampled by the SDR and passes received data. The module allows to set the number of subcarriers and make a custom map of null/data/pilot carriers, set length of cyclic prefix, set modulation, and FEC. The length of the payload is defined in bytes.

Although this module is called flexible, it has some predetermined structure. The first few symbols of each frame serve ad frame preamble and are dedicated to synchronization.

The preamble itself consists of two parts. The first part has a length of two or more OFDM symbols, depending on the number of subcarriers and their allocation. The symbols are transmitted always without CP and are used for coarse carrier frequency and timing offsets. The second part is always one OFDM symbol with CP matching the rest of the frame. It is used for fine timing and equalizer gain estimation.

The preamble is followed by a header which consists of 14 bytes where 6 of which are used internally by the library and 8 of them are user-defined. The 6 internally used bytes contain framing information such as modulation, FEC and payload length. The header itself is protected by FEC and its length in symbols depends on the number of data subcarriers. The header is followed by the payload.

For detailed information see chapter 16.7 of [5]

Frame generator

Frame generators are a group of objects in the liquid-dsp library that accepts raw data and (according to chosen modulation and its properties) produces time samples for SDR to transmit. The used frame generator is called *ofdmflexframegen* thus produces OFDM frames. Its properties allow to define:

- number and allocation of subcarriers
- cyclic prefix and taper length
- modulation
- inner and outer FEC scheme
- data validity check
- payload length

Frame synchronizer

Frame generators, similarly to frame generators, are another group of objects in the liquid-dsp library. They accept the time series of samples sampled by SDR and try to find the corresponding frame. The used synchronizer is called *ofdmflexframesync*.

The synchronizer needs to know the number and allocation of subcarriers as well as cyclic prefix length. All other parameters required for successful demodulation read out of the first six reserved bytes of each frame header.

The synchronizer thanks to the preamble and pilot subcarriers located in each transmitted symbol can compensate for carrier frequency and phase offset as well as multi-path fading. If FEC is used, the synchronizer automatically corrects errors. The received data are returned via a callback function.

3.1.2 Linear Digital Modulator/Demodulator

So-called *modem* is another module of liquid-dsp library which is used. The modulator works as part of the OFDM frame generator and the demodulator as part of the synchronizer. The supported modulations are:

- phase-shift keyinq (PSK) 2 to 256
- differential phase-shift keying (DPSK) 2 to 256
- amplitude-shift keying (ASK) 2 to 256
- quadrature amplitude-shift keying (QAM) 2 to 256
- amplitude and phase-shift keying (APSK) 2 to 256
- on-off keying (OOK)

3.1.3 Forward Error-Correction

The FEC module is also incorporated into the frame generator and synchronizer. Any supported error correction scheme can be used as a generator and synchronizer property:

- Hamming codes
- Repetition codes

- Convolutional codes
- Reed-Solomon codes
- Golay(24,12) block code
- single error correction - double error detection (SEC-DED) block codes

3.2 Implemented frame

As mentioned before, the OFDM generation is served by the liquid-dsp library. The library and its capability are described in section 3.1. It works only with singleplex frames, so the generated frame lacks the US subframe.

The implemented frame is always 10 ms long with number od OFDM symbols corresponding to the four allowed CP lengths - 26 symbols for CP $\frac{1}{4}$, 28 for $\frac{1}{8}$, 30 for $\frac{1}{16}$ and 31 for $\frac{1}{32}$.

All of the 14 PHY modes were implemented and can be arbitrarily chosen. The liquid library does not allow the selection of multiple different modulations or error codes within a single frame. The information about chosen PHY mode is carried in the frame header.

When a library libcorrec is installed before the liquid library installation, the FEC codes could be chosen as a property of the liquid frame generator object. Thus the FEC coding is done automatically.

The pilots in AWRAN change their position on the symbol to symbol basis. That is not possible using the liquid library. So the implemented version has pilots on fixed locations 7 subcarriers apart.

The preamble which liquid library generates is, as well as AWRAN uses, BPSK modulated binary sequence with the aim to low PAPR. On the other hand, AWRAN uses one OFDM symbol for SF preamble and one symbol for the frame preamble. Both with CP length of $\frac{1}{4}$. The liquid library generates preamble three OFDM symbols long. The first two OFDM symbols are two repetitions of S_0 symbol without CP. The third OFDM symbol of the preamble is S_1 symbol with CP matching the rest of the frame.

One OFDM symbol, sent after the preamble, is a header. The liquid library automatically generates 14 bytes long headers out of which 6 bytes are used by the library and 8 bytes are accessible for users. There is stored information about the used modulation and coding of the payload as well as payload length in the first 6 bytes. The 8 user-defined bytes are all set to zero in this stage. Later on, could be used for BS ID and FMW length if the usage of the liquid-dsp library will persist to further versions.

3.3 LimeSDR

Software-defined radio (SDR) is modern radio technology using direct digital processing of radio signal. This becomes possible with the increasing computational power of modern chips. Unlike traditional - hardware-defined - technologies, SDRs does its job according to software that could be on one hand overwritten on the fly, on the other hand, programmed to fit very specific requirements. SDRs are extremely flexible and allows to be programmed to transmit and receive arbitrary RF signals within its operating BW.

The LimeSDR developed by Lime microsystems contains three major integrated circuits (ICs): USB microcontroller, field programmable gate array (FPGA) and RF transceiver. See the block diagram on figure 3.2. In basic operation, the LimeSDR accepts in-phase and quadrature (IQ) samples via universal serial bus (USB) and sends corresponding RF signal via U.FL connector to the antenna, or receives RF

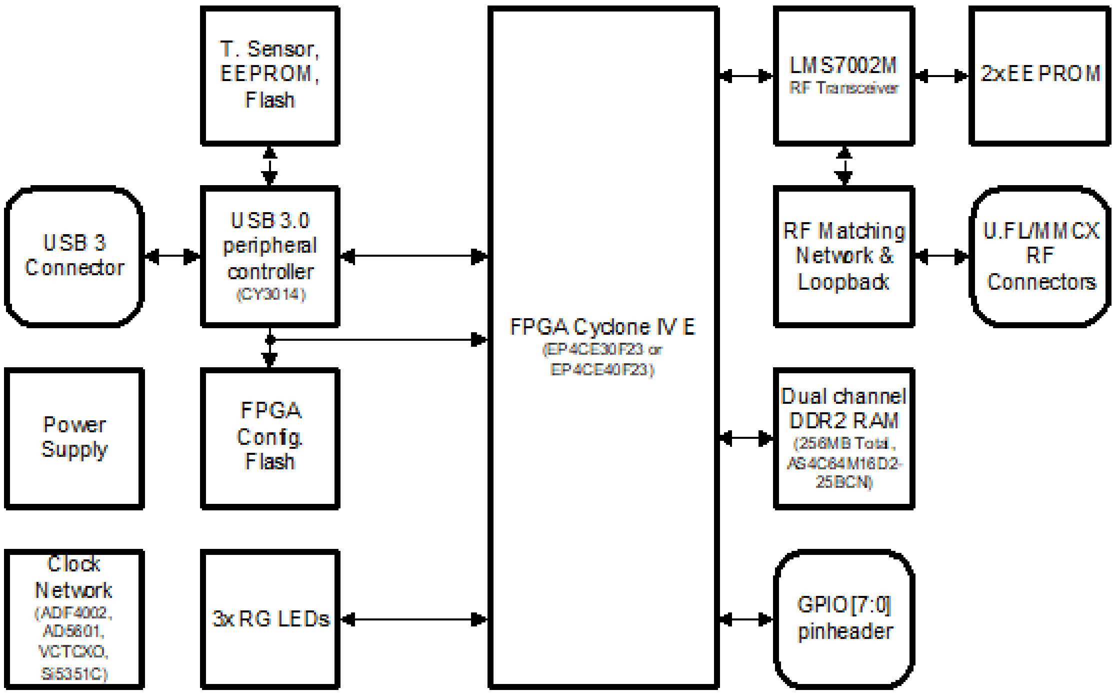

T. Sensor, EEPROM, Flash
USB 3 Connector
USB 3.0 peripheral controller (CY3014)
Power Supply
FPGA Config. Flash
Clock Network (ADF4002, AD5601, VCTCXO, Si5351C)
3x RG LEDs
FPGA Cyclone IV E (EP4CE30F23 or EP4CE40F23)
LMS7002M RF Transceiver
2xEEPROM
RF Matching Network & Loopback
U.FL/MMCX RF Connectors
Dual channel DDR2 RAM (256MB Total, AS4C64M16D2-25BCN)
GPIO[7:0] pinheader

Figure 3.2: LimeSDR block diagram [15]

signal and sends I and Q samples to USB respectively.

LimeSDR features FPGA Altera Cyclone IV EP4CE40F23 with factory pre-programmed gateware. The gateware handles the following features:

- Interface to LMS7002 LimeLightTM digital IQ interface in TRXIQ double data rate mode;
- Real-time data transfer between PC and LMS7002 chip.
- Connection to FX3 Slave FIFO interface for transferring data through USB3.0.
- TX samples synchronization with RX samples time stamp;
- SPI connection between LMS7002 chip and other on-board devices;
- WFM player which enables to load waveform to external DDR2 memory from USB3.0 host and translate to LMS7002 RXIQ interface.

- Reconfigurable PLL blocks for LMS7002 clocking.
- Internal SPI registers for FPGA control.

The RF signal itself is produced by field porgrammable radio frequency (FPRF) chip LMS7002M. Its dual-transmit and dual-receive channels enable full-duplex multiple-input multiple-output (MIMO) communications, although this application is way more simple. Each channel consists of in-phase and quadrature signal paths with phase locked loop (PLL), mixers, filters and ADCs or DACs respectively, see figure 3.3. It accepts IQ samples via LimeLight Digital IQ Interface which are converted to the analog signal, mixed with LO frequency, and amplified.

3.4 Raspberry Pi

RPi is a group of single board computers that are developed by Raspberry Pi Foundation from Cambridge, United Kingdom.[19] The first generation came out in February 2012 with a single core 32-bit ARM CPU running on 700 MHz and 256 MB of RAM. Nowadays, its successor RPi 4 model B has four core 64-bit ARM CPU clocked at 1.5 GHz. [46]

The model used in the project is called RPi Compute Module 4, it has the same processing power as RPi 4 model B but reduced input-output (IO) possibilities and differently shaped printed circuit board (PCB).

On the RPi runs a Linux-based operating system specifically developed for RPi computers called Raspberry Pi OS (its predecessor was called Raspbian). The operating system enables SSH connection and thus remote access to the boxes via the internet.

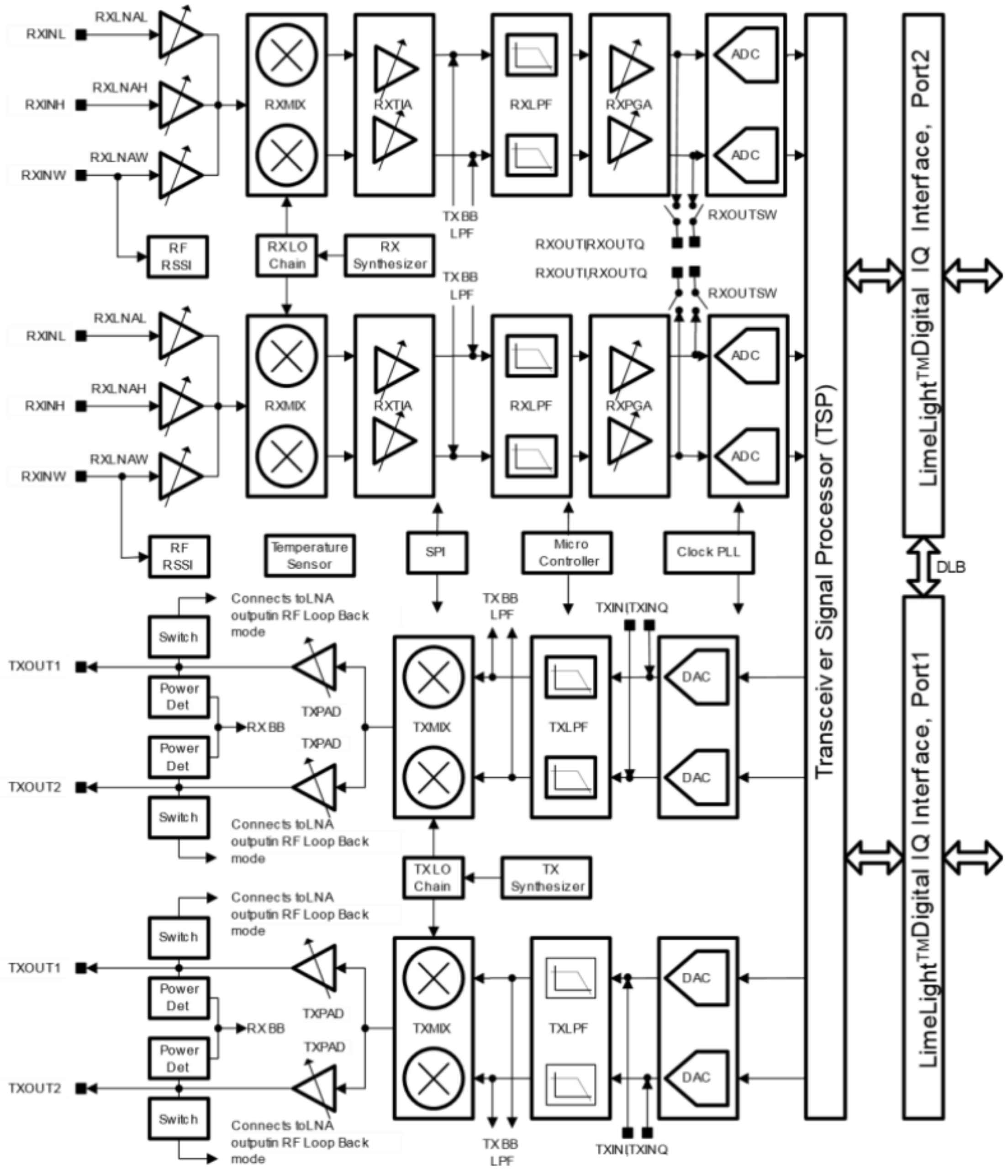


Figure 3.3: LMS7002M block diagram [16]

3.5 Radio frequency (RF) front-end

The custom RF front-end is a PCB prepared to amplify the output power of the LimeSDR while transmitting and pre-amplify the received signal before entering the SDR. [11]

It consists of power amplifiers, filters and RF switches creating eight possible RF paths: Three different frequency bands with **BP!** (**BP!**) filters, one direct path, and all this with or without power amplifier (PA). Mode with "PTT" in name use the PAs. Which RF path, or mode respectively, is active is selected by the C++ program via general purpose input-output (GPIO) pins of the LimeSDR.

The important RF ICs:

- monolithic amplifier (0-10 GHz) - PSA-14+
- switches (0 - 3.5 GHz) - HMC241AQS16E
- RF MOSFET power amplifier module (66 - 88 MHz) - RA30H0608M
- RF MOSFET amplifier module (135 - 175 MHz) - RA08H1317M
- RF MOSFET Amplifier Module (400 - 470 MHz) - RA07H4047M
- SPDT RF Switch (50 - 3000 MHz) - VSW2-33-10W

Supported modes:

- RX
- TX direct
- TX direct PTT
- TX 6 m

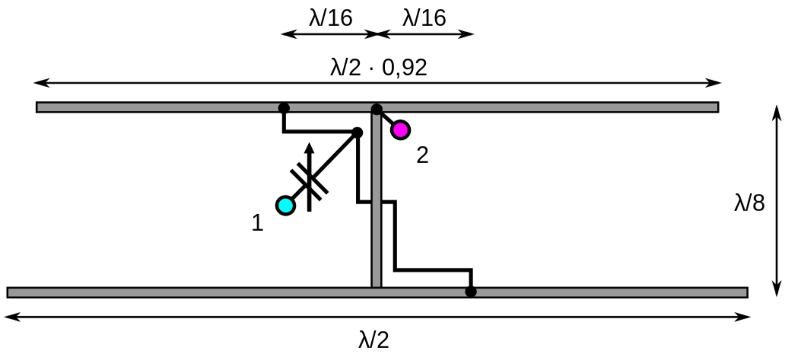


Figure 3.4: HB9CV antenna diagram [29]

- TX 6 m PTT
- TX 2 m
- TX 2 m PTT
- TX 70 cm
- TX 70 cm PTT

For more information about the RPX-100 boxes and its RF frontend including schematic and layout visit [11].

3.6 Antenna

On both stations are used directional antennas HB9CV, see fig. 3.4. It is a two-element phased array of active dipoles designed in 1960s by Rudolf Baumgartner. Its design is inspired by the ZL-Special antenna which uses two folded dipoles that were replaced by standard dipoles. Both these antennas improve the performance of the well-known single active element Yagi-Uda.

The Yagi-Uda consist of an active element surrounded by a number of passive elements which are parasitically fed. The rear element (single or more on top of each other) of Yagi-Uda, called the reflector, is longer than the resonant length which causes inductive load and thus phase shift. The front elements (could be single, but typically multiple in a row) called directors are on the other hand shorter than the resonant length causing capacitive load and opposing phase shift. All this together causes constructive interference in the forward direction and destructive interference in the backward direction creating directional radiation.

The principle of the HB9CV antenna is similar to the Yagi-Uda. There are only two elements that are actively fed which increases their efficiency. Thus the two-element HB9CV performs similarly to three or four-element Yagi-Uda. The two elements are spaced $\frac{\lambda}{8}$ (or 45 °) apart. The front element is $0.96 \times \frac{\lambda}{2}$ long which causes 45 ° shift of phase. The rear element is $1.04 \times \frac{\lambda}{2}$ long which causes -45 ° phase shift. Another -45 ° phase shift on the rear element is introduced by $\frac{\lambda}{8}$ long feeding line difference between the two elements. The remaining 180 ° phase shift is achieved by flipped feeding orientation. [2]

Reducing the number of elements (hence the size of the antenna) is convenient, especially for large wavelengths. The HB9CV was originally designed for 10, 15, and 20 m bands. But the antennas for the 6 m band are still quite large, so the reduced size is welcomed.

3.7 Experimental radio link

There is an established experimental radio link between two RPX-100 boxes. The endpoint locations:

1st: Amateur Radio Station OE3BIA, at Maidenhead Locator JN88af, at 208 m above sea level with antenna 20 m above ground, 3443 Sieghartskirchen, Austria.

2nd: Amateur Radio Station OE1XDU at Maidenhead Locator JN88ee, at 176 m above sea level with antenna 46 m above ground, 1040 Vienna, Austria.

The link is 27.3 km long with hills that obstruct the LOS. See figure 3.6.

An estimation of attenuation caused by the RF signal propagation is described on the lines below.

3.7.1 Free space propagation attenuation

The major contributor to the link attenuation is the propagation distance of 27.3 km. An attenuation of free space propagation is calculated by the equation 3.1.

$$A_0 = 20log\left(\frac{4\pi d}{\lambda}\right) = 20log\left(\frac{4\pi 27300}{5.7}\right) \approx 96 \text{ dB} \tag{3.1}$$

where:

A_0 stands for free space attenuation (dB)

d stands for link length (m)

λ stands for wavelength (m)

3.7.2 NLOS propagation attenuation

Estimation of the NLOS propagation attenuation is very rough. On the terrain profile in fig. 3.6 [25] are visible four major hills marked as A, B, C, and D. For the calculation, all of the hills were approximated with the obstacle by sphere. [13]

Hill B was selected as a major obstacle. Its contribution to overall attenuation was calculated first using equations 3.2, 3.3, 3.4 and 3.5.

Two imaginary links from top of the hill B (one to each box) were considered for further calculation of the attenuation contribution of hills A (see eq. 3.6, 3.7, 3.8 and 3.9), C (see eq. 3.10, 3.11, 3.12 and 3.13), and D (see eq. 3.14, 3.15, 3.16 and 3.17).

The eq. 3.4, 3.8, 3.12, and 3.16 were done according to plot on fig. 3.5.

Influence of hill B

$$H_0 = \sqrt{\frac{1}{3}}\sqrt{\lambda\frac{r_1(r-r_1)}{r}} = \sqrt{\frac{1}{3}}\sqrt{6\frac{7500(27300-7500)}{27300}} = 104.3m \tag{3.2}$$

where H_0 is a specific clearance, λ is a wavelength, r_1 is the distance from the first station to hill B, and r is the total distance of the link.

$$\begin{aligned} v &= 2.02\sqrt[3]{\frac{\Delta y}{\Delta x^2}\frac{r_1^2}{H_0}\left(1-\frac{r_1}{r_1+r_2}\right)^2}\sqrt[4]{1+\frac{H}{4r_1r_2}\frac{\Delta x^2}{\Delta y}} = \\ &= 2.02\sqrt[3]{\frac{200}{\Delta 5000^2}\frac{7500^2}{104.3}\left(1-\frac{7500}{7500+19800}\right)^2}\sqrt[4]{1+\frac{-260}{4\times 7500\times 19800}\frac{5000^2}{200}} = \\ &= 2.7 \end{aligned} \tag{3.3}$$

where v is an obstacle parameter, Δy and Δx are shape parameters of hill B, H is the height of the hill B with respect to the link, r_1 is the distance from the first station to hill B, and r_2 is the distance from the hill B to the second station.

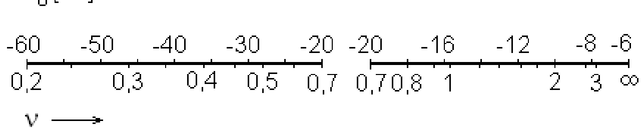


Figure 3.5: obstacle parameter to basic attenuation factor conversion

$$v = 2.7 => v_0 = -9dB \tag{3.4}$$

where v is a obstacle parameter and v_0 is a basic attenuation factor.

$$W = v_0 \left(1 - \frac{H}{H_0}\right) = -9 \left(1 - \frac{-260}{104.3}\right) = -31.4dB \tag{3.5}$$

where W is an attenuation factor, H is the height of the hill B with respect to the link, and H_0 is a specific clearance.

Influence of hill A

For estimation of contribution to attenuation of the link by the hill A is considered an imaginary link between the first station and top of the hill B.

$$H_0 = \sqrt{\frac{1}{3}} \sqrt{\lambda \frac{r_1(r - r_1)}{r}} = \sqrt{\frac{1}{3}} \sqrt{6 \frac{3000(7500 - 300)}{7500}} = 52.9m \tag{3.6}$$

where H_0 is a specific clearance, λ is a wavelength, r_1 is the distance from the first station to hill A, and r is the total distance of the imaginary link between the first station and hill B.

$$\begin{aligned} v &= 2.02 \sqrt[3]{\frac{\Delta y}{\Delta x^2} \frac{r_1^2}{H_0} \left(1 - \frac{r_1}{r_1 + r_2}\right)^2} \sqrt[4]{1 + \frac{H}{4 r_1 r_2} \frac{\Delta x^2}{\Delta y}} = \\ &= 2.02 \sqrt[3]{\frac{60}{\Delta 1000^2} \frac{3000^2}{52.9} \left(1 - \frac{3000}{3000 + 4500}\right)^2} \sqrt[4]{1 + \frac{-90}{4 \times 3000 \times 4500} \frac{1000^2}{60}} = \\ &= 3.1 \end{aligned} \tag{3.7}$$

where v is an obstacle parameter, Δy and Δx are shape parameters of the hill A, H is the height of the hill A with respect to the imaginary link, r_1 is the distance from the first station to hill A and r_2 is the distance from the hill A to the hill B.

$$v = 3.1 => v_0 = -7dB \tag{3.8}$$

where v is an obstacle parameter and v_0 is a basic attenuation factor.

$$W = v_0\left(1 - \frac{H}{H_0}\right) = -7\left(1 - \frac{-90}{52.9}\right) = -18.9dB \tag{3.9}$$

where W is an attenuation factor, H is the height of hill A with respect to the imaginary link, and H_0 is a specific clearance.

Influence of hill C

For estimation of contribution to attenuation of the link by the hill C, an imaginary link between the top of hill B and the second station is considered.

$$H_0 = \sqrt{\frac{1}{3}}\sqrt{\lambda\frac{r_1(r-r_1)}{r}} = \sqrt{\frac{1}{3}}\sqrt{6\frac{7500(19800-7500)}{19800}} = 96.5m \tag{3.10}$$

where H_0 is a specific clearance, λ is a wavelength, r_1 is the distance from the top of hill B to the hill C and r is the total distance of the imaginary link between hill B and the second station.

$$\begin{aligned} av &= 2.02\sqrt[3]{\frac{\Delta y}{\Delta x^2}\frac{r_1^2}{H_0}\left(1-\frac{r_1}{r_1+r_2}\right)^2}\sqrt[4]{1+\frac{H}{4r_1r_2}\frac{\Delta x^2}{\Delta y}} = \\ &= 2.02\sqrt[3]{\frac{70}{\Delta 2000^2}\frac{7500^2}{104.3}\left(1-\frac{7500}{7500+12300}\right)^2}\sqrt[4]{1+\frac{-50}{4\times 7500\times 12300}\frac{2000^2}{70}} = \\ &= 3.6 \end{aligned} \tag{3.11}$$

where v is an obstacle parameter, Δy and Δx are shape parameters of the hill C, H is the height of the hill C with respect to the imaginary link, r_1 is the distance from the hill B to the hill C and r_2 is the distance from the hill C to the second station.

$$v = 3.6 => v_0 = -6dB \tag{3.12}$$

where v is a obstacle parameter and v_0 is a basic attenuation factor.

$$W = v_0 \left(1 - \frac{H}{H_0}\right) = -6 \left(1 - \frac{-50}{96.5}\right) = -9.1dB \tag{3.13}$$

where W is an attenuation factor, H is the height of the hill C with respect to the imaginary link, and H_0 is a specific clearance.

Influence of hill D

For estimation of contribution to attenuation of the link by the hill D, an imaginary link between the top of the hill C and the second station is considered.

$$H_0 = \sqrt{\frac{1}{3}}\sqrt{\lambda \frac{r_1(r - r_1)}{r}} = \sqrt{\frac{1}{3}}\sqrt{6\frac{6000(13300 - 6000)}{13300}} = 81.2m \tag{3.14}$$

where H_0 is a specific clearance, λ is a wavelength, r_1 is the distance from the hill C to the hill D and r is the total distance of the imaginary link between hill C and the second station.

$$\begin{aligned} v &= 2.02\sqrt[3]{\frac{\Delta y}{\Delta x^2}\frac{r_1^2}{H_0}\left(1 - \frac{r_1}{r_1 + r_2}\right)^2}\sqrt[4]{1 + \frac{H}{4r_1 r_2}\frac{\Delta x^2}{\Delta y}} = \\ &= 2.02\sqrt[3]{\frac{50}{\Delta 1500^2}\frac{6000^2}{81.2}\left(1 - \frac{6000}{6000 + 7300}\right)^2}\sqrt[4]{1 + \frac{30}{4 \times 6000 \times 7300}\frac{1500^2}{50}} = \\ &= 2.9 \end{aligned} \tag{3.15}$$

where v is an obstacle parameter, Δy and Δx are shape parameters of the hill D, H is the height of the hill D with respect to the imaginary link, r_1 is the distance from the hill C to hill D and r_2 is the distance from the hill D to the second station.

$$v = 2.9 => v_0 = -8.5dB \tag{3.16}$$

where v is a obstacle parameter and v_0 is a basic attenuation factor.

$$W = v_0\left(1 - \frac{H}{H_0}\right) = -8.5\left(1 - \frac{30}{81.2}\right) = -5.4dB \tag{3.17}$$

where W is an attenuation factor, H is the height of the hill D with respect to the imaginary link, and H_0 is a specific clearance.

$$W_{tot} = W_A + W_B + W_C + W_D = -18.9 - 31.4 - 9.1 - 5.4 = 64.8 \approx 65dB \tag{3.18}$$

The total attenuation caused by the obstacles is estimated in equation 3.18 to be 65 dB.

3.7.3 Total propagation attenuation

is estimated as a sum of the free space attenuation and contribution by obstacles. In our case the estimated attenuation is (96+65) dB = 161 dB

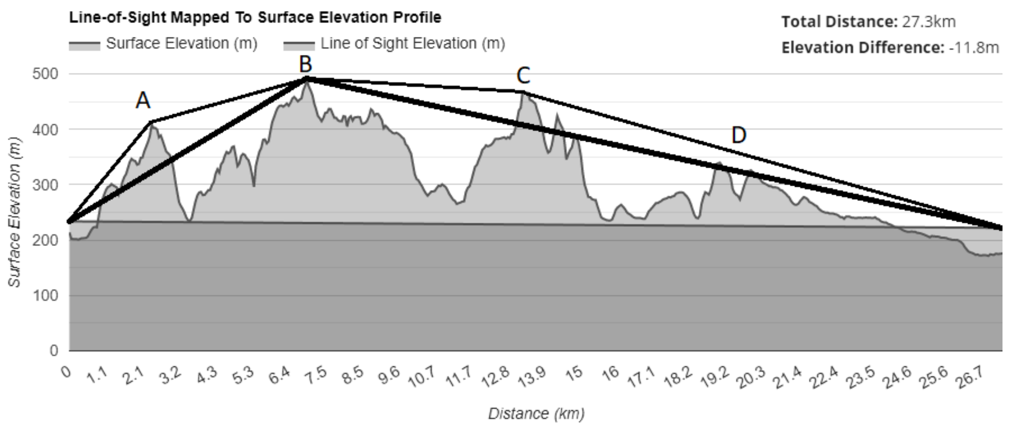


Figure 3.6: Terrain profile for the RF link between Station OE3BIA, Locator:JN88af, on the left and Station OE1XTU, Locator:JN88ee, on the right.

CHAPTER 4

Programs

In this chapter, four individual programs are described. The first two - "FM receiver" and "FSK prototype" - could be considered as development stages and the last two - "transceiver" and "BER simulator" - as working prototypes.

4.1 FM receiver

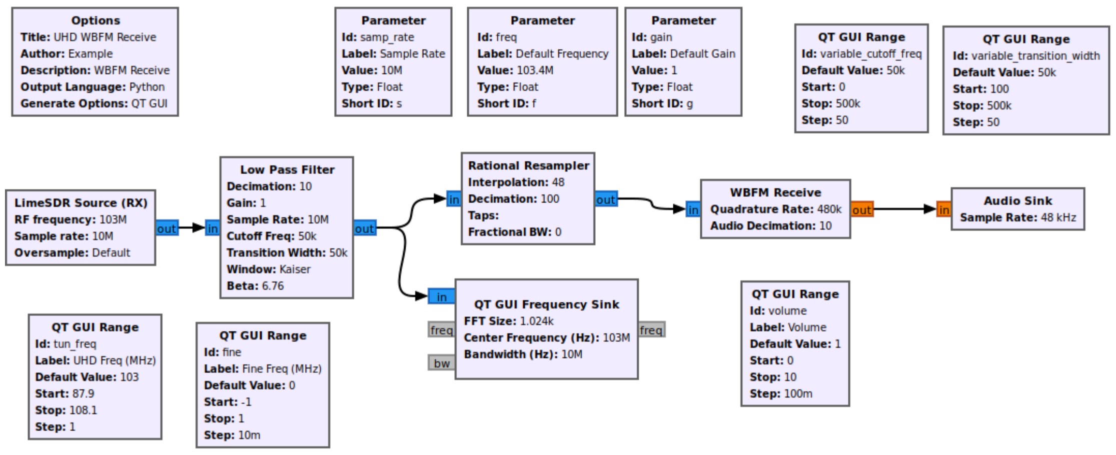
At the very beginning, there was a need to connect the LimeSDR to the computer and validate its functionality.

A software (SW) collection called Lime Suite [14] was installed. It contains drivers for the LMS7002M transceiver radio-frequency integrated circuit (RFIC), and other tools for developing with LMS7-based hardware such as Lime SDR.

The FM receiver program was created in a graphical tool GNU radio which provides various signal processing blocks such[23]:

- analog modulation
- audio interface

- channel model blocks
- digital modulation
- packet communication
- FEC
- FFT
- voice coders and decoders



Options
Title: UHD WBFM Receive
Author: Example
Description: WBFM Receive
Output Language: Python
Generate Options: QT GUI

Parameter
Id: samp_rate
Label: Sample Rate
Value: 10M
Type: Float
Short ID: s

Parameter
Id: freq
Label: Default Frequency
Value: 103.4M
Type: Float
Short ID: f

Parameter
Id: gain
Label: Default Gain
Value: 1
Type: Float
Short ID: g

QT GUI Range
Id: variable_cutoff_freq
Default Value: 50k
Start: 0
Stop: 500k
Step: 50

QT GUI Range
Id: variable_transition_width
Default Value: 50k
Start: 100
Stop: 500k
Step: 50

LimeSDR Source (RX)
RF frequency: 103M
Sample rate: 10M
Oversample: Default

Low Pass Filter
Decimation: 10
Gain: 1
Sample Rate: 10M
Cutoff Freq: 50k
Transition Width: 50k
Window: Kaiser
Beta: 6.76

Rational Resampler
Interpolation: 48
Decimation: 100
Taps:
Fractional BW: 0

WBFM Receive
Quadrature Rate: 480k
Audio Decimation: 10

Audio Sink
Sample Rate: 48 kHz

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 103M
Bandwidth (Hz): 10M

QT GUI Range
Id: volume
Label: Volume
Default Value: 1
Start: 0
Stop: 10
Step: 100m

QT GUI Range
Id: tun_freq
Label: UHD Freq (MHz)
Default Value: 103
Start: 87.9
Stop: 108.1
Step: 1

QT GUI Range
Id: fine
Label: Fine Freq (MHz)
Default Value: 0
Start: -1
Stop: 1
Step: 10m

Figure 4.1: FM receiver in GNU radio

The Lime SDR was connected with a computer via USB. To the LimeSDR was connected a whip antenna. The block structure on figure 4.1 was done according to an example available on [15]. The LimeSDR Source (RX) block sets parameters of the SDR. When a proper frequency was set, the tuned radio station was audible via computer speakers.

4.2 FSK prototype

In order to learn how the liquid-dsp library and Lime Suite work, and how to interface between them, the first C++ program prototype was written with one of the simplest digital modulations - the frequency shift keying (FSK). The starting point was a code skeleton serving RF frontend settings.

For time samples creation of the FSK modulated signal, the *fskmodem* module of liquid-dsp library was used. The module brings objects `fskmod` (modulator) and `fskdem` (demodulator). These objects are returned by functions `fskmod_create` or `fskdem_create` respectively. Both these functions accept three parameters: number of bits per symbol, number of samples per symbol, and frequency spacing.

The objects of the modulator and demodulator were defined with basic settings. Single bit per symbol to have just two separate frequencies in the signal. Four samples per symbol and a default frequency spacing of 0.2.

Function `fskmod_modulate` has three parameters: the modulator object, the input symbol, and a pointer to a transmit buffer. This function fills the buffer with complex time samples of the desired symbol which needs to be passed to the LimeSDR. Function `fskdem_demodulate` works similarly with the demodulator object and receive (RX) buffer as parameters and received symbol as return value.

4.2.1 LimeSDR usage

Usage of LimeSDR is provided by LimuSuite library. The initialization procedure is following. An array of type `lms_info_str_t` needs to be created to hold information about all potential connected LimeSDRs. In this case, only one is used, but the array is prepared for up to eight devices. The following functions must be executed for initialization and setting up the LimeSDR:

The function LMS_GetDeviceList with the mentioned array as its argument searches for connected LimeSDRs. The array is filled with information about all discovered LimeSDRs and a number of discovered devices is returned.

A pointer to data type lms_device_t needs to be prepared for addressing the desired device. The pointer is used as an argument for all the following functions.

A function LMS_Open accepts the memory address of the device pointer and the information about the discovered LimeSDR. Execution of the function makes the device pointer point on the opened LimeSDR.

A LMS_Init function initializes the opened device with default settings. Function LMS_EnableChannel with either LMS_CH_TX or LMS_CH_RX macro as its argument enables transmit (TX) or RX channel. Function LMS_SetSample- -Rate sets sample rate of the SDR. the functions LMS_SetLOFrequency, and LMS_SetAntenna set properties for particular channel.

Both, TX and RX gain are adjustable using one of the functions LMS_SetNormali- -zedGain and LMS_SetGaindB depending on desired units. [18]

For the data passage to the SDR, an object of class lms_stream_t representing the stream of time samples has to be created. The object has the following attributes: channel, fifoSize, throughputVsLatency, isTx, and dataFmt (data format of the time samples). An object of class lms_stream_meta_t could be defined for the precision timing of the transmission.

Once the stream is set by function LMS_SetupStream and started by function LMS_StartStream, the transmit buffer containing I and Q samples of the desired signal could be transmitted using function LMS_Send- -Stream. In case of receiving a function LMS_RecvStream fills the RX buffer with received I and Q samples.

An important note is, that functions LMS_SendStream and LMS_RecvStream accepts as arguments buffers of interleaved I and Q samples and a number of samples, but the number of samples means number of I and Q pairs. Thus the number of samples shall be half the buffer length.

The stram is closed by function LMS_StopStream and the object is destroyed by function LMS_DestroyStream after the transmission. The SDR is closed by function LMS_Close before the program terminates.

4.2.2 liquid-dsp and LimeSuite data handover

An important part of the program is the handover of time samples between the two libraries. The liquid-dsp works with proprietary data type liquid_float_complex. The LimeSuite on the other hand uses standard float. The data stream consists of repeating I and Q samples where I sample matches the real part of the complex number and Q sample matches the imaginary part.

The buffer of complex numbers working with liquid-dsp functions is half the size of the real numbers buffer working with LimeSuite. The conversions are done in for loops, see the following code examples.

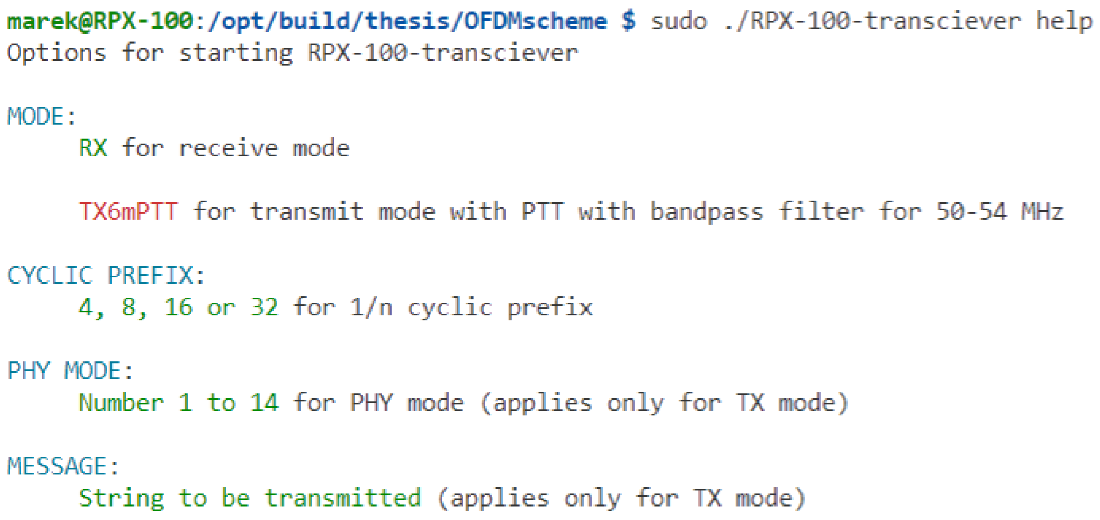
```
liquid_float_complex    complex_i (0, 1);
liquid_float_complex    c_buffer[c_buffer_len];
float                   r_buffer[2*c_buffer_len];

//complex to real buffer conversion
for (int i = 0; i < c_buffer_len; i++) {
    r_buffer[2*c_buffer_len+2*i] = c_buffer[i].real();
    r_buffer[2*c_buffer_len+2*i+1] = c_buffer[i].imag();}
```

```
//real to complex buffer conversion
for (int i = 0; i < c_buffer_len; i++) {
    c_buffer[i] = r_buffer[2*i] +
                  r_buffer[2*i+1] * complex_i.imag();}
```

4.3 Transceiver

The transceiver is the program, that transmits or receives the simplified AWRAN frame described in the chapter 3.2. It writes possible settings to the terminal (see fig. 4.2) when is called with argument help. Otherwise is started as a transmitter or receiver using arguments TX6mPTT or RX respectively. The default mode is RX.



```
marek@RPX-100:/opt/build/thesis/OFDMscheme $ sudo ./RPX-100-transciever help
Options for starting RPX-100-transciever

MODE:
     RX for receive mode

     TX6mPTT for transmit mode with PTT with bandpass filter for 50-54 MHz

CYCLIC PREFIX:
     4, 8, 16 or 32 for 1/n cyclic prefix

PHY MODE:
     Number 1 to 14 for PHY mode (applies only for TX mode)

MESSAGE:
     String to be transmitted (applies only for TX mode)
```

Figure 4.2: RPX-100-transciever with argument help

4.3.1 TX6mPTT mode

When is started in TX mode it accepts three additional arguments: CP length, PHY mode, and a message string.

The CP length argument could be 4, 8, 16, or 32. The number is a denominator of the desired CP length. This number defines the sample rate of the SDR and affects the number of transmitted OFDM symbols.

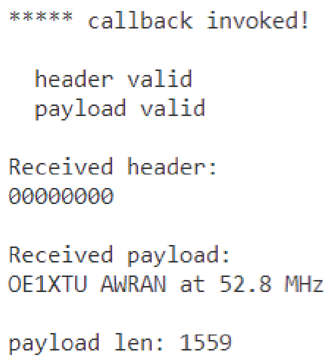
The payload length of the frame depends on the number of OFDM symbols as well as on the selected PHY mode which are specified in chapter 2.2. The PHY mode is a third argument of the program and could be set anywhere from one to fourteen. The default PHY mode 1 is used when the argument is not used.

The last accepted argument is a message of type string. For the transmission of multiple words, quotation marks must be used. When the message is not specified, the default message "OE1XTU AWRAN at 52.8 MHz" is transmitted.

If the message argument is longer than the payload length with current settings (CP and PHY mode), only the corresponding part is accepted and transmitted.

4.3.2 RX mode

When is started in RX mode it accepts only one additional argument - CP length (see 4.3.1). Then enters the infinite reception loop. Once a frame is recognized, its header and payload are printed into the terminal by a callback function, see fig. 4.3.



```
***** callback invoked!

  header valid
  payload valid

Received header:
00000000

Received payload:
OE1XTU AWRAN at 52.8 MHz

payload len: 1559
```

Figure 4.3: Received frame

4.3.3 Transmission over test link

Unfortunately, neither of the transmitted OFDM frames was actually received by the other station on the test link described in chapter 3.7. The reason is insufficiently strong PA in the RF frontend. The RPX-100 is under ongoing development and this part is not ready yet.

There is another software running on the RPX-100 which shows a waterfall graph of the transmission band. When a transceiver is started, on the graph is visible the calibration procedure of the SDR as a small peak close above the noise floor caused by powered on PA amplifying the local oscillator signal. This indicates that some of the transmitted energy is actually received. And it is proven by successful transmission and reception of a narrowband FSK signal transmitted by the FSK prototype described in chapter 4.2.

The OFDM frame has a very broadband signal, compared to the FSK. Thus the small power produced by the PA is spread across the band and hidden under the noise.

The transceiver was modified to transmit the frame one hundred times in a row, thus the transmission takes one full second and is observable on the waterfall graph. Look at figure 4.4, the bright line indicates the calibration procedure, then is followed by vanished broad line and after the one-second OFDM transmission ends, the bright line appears again.

4.3.4 Program structure

In the main function, program arguments are parsed and control variables are set accordingly. Then the LimeSDR initialization and all settings are done as described in chapter 4.2.1 in function `SDRinit`. The sample rate is set according to selected CP. The calculation for CP $\frac{1}{4}$ is done in equation 4.1 and all the possibilities

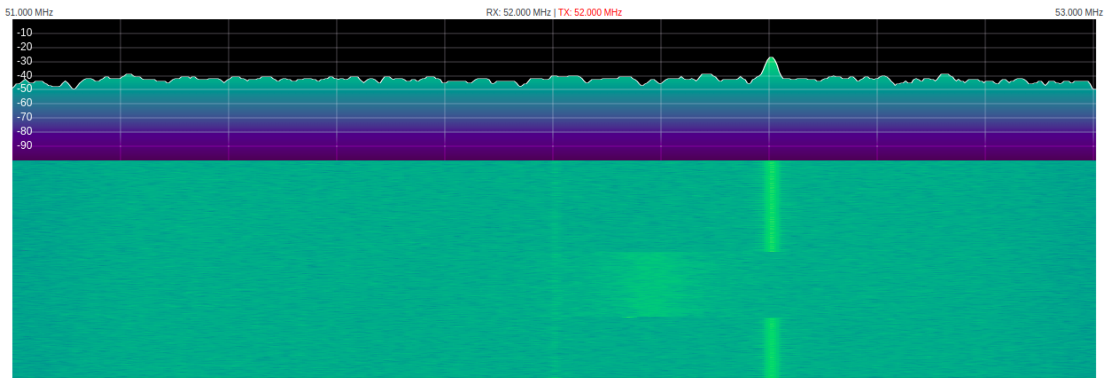


Figure 4.4: Waterfall spectrogram of received OFDM signal across the test link

CP	number symbols	sample rate
$\frac{1}{4}$	26	3328000
$\frac{1}{8}$	28	3225600
$\frac{1}{16}$	30	3264000
$\frac{1}{32}$	31	3273600

Table 4.1: Sample rate settings according to selected CP

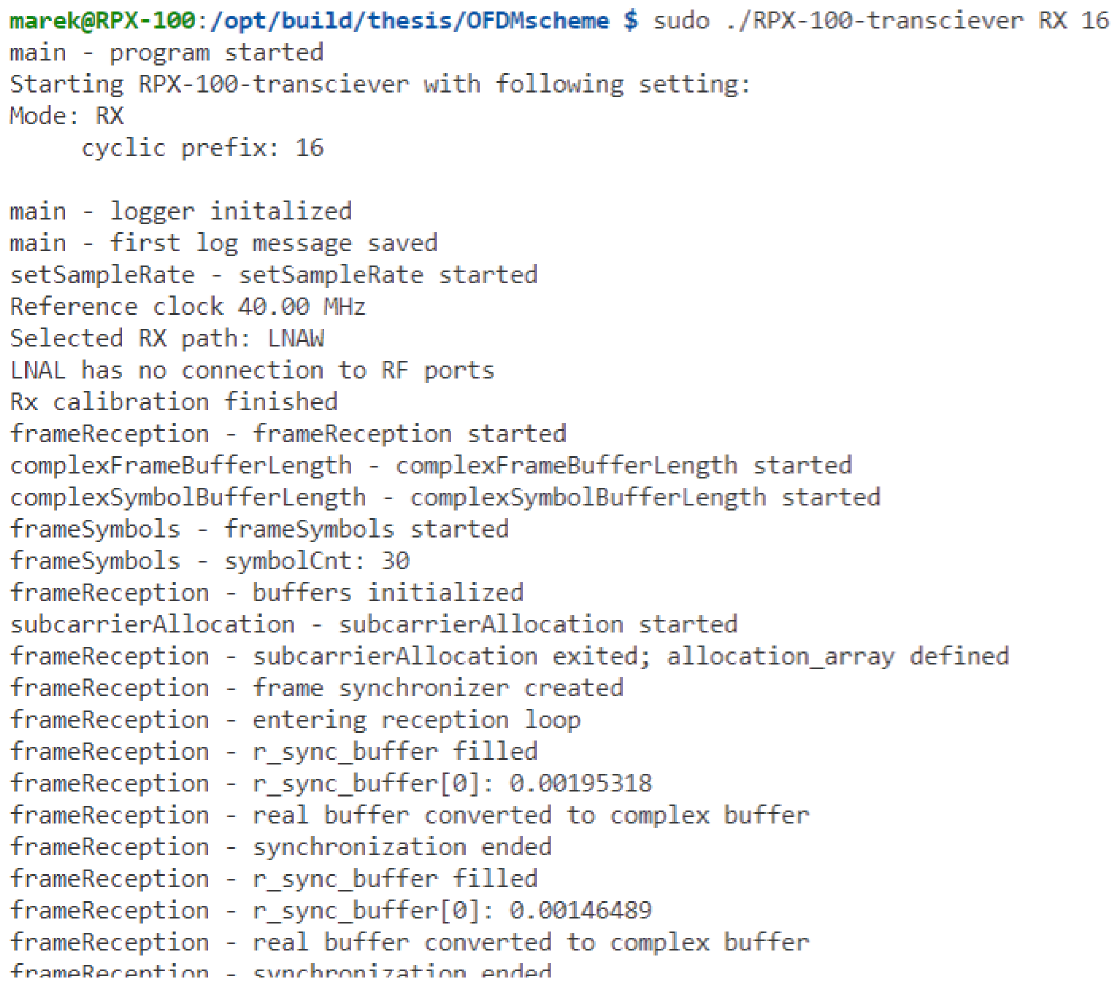
are listed in table 4.1. In case of successful initialization either `sendFrame` or `frameReception` function is called, depending on selected operation mode.

$$SR = \frac{n_{FFT} \times n_{symbols} \times (1+CP)}{t_{frame}} = \frac{1024 \times 26 \times (1+\frac{1}{4})}{10 \text{ ms}} = 3328000 \quad (4.1)$$

If RX mode is selected, the reception is done in an endless loop. If TX mode is selected, in the `sendFrame` function executes `frameAssemble` which fills transmit buffer with time samples using liquid-dsp and passes the buffer for the actual transmission to the function `startSDRTXstream`. After the transmission, the `SDRinit` function is called again with the RX settings which disables PA on RF front end. Before termination of the program, the LimeSDR is disconnected by calling function `LMS_Close`.

All these functions are described in chapter 5.

The terminal output with enabled debug messages (see 5.1) for RX mode is on fig. 4.5 and for TX mode is on fig. 4.6.



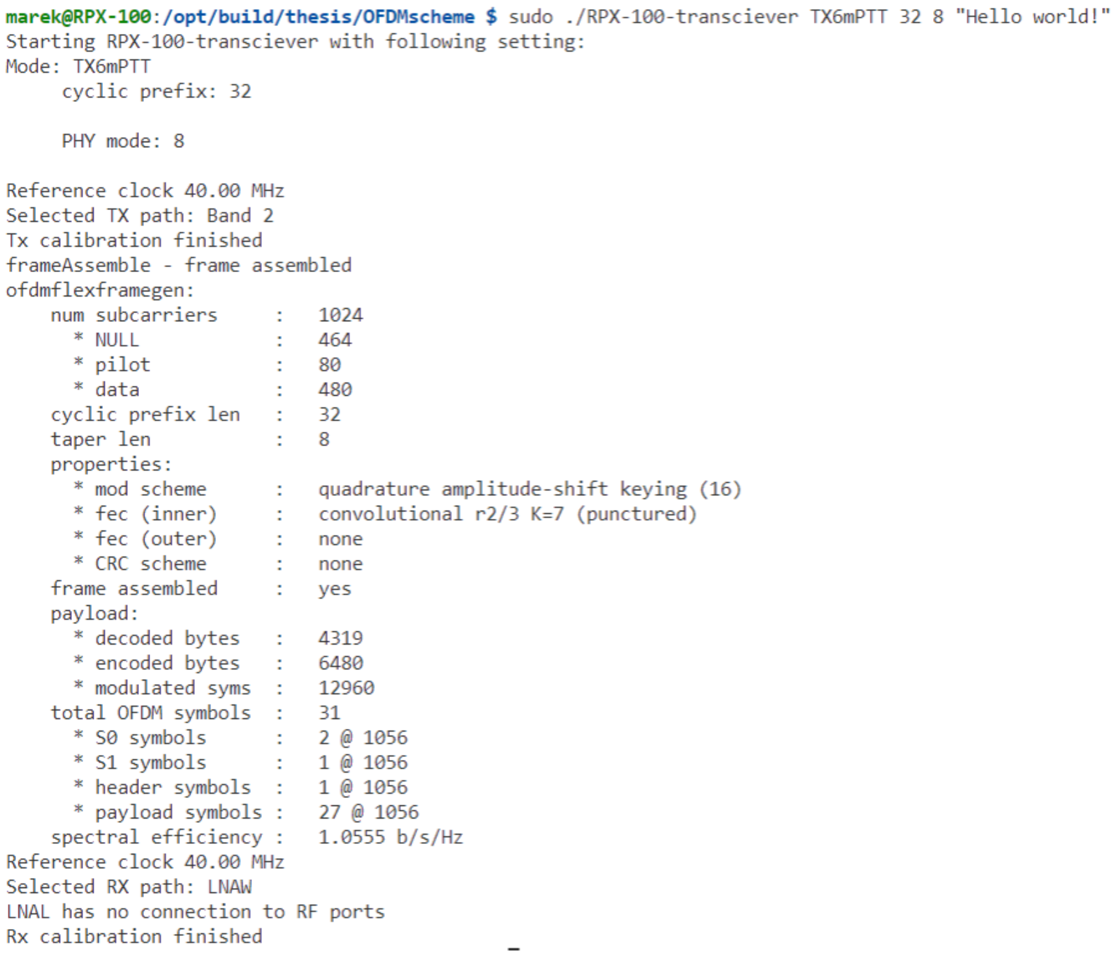
```
marek@RPX-100:/opt/build/thesis/OFDMscheme $ sudo ./RPX-100-transciever RX 16
main - program started
Starting RPX-100-transciever with following setting:
Mode: RX
     cyclic prefix: 16

main - logger initalized
main - first log message saved
setSampleRate - setSampleRate started
Reference clock 40.00 MHz
Selected RX path: LNAW
LNAL has no connection to RF ports
Rx calibration finished
frameReception - frameReception started
complexFrameBufferLength - complexFrameBufferLength started
complexSymbolBufferLength - complexSymbolBufferLength started
frameSymbols - frameSymbols started
frameSymbols - symbolCnt: 30
frameReception - buffers initialized
subcarrierAllocation - subcarrierAllocation started
frameReception - subcarrierAllocation exited; allocation_array defined
frameReception - frame synchronizer created
frameReception - entering reception loop
frameReception - r_sync_buffer filled
frameReception - r_sync_buffer[0]: 0.00195318
frameReception - real buffer converted to complex buffer
frameReception - synchronization ended
frameReception - r_sync_buffer filled
frameReception - r_sync_buffer[0]: 0.00146489
frameReception - real buffer converted to complex buffer
frameReception - synchronization ended
```

Figure 4.5: termial output of RPX-100-transceiver in RX mode

4.4 BER simulator

The BER simulator program is based on the transceiver. All SDR related chunks of code were removed and the frame assembling part with frame synchronizing part were interconnected by an artificial channel.



```
marek@RPX-100:/opt/build/thesis/OFDMscheme $ sudo ./RPX-100-transciever TX6mPTT 32 8 "Hello world!"
Starting RPX-100-transciever with following setting:
Mode: TX6mPTT
     cyclic prefix: 32

     PHY mode: 8

Reference clock 40.00 MHz
Selected TX path: Band 2
Tx calibration finished
frameAssemble - frame assembled
ofdmflexframegen:
    num subcarriers     :   1024
      * NULL            :   464
      * pilot           :   80
      * data            :   480
    cyclic prefix len   :   32
    taper len           :   8
    properties:
      * mod scheme      :   quadrature amplitude-shift keying (16)
      * fec (inner)     :   convolutional r2/3 K=7 (punctured)
      * fec (outer)     :   none
      * CRC scheme      :   none
    frame assembled     :   yes
    payload:
      * decoded bytes   :   4319
      * encoded bytes   :   6480
      * modulated syms  :   12960
    total OFDM symbols  :   31
      * S0 symbols      :   2 @ 1056
      * S1 symbols      :   1 @ 1056
      * header symbols  :   1 @ 1056
      * payload symbols :   27 @ 1056
    spectral efficiency :   1.0555 b/s/Hz
Reference clock 40.00 MHz
Selected RX path: LNAW
LNAL has no connection to RF ports
Rx calibration finished
```

Figure 4.6: terminal output of RPX-100-transceiver in TX6mPTT mode

In the first part of the main function are four nested `for` loops. In the inner one, artificial channel and frame reception is executed for all simulated SNRs. The second is responsible for going thru all the PHY modes such, that a new frame is assembled for each PHY mode, and then the assembled frame enters the inner loop which executes the artificial channel and frame reception.

The second most outer `for` loop is responsible for going thru all CP lengths. All this together simulated all the possible settings of the transceiver with all desired SNRs.

The BER is calculated for each combination and stored in a three-dimensional (CP, PHY mode, and SNR) global array `BER_log`. Because usually, desired BER values are very low (eg. 1e9 meaning one error bit per one billion bites) a statistic over a single frame is not sufficient.

For increased resolution, the final outer `for` loop was added. It executes `alterMessage` function which creates a new random payload and repeats the whole process according to a number stored in macro `SIMULATION_REPETITIONS`. All the resultant BER values are summed up in the `BER_log` array.

After that is done, another three nested `for` loops goes thru the `BER_log` array and divide each record by the number of repetition. This computes an average of all results.

At the end of the program, function `exportBER` which stores the results into a `.csv` file is executed.

The BER is considered to be one when the frame is not recognized at all or is not decodable. If the frame is decoded, function `calculateBER` compares the received payload with the transmitted one on a bit-to-bit basis. The BER is the number of unmatched bits to the total number of payload bits.

The liquid-dsp channel enables broad possibilities for setting different channel properties. In this version only SNR was taken into account, but for further investigation also frequency offsets, fading, or multi-path propagation modes could be set.

Because the simulation, especially with a high number of repetitions, takes a long time, a note is written to the terminal output each time the second outer `for` loop starts with an updated CP number.

A simulation with 1000 repetitions was launched across SNRs 0 to 25. Plots of the results are in figures 4.7 and 4.8. It is clearly visible that with a higher bit rate (used in higher-number PHY modes, see table 2.2) a higher SNR is required to achieve low BER.

There are no visible variations between different CPs. There would be interesting to play around with multipath parameters of the artificial channel and observe their impact on the performance.

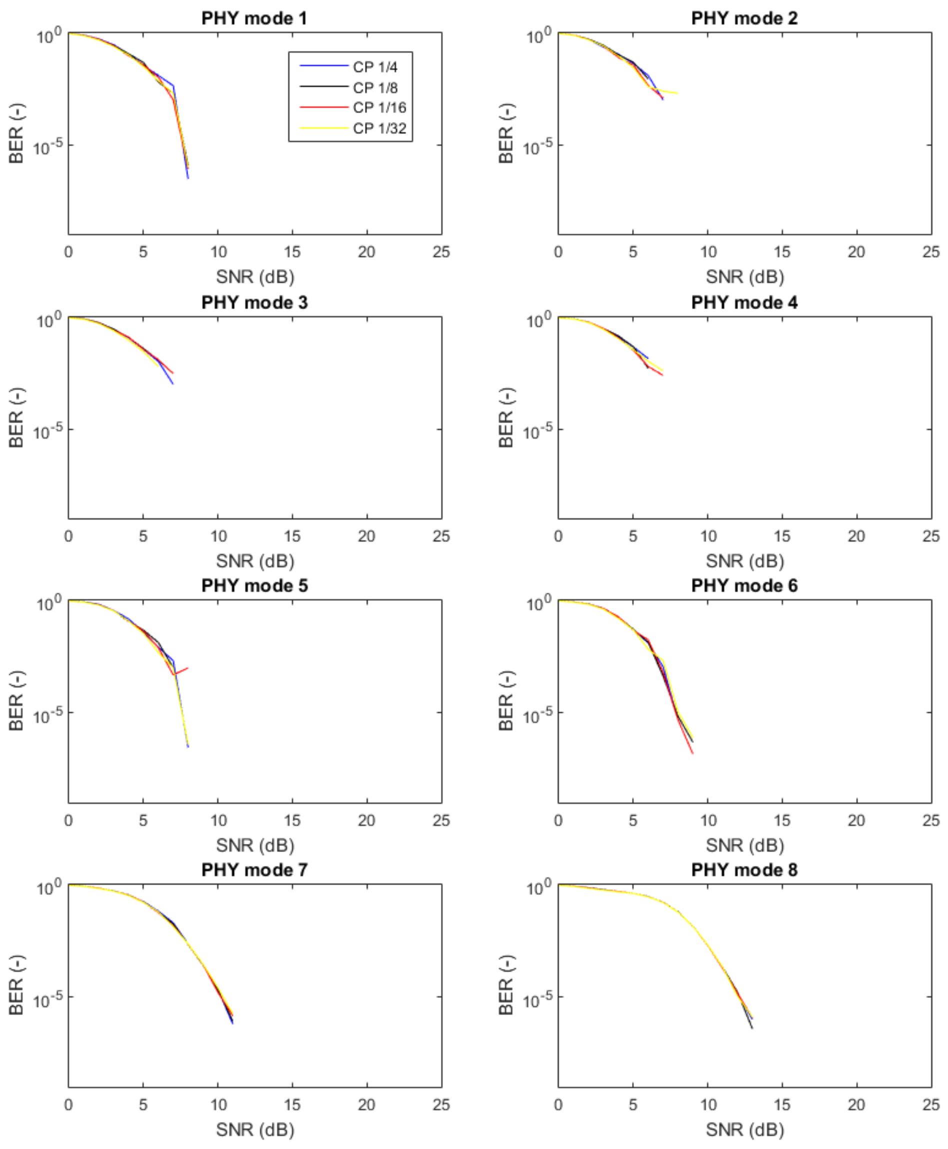


Figure 4.7: BER simulation outcome - PHY modes 1 to 8

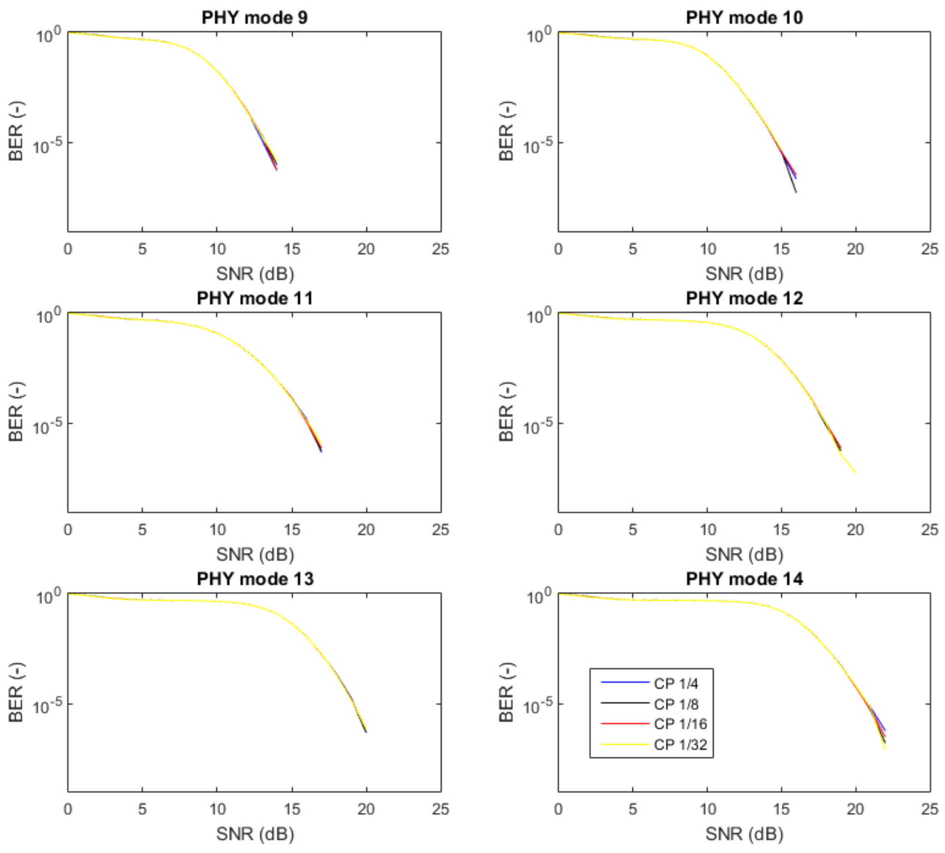


Figure 4.8: BER simulation outcome - PHY modes 9 to 14

CHAPTER 5

Function set

Initially, all the code was written in a linear manner such, that any revisions and modifications were progressively more difficult. The need for a new structure of the code was progressively raising. The code was reviewed and several functions were defined with the aim to create building blocks that make upcoming program versions and derivations easier to create, read and debug. These functions are described in this chapter.

5.1 Debug messages

One major thing that all the functions have in common is a condition `if(PRINT)`. The `PRINT` is a macro that can be set either to `true` or `false` depending if the debug messages are needed or not. The command following the condition prints into the terminal output name of the function where the program currently is and the last action that was performed in format
`functionName - last performed action`.

5.2 setSampleRate

```
void setSampleRate(int cyclic_prefix);
```

The sample rate of the SDR can not be hardcoded in the program because it needs to be adjusted together with the CP in order to maintain the duration of the frame at 10 ms. This function modifies directly a global variable `sampleRate`.

5.3 SDRinit

```
int SDRinit(double frequency, double sampleRate, int modeSelector,
double normalizedGain);
```

This function initializes the LimeSDR as described in 4.2.1.

5.4 defineFrameGenerator

```
ofdmflexframegen DefineFrameGenerator (int dfg_cycl_pref, int
dfg_PHYmode);
```

The frame generator object returned by the function is specified according to selected PHY mode and CP.

Although some parameters like the number of subcarriers, CP or taper length are direct parameters of the `ofdmflexframegen_create` function within the liquid library, for the full definition of the frame generator a lot of code is needed. Other parameters are encapsulated in properties structure `ofdmflexframegenprops_s` which needs to be created and set according to selected PHY mode and the subcarrier allocation array needs to be created and filled.

5.5 subcarrierAllocation

```
void subcarrierAllocation (unsigned char *array);
```

This function accepts the pointer to the allocation array as an input parameter, which is needed for the frame generator and synchronizer definition. The implemented scheme uses a fixed number of subcarriers so additional information is not needed. Inside the function a `for` loop is found which marks the entries in the array with zeros for unused subcarriers, ones for pilots, and twos for the data subcarriers.

5.6 frameAssemble

```
void frameAssemble(float *r_frame_buffer, int cyclic_prefix,
int phy_mode);
```

The frameAssemble is a large function accepting a pointer to buffer for real time-samples of the whole frame and CP with PHY mode. The function reads global variable `message` and do all the steps needed to fill the buffer with time samples ready to stream into LimeSDR.

5.7 frameReception

```
void frameReception(int cyclic_prefix);
```

The frameReception creates a synchronizer object of the liquid library. Then continuously translates real time-samples from SDR into complex samples for the synchronizer object and executes the synchronization. Every time the synchronizer recognizes a frame in the received signal, a callback function specified in the synchronizer object is invoked and executed.

5.8 frameSymbols

```
int frameSymbols(int cyclic_prefix);
```

This a very simple function that return number of OFDM symbols in one frame according to chosen CP using `switch - case` structure.

5.9 payloadLength

```
uint payloadLength(int cyclic_prefix, int phy_mode);
```

This function returns the length of payload according to selected CP and PHY mode.

A `switch - case` structure selects number of OFDM symbols that carries the payload. Then another `switch - case` computes how many bits are modulated on one subcarrier according to selected modulation and FEC. Both these numbers are multiplied together with the number of data subcarriers and divided by 8 to obtain bytes instead of bits.

5.10 complexSymbolBufferLength

```
uint complexSymbolBufferLength(int cyclic_prefix);
```

This function simply calculates and returns the number of complex samples per OFDM by the formula $(1 + CP) \times FFT_{length}$.

5.11 complexFrameBufferLength

```
uint complexFrameBufferLength(int cyclic_prefix);
```

This function returns the number of a complex symbol across the whole frame. It calls complexSymbolBufferLength and frameSymbols and multiplies their results.

5.12 printByteByByte

`void printByteByByte(unsigned int payload_len, unsigned char *transmitted, unsigned char *received);`

This function was defined for debugging of BER simulator code. It accepts pointers on transmitted and received char arrays with their length.

All the transmitted and receiver bytes are printed into the terminal as decimal numbers.

5.13 sendFrame

`void sendFrame(int cyclic_prefix, int phy_mode);`

The sendFrame function creates a buffer for storing all real time-samples of the frame, invokes `frameAssemble` function which fills that buffer, sets the sampling rate of the SDR and passes the buffer for transmission.

startSDRTXStream `int startSDRTXStream(int *tx_buffer, int FrameSampleCnt);`

This function passes the `tx_buffer` into the LimeSDR as described in 4.2.1.

5.14 artificialChannel

`void artificialChannel(int cyclic_prefix);`

This function was created for the BER simulation. It works with global buffers of real time-samples `before_cahnnel_buffer` and `after_cahnnel_buffer`.

The object `channel_cccf` is created and SNR is set according to global variable `artificial_SNR`, then the first buffer is converted to complex numbers. The channel is then executed over the complex buffer and then converted again into real buffer for `frameReception` function.

5.15 calculateBER

```
float calculateBER(unsigned int payload_len, string transmitted,
unsigned char *received);
```

This function is meant to be called from a callback function for BER simulation. The callback function must handle cases when the frame was recognized but synchronization was not successful. The function itself goes bit-by-bit over the payload and compares the transmitted message with the received one. At the end divides the number of errors by the number of bits and pass the result.

5.16 exportBER

```
int exportBER(void);
```

Calling this function exports the `BER_log` array into a .csv file in folder `BER_calculation_out` called `BER_simulation_currentDateAndTime.csv`. Before the function is called is necessary to divide all the values of `BER_log` by the number of repetitions of the BER calculation.

5.17 alterMessage

```
string alterMessage(int payload_len);
```

While the channel is simulated many times for increasing the resolution of BER results, this function alters the global string `message` for increasing the credibility of the simulation.

CHAPTER 6

Conclusion

This work investigates the telecommunication standard IEEE 802.22 WRAN and on its basis creates a new standard for access to the amateur radio network HAMNET. This new standard was named Amateurradio Wireless Regional Area Network (AWRAN), and is designed to operate in the 6 m, 2 m, and 70 cm bands with 2 MHz, 4 MHz, and 10 MHz BW. The OFDM parameters of WRAN have been adapted to these BWs.

AWRAN can accommodate up to 63 subscribers under a single BS and allows up to four overlapping BSs to operate simultaneously on a single band. For this purpose, fourteen management messages have been defined for communication between BSs and subscribers providing efficient allocation of the available spectrum.

The simplified frame structure of the defined standard has been implemented on the SDR using the liquid-dsp library. A test link has been established, which unfortunately is not fully functional yet. Only narrowband FSK transmission has been successfully adopted. For wideband OFDM transmission, the power is too spread out for successful frame reception.

For full-scale AWRAN implementation, the chosen liquid-dsp library is unsuitable. However, the program is divided into basic control functions that will allow easy adaptation of new changes.

A program simulating the BER dependence on SNR was prepared and the first results are included in this thesis. This program can be easily modified to test other wireless channel phenomena such as multipath propagation or fading.

List of Figures

List of Tables

List of Algorithms

ADC	analog to digital converter
AM	amplitude modulation
AMI	alternate mark inversion
APSK	amplitude and phase-shift keying
ASK	amplitude-shift keying
AWGN	additive white Gaussian noise
A-WRAN	advanced WRAN
AWRAN	Amateurradio Wireless Regional Area Network
BER	bit error rate
BPSK	binary phase-shift keying
BS	base station
BW	bandwidth
CDMA	code division multiple access
CP	cyclic prefix
CPE	customer premises equipment
DAC	digital to analog converter
DCD	downstream channel descriptor

DFT	discrete Fourier transform
DPSK	differential phase-shift keying
DS	downstream
$\mathbf{E_b}$ / $\mathbf{N_0}$	energy per bit to noise power spectral density ratio
ECC	error correction codes
EDC	error detection codes
EIRP	equivalent isotropic radiated power
EM	electromagnetic
FA	frame allocation
FDM	frequency division multiplexing
FDMA	frequency division multiple access
FEC	forward error correction
FFT	fast Fourier transform
FCH	frame control header
FM	frequency modulation
FMW	frame management window
FPGA	field programmable gate array
FPRF	field porgrammable radio frequency
FSA	free space attenuation
FSK	frequency shift keying
FT	Fourier transform
GPIO	general purpose input-output
HAMNET	Highspeed Amateurradio Multimedia NETwork

I	in-phase
IC	integrated circuit
ID	identification
IFFT	inverse fast Fourier transform
IO	input-output
IoT	Internet of Things
IQ	in-phase and quadrature
ISI	inter symbol interference
LFSR	linear feedback shift register
LOS	line-of-sight
LTS	long training sequence
MAN	Metropolitan Area Network
MC-CDMA	multi carrier code division multiple access
MIMO	multiple-input multiple-output
NLOS	non line-of-sight
NRZ	non-return to zero
OFDM	Orthogonal Frequency Division Multiplex
OFDMA	Orthogonal Frequency Division Multiple Access
OOK	on-off keying
OW	opportunistic window
PA	power amplifier
PAPR	peak to average power ratio
PCB	printed circuit board

PCM	pulse code modulation
PHY	physical layer
PLL	phase locked loop
PM	phase modulation
PPM	pulse position modulation
PRBS	pseudo-random binary sequence
PSK	phase-shift keyinq
PWM	pulse width modulation
Q	quadrature
QAM	quadrature amplitude-shift keying
QPSK	quadrature phase-shift keying
RAN	Regional Area Network
RF	radio frequency
RFIC	radio-frequency integrated circuit
RPi	Raspberry Pi
RX	receive
RZ	return to zero
SCW	self-coexistence window
SDR	software-defined radio
SEC-DED	single error correction - double error detection
SF	superframe
SCH	subchannel
SCH	superframe control header

SM	spectrum manager
SNR	signal-to-noise ratio
STS	short training sequence
SW	software
TDM	time division multiplexing
TDMA	time division multiple access
TPC	transmit power control
TTG	transmit-receive turnaround gap
TV	television
TX	transmit
UCD	upstream channel descriptor
US	upstream
USB	universal serial bus
WAN	Wide Area Network
WRAN	Wireless Regional Area Network

Bibliography

[1] Stefan Aust, R. Venkatesha Prasad, and Ignas G. M. M. Niemegeers. “IEEE 802.11ah: Advantages in standards and further challenges for sub 1 GHz Wi-Fi”. In: (2012), pp. 6885–6889. DOI: 10.1109/ICC.2012.6364903.

[2] Rudolf Baumgartner. *translation of Die HB9CV Richtstrahlantenne*. [Online; accessed 5-August-2022]. DL1CU. 1969. URL: https://www.ok2kkw.com/hb9cv/hb9cv_1969.htm.

[3] Dan Dong et al. “Joint source–channel rate allocation with unequal error protection for space image transmission”. In: *International Journal of Distributed Sensor Networks* 13 (July 2017), p. 155014771772114. DOI: 10.1177/1550147717721145.

[4] Adriana B. Flores et al. “IEEE 802.11af: a standard for TV white space spectrum sharing”. In: *IEEE Communications Magazine* 51.10 (2013), pp. 92–100. DOI: 10.1109/MCOM.2013.6619571.

[5] Joseph D. Gaeddert. 2012. URL: https://www.liquidsdr.org/downloads/liquid-dsp-1.0.0.pdf.

[6] Joseph D. Gaeddert. *liquidsdr.org making software radio portable since 2007*. URL: https://liquidsdr.org/. (accessed: 05.04.2021).

[7] Paul Heckbert. *Fourier Transforms and the Fast Fourier Transform (FFT) Algorithm*. Feb. 1995.

[8] Systems Iain Explains Signals and Digital Comms. *What are Fast Fading and Slow Fading?* [Online; accessed 20-July-2022]. Youtube. 2020. URL: `https://www.youtube.com/watch?v=Tm-Uyajcuqs`.

[9] Systems Iain Explains Signals and Digital Comms. *What are Flat Fading and Frequency Selective Fading?* [Online; accessed 20-July-2022]. Youtube. 2020. URL: `https://www.youtube.com/watch?v=KiKPFT4rtHg`.

[10] "IEEE Standard - Information Technology-Telecommunications and information exchange between systems-Wireless Regional Area Networks-Specific requirements-Part 22: Cognitive Wireless RAN MAC and PHY specifications: Policies and Procedures for Operation in the Bands that Allow Spectrum Sharing where the Communications Devices May Opportunistically Operate in the Spectrum of Primary Service". In: *IEEE Std 802.22-2019 (Revision of IEEE Std 802.22-2011)* (2020), pp. 1–1465. DOI: `10.1109/IEEESTD.2020.9086951`.

[11] Bernhard Isemann. *RPX-100.* [Online; accessed 20-July-2022]. Austrian Radio Amateur Association. 2022. URL: `https://rpx-100.net/#`.

[12] Tao Jiang and Yiyan Wu. "An Overview: Peak-to-Average Power Ratio Reduction Techniques for OFDM Signals". In: *IEEE Communications Magazine* 54.2 (2008), pp. 257–268. DOI: `10.1109/TBC.2008.915770`.

[13] Jaroslav Láčik. *Antennas and Radio Links - Lecture 9: Propagation of Radio Waves for Terrestrial Radio Links, Surface and Space Wave.* Apr. 2020.

[14] Lime Microsystems Limited. *Lime Suite.* URL: `https://wiki.myriadrf.org/Lime_Suite`. (accessed: 22. 01. 2021).

[15] Lime Microsystems Limited. *LimeSDR - USB - hardware description.* URL: `https://wiki.myriadrf.org/LimeSDR-USB_hardware_description`. (accessed: 29. 03. 2022).

[16] Lime Microsystems Limited. *LMS7002M - FPRF MIMO Transceiver IC With Integrated Microcontroller.* URL: https://cz.mouser.com/datasheet/2/982/LMS7002M-Data-Sheet-v3.1r00-1600568.pdf. (accessed: 29. 03. 2022).

[17] Wireless Excellence Limited. *Comparing Microwave Links using 512-QAM, 1024-QAM, 2048-QAM, 4096-QAM.* https://www.microwave-link.com/tag/4096qam/. [Online; accessed 10-August-2022]. 2018.

[18] *LMS API - Quick start guide.* [Online; accessed 5-December-2020]. lime microsystems. 2017. URL: https://usermanual.wiki/Document/lms7apiquickstartguide.1805960724.pdf.

[19] Raspberry Pi Ltd. *Raspberry Pi Foundation.* URL: https://www.raspberrypi.org/about/. (accessed: 25.03.2022).

[20] Martha C. Paredes Paredes and M. Julia Fenández-Getino García. *The Problem of Peak-to-Average Power Ratio in OFDM Systems.* URL: https://arxiv.org/pdf/1503.08271.pdf. (accessed: 15. 07. 2022).

[21] Frode Bøhagen Per Hjalmar Lehne. *OFDM(A) for wireless communication.* RI Research Report. Telenor, 2008. ISBN: 82-423-0614-1.

[22] Dragan Poljak and Mario Cvetković. *Chapter 2 - Theoretical Background: an Outline of Computational Electromagnetics (CEM).* Ed. by Dragan Poljak and Mario Cvetković. 2019. DOI: https://doi.org/10.1016/B978-0-12-816443-3.00010-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128164433000108.

[23] GNU Radio project. *GNU Radio Manual and C++ API Reference.* URL: https://www.gnuradio.org/doc/doxygen/page_components.html#components_blocks. (accessed: 06. 07. 2022).

[24] Sohangjot Kaur Randhawa and Prof. Daljeet Singh Bajwa. “PAPR Reduction in OFDM using PTS Technique”. In: *International Journal of Engineering Research Technology* 6.7 (2017), pp. 466–468. ISSN: 2278-0181.

[25] Solwise. *Surface elevation tool: Solwise ltd,* URL: `https://www.solwise.co.uk/wireless-elevationtool.html`. (accessed: 07. 04. 2022).

[26] C. R. Stevenson et al. “IEEE 802.22: The first cognitive radio wireless regional area network standard”. In: *IEEE Communications Magazine* 47.1 (2009), pp. 130–138. DOI: `10.1109/MCOM.2009.4752688`.

[27] Justin Thiel. *Metropolitan and Regional Wireless Networking.* URL: `https://www.cse.wustl.edu/~jain/cse574-06/ftp/wimax/`. (accessed: 05.04.2021).

[28] Ch.Radika Venkateswarlu S. Rani. “Channel Modelling- Parameters and Conditions to be Considered”. In: *International Journal of Engineering and Manufacturing Science* 7.2 (2017), pp. 1–8. ISSN: 2249-3115.

[29] Wikimedia. *hb9cv.svg.* `https://commons.wikimedia.org/wiki/File:Hb9cv.svg`. [Online; accessed 10-August-2022]. 2011.

[30] Wikipedia. *Additive white Gaussian noise — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Additive%20white%20Gaussian%20noise&oldid=1051832268`. [Online; accessed 10-August-2022]. 2022.

[31] Wikipedia. *Amateur radio — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Amateur%20radio&oldid=1101690277`. [Online; accessed 11-August-2022]. 2022.

[32] Wikipedia. *Block code — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Block%20code&oldid=1079065291`. [Online; accessed 10-August-2022]. 2022.

[33] Wikipedia. *Carrier-to-noise ratio.* URL: https://en.wikipedia.org/wiki/Carrier-to-noise_ratio. (accessed: 12. 07. 2022).

[34] Wikipedia. *Colors of noise — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Colors%20of%20noise&oldid=1100630200. [Online; accessed 10-August-2022]. 2022.

[35] Wikipedia. *Communication channel — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Communication%20channel&oldid=1096192105. [Online; accessed 31-July-2022]. 2022.

[36] Wikipedia. *Convolutional code — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Convolutional%20code&oldid=1100051594. [Online; accessed 10-August-2022]. 2022.

[37] Wikipedia. *Crest factor.* URL: https://en.wikipedia.org/wiki/Crest_factor. (accessed: 15. 07. 2022).

[38] Wikipedia. *Error correction code — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Error%20correction%20code&oldid=1098274639. [Online; accessed 10-August-2022]. 2022.

[39] Wikipedia. *Fading — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Fading&oldid=1087212868. [Online; accessed 31-July-2022]. 2022.

[40] Wikipedia. *Free-space path loss — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Free-space%20path%20loss&oldid=1080664589. [Online; accessed 31-July-2022]. 2022.

[41] Wikipedia. *Hadamard matrix — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Hadamard%20matrix&oldid=1076843854. [Online; accessed 31-July-2022]. 2022.

[42] Wikipedia. *Multi-carrier code-division multiple access — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Multi-carrier%20code-division%20multiple%20access&oldid=972219240. [Online; accessed 10-August-2022]. 2022.

[43] Wikipedia. *Phase-shift keying — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Phase-shift%20keying&oldid=1101144122. [Online; accessed 10-August-2022]. 2022.

[44] Wikipedia. *Punctured code — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Punctured%20code&oldid=1093456207. [Online; accessed 10-August-2022]. 2022.

[45] Wikipedia. *Quantization (signal processing) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Quantization%20(signal%20processing)&oldid=1084500420. [Online; accessed 10-August-2022]. 2022.

[46] Wikipedia. *Raspberry Pi*. URL: https://en.wikipedia.org/wiki/Raspberry_Pi. (accessed: 25.03.2022).

[47] Wikipedia. *Spectral density — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Spectral%20density&oldid=1103158362. [Online; accessed 10-August-2022]. 2022.

Attachments

RPX-100-transceiver.h

RPX-100-transceiver.cpp

RPX-100-BER_simulator.h

RPX-100-BER_simulator.cpp

```
/*******************************************************************************
 * C++ source of RPX-100-transceiver
 *
 * File:   RPX-100-transceiver.h
 * Author: Bernhard Isemann
 *         Marek Honek
 *
 * Created on 21 Jul 2022, 16:20
 ******************************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sstream>
#include <syslog.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <cstdio>
#include <ctime>
#include <math.h>
#include <complex.h>
#include <time.h>
#include <chrono>
#include <cstring>
#include <bitset>
#include "stuff/ini.h"
#include "stuff/log.h"
#include <chrono>
#include "lime/LimeSuite.h"
#include "liquid/liquid.h"
#include "stuff/ServerSocket.h"
#include "stuff/SocketException.h"
#include <iterator>
#include <signal.h>
#include "stuff/Util.h"
#include "stuff/WebSocketServer.h"
#include <correct.h>
#pragma once


lms_device_t *device = NULL;


#define SUBCARRIERS 1024
#define DATACARRIERS 480


#define TX_6m_MODE 6
#define RX_MODE 0

// print each step for debuggigng
#define PRINT false


// Radio Frontend - Define GPIO settings for CM4 hat module
uint8_t setRX = 0x18;       // GPIO0=LOW - RX, GPIO3=HIGH - PTT off,
uint8_t setTXDirect = 0x0F; // GPIO0=HIGH - TX, GPIO3=HIGH - PTT off, GPIO1=HIGH, GPIO2=HIGH
uint8_t setTX6m = 0x0D;     // GPIO0=HIGH - TX, GPIO3=HIGH - PTT off, GPIO1=LOW, GPIO2=LOW
uint8_t setTX2m = 0x09;     // GPIO0=HIGH - TX, GPIO3=HIGH - PTT off, GPIO1=LOW, GPIO2=HIGH
uint8_t setTX70cm = 0x0B;   // GPIO0=HIGH - TX, GPIO3=HIGH - PTT off, GPIO1=HIGH, GPIO2=LOW

uint8_t setTXDirectPTT = 0x07; // GPIO0=HIGH - TX, GPIO3=LOW - PTT on, GPIO1=HIGH, GPIO2=HIGH
uint8_t setTX6mPTT = 0x05;     // GPIO0=HIGH - TX, GPIO3=LOW - PTT on, GPIO1=LOW, GPIO2=LOW
```

```
uint8_t setTX2mPTT = 0x01;     // GPIO0=HIGH - TX, GPIO3=LOW - PTT on, GPIO1=LOW, GPIO2=HIGH
uint8_t setTX70cmPTT = 0x03;   // GPIO0=HIGH - TX, GPIO3=LOW - PTT on, GPIO1=HIGH, GPIO2=LOW

string modeName[9] = {"RX", "TXDirect", "TX6m", "TX2m", "TX70cm", "TXDirectPTT", "TX6mPTT", "TX2mPTT", "TX70cmPTT"};
uint8_t modeGPIO[9] = {setRX, setTXDirect, setTX6m, setTX2m, setTX70cm, setTXDirectPTT, setTX6mPTT, setTX2mPTT, setTX70cmPTT};


int error();
string exec(string command);

// Log facility
void print_gpio(uint8_t gpio_val);
stringstream msgSDR;
stringstream HEXmsg;

// Initialize sdr buffers
liquid_float_complex complex_i(0,1);
int samplesRead = 1048;

bool rxON = true;
bool txON = true;

int startSDRTXStream(float *tx_buffer, int FrameSampleCnt);
void frameAssemble(float *r_frame_buffer, int cyclic_prefix, int phy_mode, string message);
void subcarrierAllocation (unsigned char *array);
ofdmflexframegen DefineFrameGenerator (int dfg_cycl_pref, int dfg_PHYmode);
int frameSymbols(int cyclic_prefix);
uint complexFrameBufferLength(int cyclic_prefix);
uint complexSymbolBufferLength(int cyclic_prefix);
uint payloadLength(int cyclic_prefix, int phy_mode);
double setSampleRate(int cyclic_prefix);
void sendFrame(int cyclic_prefix, int phy_mode, string message);
void frameReception(int cyclic_prefix);
int SDRinit(double frequency, double sampleRate, int modeSelector, double normalizedGain);
int startSDRTXStream(int *tx_buffer, int FrameSampleCnt);



int callbackWhatsReceived(unsigned char *_header,
               int _header_valid,
               unsigned char *_payload,
               unsigned int _payload_len,
               int _payload_valid,
               framesyncstats_s _stats,
               void *_userdata);


```

```cpp
/*******************************************************************************
 * C++ source of RPX-100-transceiver
 *
 * File:   RPX-100-transceiver.cpp
 * Author: Bernhard Isemann
 *         Marek Honek
 *
 * Created on 21 Jul 2022, 16:20
 *
 * Predecessor  RPX-100-synchronizer.cpp
 ******************************************************************************/

#include "RPX-100-transceiver.h"

using namespace std;


int main(int argc, char *argv[])
{
    if (PRINT)
        cout << "main - program started" << endl;

    // Default start values
    string mode = "RX";
    int cycl_prefix = 4;
    int phy_mode = 1;
    string message = "OE1XTU AWRAN at 52.8 MHz";


    // Default SDR values
    double sampleRate = 3328000;
    double normalizedGain = 1;
    double frequency = 52.8e6;
    int modeSel = RX_MODE;


    if (argc == 1)
    {
        cout << "Starting RPX-100-transciever with default settings:\n";
        cout << "Mode: RX" << endl;
        cout << "Cyclic prefix: 1/4" << endl;
        cout << endl;
        cout << "type \033[36m'RPX-100-transciever help'\033[0m to see all options!"
<< endl;
    }
    else if (argc >= 2)
    {
        for (int c = 1; c < argc; c++)
        {
            switch (c)
            {
            case 1:
                mode = (string)argv[c];
                if (mode == "RX")
                {
                    cout << "Starting RPX-100-transciever with following setting:\n";
                    cout << "Mode: " << argv[c] << endl;
                    modeSel = RX_MODE;
                }
                else if (mode == "TX6mPTT")
                {
                    cout << "Starting RPX-100-transciever with following setting:\n";
                    cout << "Mode: " << argv[c] << endl;
                    modeSel = TX_6m_MODE;
                }
                else if (mode == "help")
                {
                    cout << "Options for starting RPX-100-transciever" << endl;
                    cout << endl;
                    cout << "\033[36mMODE\033[0m:" << endl;
                    cout << "     \033[32mRX\033[0m for receive mode" << endl;
                    cout << endl;
                    cout << "     \033[31mTX6mPTT\033[0m for transmit mode with PTT
with bandpass filter for 50-54 MHz" << endl;
                    cout << endl;
                    cout << "\033[36mCYCLIC PREFIX\033[0m:" << endl;
                    cout << "     \033[32m4, 8, 16 or 32\033[0m for 1/n cyclic prefix" << endl;
                    cout << endl;
                    cout << "\033[36mPHY MODE\033[0m:" << endl;
                    cout << "     \033[32mNumber 1 to 14\033[0m for PHY mode (applies only for TX mode)" << endl;
                    cout << endl;
                    cout << "\033[36mMESSAGE\033[0m:" << endl;
                    cout << "     \033[32mString to be transmitted\033[0m (applies only for TX mode)" << endl;
                    cout << endl;
                    return 0;
                }
                else
                {
                    cout << "Wrong settings, please type  \033[36m'RPX-100-transciever help'\033[0m to see all options !" << endl;
                    return 0;
                }
                break;
            case 2:
                cycl_prefix = stoi((string)argv[c]);
                if ((cycl_prefix == 4) || (cycl_prefix == 8) || (cycl_prefix == 16) || (cycl_prefix == 32))
                {
                    cout << "     cyclic prefix: " << cycl_prefix << endl;
                    cout << endl;
                }
                else
                {
                    cout << "Wrong settings, please type  \033[36m'RPX-100-transciever help'\033[0m to see all options !" << endl;
                    return 0;
                }
                break;
            case 3:
                phy_mode  = stoi((string)argv[c]);
                if (phy_mode > 0 && phy_mode < 15)
                {
                    cout << "     PHY mode: " << phy_mode << endl;
                    cout << endl;
                }
                else
                {
                    cout << "Wrong settings, please type  \033[36m'RPX-100-transciever help'\033[0m to see all options !" << endl;
                    return 0;
                }
                break;
            case 4:
                message = (string)argv[c];
                break;
            }
        }
    }

    LogInit();
    if (PRINT)
        cout << "main - logger initalized" << endl;

    Logger("RPX-100-synchronizer was started.\n");
    msgSDR << "Mode: " << modeSel << endl;
    msgSDR << "Cyclic prefix: " << cycl_prefix << endl;
    msgSDR << "PHY mode: " << phy_mode << endl;
    Logger(msgSDR.str());
    msgSDR.str("");
    if (PRINT)
        cout << "main - first log message saved" << endl;

    sampleRate = setSampleRate(cycl_prefix);
    if (sampleRate == -1)
        return -1;


    if (SDRinit(frequency, sampleRate, modeSel, normalizedGain) != 0)
    {
        msgSDR.str("");
        msgSDR << "ERROR: " << LMS_GetLastErrorMessage();
        Logger(msgSDR.str());
        cout << msgSDR.str() << endl;
    }

    if (device == NULL)
    {
        cout << "main - device is NULL" << endl;
    }
    else
    {
        if (modeSel == TX_6m_MODE)
        {
            sendFrame(cycl_prefix, phy_mode, message);
            if (PRINT)
                cout <<"main - sendFrame exitted" << endl;
        }
        if (modeSel == RX_MODE)
        {
            frameReception(cycl_prefix);
            if (PRINT)
                cout <<"main - frameReception exitted" << endl;
        }
    }

    SDRinit(frequency, sampleRate, RX_MODE, normalizedGain);

    // Close device
    if (LMS_Close(device) == 0)
    {
        msgSDR.str("");
        msgSDR << "Closed" << endl;
        Logger(msgSDR.str());
    }


    return(0);
}


void sendFrame(int cyclic_prefix, int phy_mode, string message)
{
    if (PRINT)
        cout << "sendFrame - sendFrame started - cyclic_prefix: " << cyclic_prefix <<
"; phy_mode: " << phy_mode << endl;

    int tx_buffer_len = 2*complexFrameBufferLength(cyclic_prefix);
    if (PRINT)
        cout << "sendFrame - complexFrameBufferLength exitted - buffer_len: " <<
tx_buffer_len << endl;

    float tx_buffer[tx_buffer_len];  //buffer for whole frame
    if (PRINT)
        cout << "sendFrame - buffer initialized" << endl;

    frameAssemble(tx_buffer, cyclic_prefix, phy_mode, message);
    if (PRINT)
    {
        cout << "sendFrame - frameAssemble exitted" << endl;
        cout << "sendFrame - tx_buffer[0]: " << tx_buffer[0] << endl;
    }

    if (PRINT)
        cout << "sendFrame - setSampleRate exitted" << endl;


    if (PRINT)
        cout << "sendFrame - SDR has been set" << endl;


    startSDRTXStream(tx_buffer, complexFrameBufferLength(cyclic_prefix));


    if (PRINT)
    {
```

```cpp
        cout << "sendFrame - frame in tx_buffer has been transmitted" << endl;
    }

}

void frameAssemble(float *r_frame_buffer, int cyclic_prefix, int phy_mode, string message)
{
    if (PRINT)
        cout << "frameAssemble - frameAssemble started" << endl;

    liquid_float_complex complex_i(0, 1);

    unsigned int payload_len = payloadLength(cyclic_prefix, phy_mode); //depends on PHY mode and cyclic prefix
    if (PRINT)
    {
        cout << "frameAssemble - payloadLength exitted" << endl;
        cout << "frameAssemble - payload_len: " << payload_len << endl;
    }

    uint c_buffer_len = complexSymbolBufferLength(cyclic_prefix); //depends on cyclic prefix
    if (PRINT)
        cout << "frameAssemble - complexSymbolBufferLength exitted" << endl;

    // create frame generator
    ofdmflexframegen fg = DefineFrameGenerator(cyclic_prefix, phy_mode);
    if (PRINT)
    {
        cout << "frameAssemble - DefineFrameGenerator exitted, frame generator setted" << endl;
        ofdmflexframegen_print(fg);
    }

    // buffers
    liquid_float_complex c_buffer[c_buffer_len]; // time-domain buffer
    unsigned char header[8];                     // header data
    unsigned char payload[payload_len] = {};          // payload data
    if (PRINT)
        cout << "frameAssemble - header and payload buffers initialized" << endl;

    // ... initialize header/payload ...
    strcpy((char *)payload, message.c_str());

    header[0] = '0';
    header[1] = '0';
    header[2] = '0';
    header[3] = '0';
    header[4] = '0';
    header[5] = '0';
    header[6] = '0';
    header[7] = '0';

    if (PRINT)
    {
        cout << "frameAssemble - header and payload written" << endl;
        cout << "frameAssemble - payload: " << payload << endl;  // prints as text
        cout << "frameAssemble - payload: ";                      // prints as numbers
        for (int i=0; i<payload_len; i++)
        {
            cout << unsigned(payload[i]) << " ";
        }
        cout << endl << endl;
    }


    // assemble frame
    ofdmflexframegen_assemble(fg, header, payload, payload_len);
    if (true)
    {
        cout << "frameAssemble - frame assembled" << endl;
        ofdmflexframegen_print(fg);
    }

    int last_symbol = 0;
    int i = 0;
    int l = 0;

    while (!last_symbol)
    {
        // generate each OFDM symbol
        last_symbol = ofdmflexframegen_write(fg, c_buffer, c_buffer_len);
        if (PRINT)
        {
            cout << "frameAssemble - symbol " << l+1 << " written" << endl;
            cout << "frameAssemble - c_buffer[0]: " << c_buffer[0] << endl;
            cout << "frameAssemble - last_symbol value: " << last_symbol << endl;
        }


        if (!last_symbol)
        {
            if (PRINT)
                cout << "frameAssemble - starting complex to real buffer conversion"
<< endl;
            for (i = 0; i < c_buffer_len; i++)
            {
                r_frame_buffer[l*2*c_buffer_len+2*i]=c_buffer[i].real();
                r_frame_buffer[l*2*c_buffer_len+2*i+1]=c_buffer[i].imag();
            }
        }
        if (PRINT)
        {
            cout << "frameAssemble - r_frame_buffer[0]: " << r_frame_buffer[0] <<
endl;
            cout << "frameAssemble - exiting complex to real buffer conversion" <<
endl;
        }

        l++;
    }
    ofdmflexframegen_destroy(fg);
}

void frameReception(int cyclic_prefix)
{
    if (PRINT)
        cout << "frameReception - frameReception started" << endl;

    liquid_float_complex complex_i (0, 1);

    // Initialize stream
    lms_stream_t streamId;                          //stream structure
    streamId.channel = 0;                           //channel number
    streamId.fifoSize = 1024 * 1024;                //fifo size in samples
    streamId.throughputVsLatency = 1.0;             //optimize for max throughput
    streamId.isTx = false;                          //RX channel
    streamId.dataFmt = lms_stream_t::LMS_FMT_F32; //12-bit integers
    if (LMS_SetupStream(device, &streamId) != 0)
        error();


    int c_sync_buffer_len = complexFrameBufferLength(cyclic_prefix); //synchronizer
buffer can be  of arbitrary length
    liquid_float_complex c_sync_buffer[c_sync_buffer_len];
    float r_sync_buffer[c_sync_buffer_len*2];
    if (PRINT)
        cout << "frameReception - buffers initialized" << endl;

    unsigned char allocation_array[SUBCARRIERS];    // subcarrier allocation array
(null/pilot/data)
    subcarrierAllocation(allocation_array);
    if (PRINT)
        cout << "frameReception - subcarrierAllocation exited; allocation_array
defined" << endl;

    unsigned int cp_len = (int)SUBCARRIERS / cyclic_prefix; // cyclic prefix length
    unsigned int taper_len = (int)cp_len / 4;           // taper length

    ofdmflexframesync fs = ofdmflexframesync_create(SUBCARRIERS, cp_len, taper_len,
allocation_array, callbackWhatsReceived, NULL);
```

```
    //ofdmflexframesync fs = ofdmflexframesync_create(SUBCARRIERS, cp_len, taper_len, allocation_array, callbackBERCalculation, NULL);
    if (PRINT)
        cout << "frameReception - frame synchronizer created" << endl;

    // Start streaming
    LMS_StartStream(&streamId);

    cout << "frameReception - entering reception loop" << endl;
    while(rxON)
    {
        //Receive samples
        LMS_RecvStream(&streamId, r_sync_buffer, c_sync_buffer_len, NULL, 1000); //should work, for now replaced by tx_buffer
        //I and Q samples are interleaved in r_sync_buffer: IQIQIQ...

        if (PRINT)
        {
            cout << "frameReception - r_sync_buffer filled" << endl;
            cout << "frameReception - r_sync_buffer[0]: " << r_sync_buffer[0] << endl;
        }

        for (int i = 0; i < c_sync_buffer_len; i++)
        {
            c_sync_buffer[i]=r_sync_buffer[2*i]+r_sync_buffer[2*i+1] * complex_i.imag();
        }
        if (PRINT)
            cout << "frameReception - real buffer converted to complex buffer" << endl;

        // receive symbol (read samples from buffer)
        ofdmflexframesync_execute(fs, c_sync_buffer, c_sync_buffer_len);
        if (PRINT)
            cout << "frameReception - synchronization ended" << endl;
    }
    ofdmflexframesync_destroy(fs);

    // Stop streaming
    LMS_StopStream(&streamId);            // stream is stopped but can be started again with LMS_StartStream()
    LMS_DestroyStream(device, &streamId); // stream is deallocated and can no longer be used
}


void print_gpio(uint8_t gpio_val)
{
    for (int i = 0; i < 8; i++)
    {
        bool set = gpio_val & (0x01 << i);
        msgSDR << "GPIO" << i << ": " << (set ? "High" : "Low") << endl;
        Logger(msgSDR.str());
        msgSDR.str("");
    }
}

double setSampleRate(int cyclic_prefix)
{
    if (PRINT)
        cout << "setSampleRate - setSampleRate started" << endl;
    switch (cyclic_prefix)
    {
    case 4:
        return 3328000;
    case 8:
        return 3225600;
    case 16:
        return 3264000;
    case 32:
        return 3273600;
    }
    return -1;
}

```

```
ofdmflexframegen DefineFrameGenerator (int dfg_cycl_pref, int dfg_PHYmode)
{
    if (PRINT)
        cout << "DefineFrameGenerator - DefineFrameGenerator started" << endl;

    // initialize frame generator properties
    ofdmflexframegenprops_s fgprops;
    ofdmflexframegenprops_init_default(&fgprops);
    fgprops.check = LIQUID_CRC_NONE;
    fgprops.fec1 = LIQUID_FEC_NONE;

    unsigned int cp_len = (int)SUBCARRIERS / dfg_cycl_pref; // cyclic prefix length
    unsigned int taper_len = (int)cp_len / 4;          // taper length

    switch (dfg_PHYmode)
    {

    case 1:
        fgprops.fec0 = LIQUID_FEC_NONE;
        fgprops.mod_scheme = LIQUID_MODEM_PSK2;
        break;
    case 2:
        fgprops.fec1 = LIQUID_FEC_REP3;
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 3:
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 4:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P23;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 5:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P34;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
         break;
    case 6:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P56;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 7:
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 8:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P23;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 9:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P34;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 10:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P56;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 11:
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    case 12:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P23;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    case 13:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P34;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    case 14:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P56;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    default:
        return NULL;
    }

    unsigned char allocation_array[SUBCARRIERS]; // subcarrier allocation array(null/pilot/data)
    subcarrierAllocation(allocation_array);
    if (PRINT)
        cout << "DefineFrameGenerator - subarrierAllocation exitted" << endl;

    return ofdmflexframegen_create(SUBCARRIERS, cp_len, taper_len, allocation_array, &fgprops);
}

void subcarrierAllocation (unsigned char *array)
{
    if (PRINT)
        cout << "subcarrierAllocation - subcarrierAllocation started" << endl;
    for (int i = 0; i < 1024; i++)
    {
        if (i < 232)
            array[i] = 0; // guard band

        if (231 < i && i < 792)
            if (i % 7 == 0)
                array[i] = 1; // every 7th carrier pilot
            else
                array[i] = 2; // rest data

        if (i > 791)
            array[i] = 0; // guard band
    }
}

int frameSymbols(int cyclic_prefix)
{
    if (PRINT)
        cout << "frameSymbols - frameSymbols started" << endl;
    int symbolCnt;
    switch (cyclic_prefix)
    {
    case 4:
        symbolCnt = 22+4;
        break;
    case 8:
        symbolCnt = 24+4;
        break;
    case 16:
        symbolCnt = 26+4;
        break;
    case 32:
        symbolCnt = 27+4;
        break;
    default:
        symbolCnt = 0;
    }

    if (PRINT)
        cout << "frameSymbols - symbolCnt: " << symbolCnt << endl;
    return symbolCnt;
}

uint complexFrameBufferLength(int cyclic_prefix)
{
    if (PRINT)
        cout << "complexFrameBufferLength - complexFrameBufferLength started" << endl;
    return complexSymbolBufferLength(cyclic_prefix)*frameSymbols(cyclic_prefix);
}

uint complexSymbolBufferLength(int cyclic_prefix)
{
    if (PRINT)
        cout << "complexSymbolBufferLength - complexSymbolBufferLength started" << endl;
    return (SUBCARRIERS + ((int)SUBCARRIERS / cyclic_prefix));
}

```

```cpp
uint payloadLength(int cyclic_prefix, int phy_mode)
{
    if (PRINT)
        cout << "payloadLength - payloadLength started" << endl;
    uint8_t useful_symbols; // number of OFDM symbols carrying payload
    float coding_rate;  // uncoded to coded ratio

    switch (cyclic_prefix)
    {
        case 4:
            useful_symbols = 22;
            break;
        case 8:
            useful_symbols = 24;
            break;
        case 16:
            useful_symbols = 26;
            break;
        case 32:
            useful_symbols = 27;
            break;
        default:
            return 0;
    }


    switch (phy_mode)
    {
        case 1:
            coding_rate = 1;
            break;
        case 2:
            coding_rate = 2.0 / (2.0 * 3.0f);
            break;
        case 3:
            coding_rate = 2.0f / 2.0f;
            break;
        case 4:
            coding_rate = 2.0f * 2.0f / 3.0f;
            break;
        case 5:
            coding_rate = 2.0f * 3.0f / 4.0f;
            break;
        case 6:
            coding_rate = 2.0f * 5.0f / 6.0f;
            break;
        case 7:
            coding_rate = 4.0f / 2.0f;
            break;
        case 8:
            coding_rate = 4.0f * 2.0f / 3.0f;
            break;
        case 9:
            coding_rate = 4.0f * 3.0f / 4.0f;
            break;
        case 10:
            coding_rate = 4.0f * 5.0f / 6.0f;
            break;
        case 11:
            coding_rate = 6.0f / 2.0f;
            break;
        case 12:
            coding_rate = 6.0f * 2.0f / 3.0f;
            break;
        case 13:
            coding_rate = 6.0f * 3.0f / 4.0f;
            break;
        case 14:
            coding_rate = 6.0f * 5.0f / 6.0f;
            break;
        default:
            return 0;
    }

    return (uint)floor(DATACARRIERS * useful_symbols * coding_rate / 8)-1; // Without
the -1, frame generator produces excess symbol
```

```cpp
}

// callback function
int callbackWhatsReceived(unsigned char *_header,
               int _header_valid,
               unsigned char *_payload,
               unsigned int _payload_len,
               int _payload_valid,
               framesyncstats_s _stats,
               void *_userdata)
{
    cout << endl;
    cout << "***** callback invoked!" << endl << endl;
    if (_header_valid)
    {
        cout << "  header valid" << endl;
    }
    else
    {
        cout << "  header invalid" << endl;
    }

    if (_payload_valid)
    {
        cout << "  payload valid" << endl;
    }
    else
    {
        cout << "  payload invalid" << endl;
    }
    cout << endl;

    unsigned int i;
    if (_header_valid)
    {
        cout << "Received header: "<< endl;
        for (i = 0; i < 8; i++)
        {
            cout << _header[i];
        }
        cout << endl << endl;
    }

    if (_payload_valid)
    {
        cout << "Received payload: " << endl;
        for (i = 0; i < _payload_len; i++)
        {
            if (_payload[i] == 0)
                break;
            cout << _payload[i];
        }
        cout << endl << endl;
    }

    cout << "payload len: " << _payload_len << endl;
    cout << endl;

    return 0;
}


int startSDRTXStream(float *tx_buffer, int FrameSampleCnt)
{
    // Initialize stream
    lms_stream_t streamId;                          // stream structure
    streamId.channel = 0;                           // channel number
    streamId.fifoSize = 1024 * 1024;                // fifo size in samples
    streamId.throughputVsLatency = 1.0;             // optimize for max throughput
    streamId.isTx = true;                           // TX channel
    streamId.dataFmt = lms_stream_t::LMS_FMT_F32; // 12-bit integers
    if (LMS_SetupStream(device, &streamId) != 0)
        error();

    // Start streaming
    LMS_StartStream(&streamId);

    if (PRINT)
        cout << "startSDRTXStream - FrameSampleCnt: " << FrameSampleCnt << endl;

    for (int i = 0; i < 100; i++)
        int ret = LMS_SendStream(&streamId, tx_buffer, FrameSampleCnt, nullptr,
1000);


    // Stop streaming
    LMS_StopStream(&streamId);            // stream is stopped but can be started
again with LMS_StartStream()
    LMS_DestroyStream(device, &streamId); // stream is deallocated and can no longer
be used

    // Close device
    if (LMS_Close(device) == 0)
    {
        msgSDR.str("");
        msgSDR << "Closed" << endl;
        Logger(msgSDR.str());
    }

    sleep(1);

    return 0;
}

int SDRinit(double frequency, double sampleRate, int modeSelector, double
normalizedGain)
{
    // Find devices
    int n;
    lms_info_str_t list[8]; // should be large enough to hold all detected devices
    if ((n = LMS_GetDeviceList(list)) < 0)
    {
        error(); // NULL can be passed to only get number of devices
    }
    msgSDR.str("");
    msgSDR << "Number of devices found: " << n;
    Logger(msgSDR.str()); // print number of devices
    if (n < 1)
    {
        return -1;
    }

    // open the first device
    if (LMS_Open(&device, list[0], NULL))
    {
        error();
    }
    sleep(1);

    // Initialize device with default configuration
    if (LMS_Init(device) != 0)
    {
        error();
    }
    sleep(1);

    // Set SDR GPIO diretion GPIO0-5 to output and GPIO6-7 to input
    uint8_t gpio_dir = 0xFF;
    if (LMS_GPIODirWrite(device, &gpio_dir, 1) != 0)
    {
        error();
    }

    // Read and log GPIO direction settings
    uint8_t gpio_val = 0;
    if (LMS_GPIODirRead(device, &gpio_val, 1) != 0)
    {
        error();
    }
    msgSDR.str("");
    msgSDR << "Set GPIOs direction to output.\n";
    Logger(msgSDR.str());

    // Set GPIOs to RX mode (initial settings)
    if (LMS_GPIOWrite(device, &modeGPIO[modeSelector], 1) != 0)
    {
        error();
    }

    // Read and log GPIO values
    if (LMS_GPIORead(device, &gpio_val, 1) != 0)
    {
        error();
    }
    msgSDR.str("");
    msgSDR << "GPIO Output to High Level:\n";
    print_gpio(gpio_val);
    Logger(msgSDR.str());

    msgSDR.str("");
    msgSDR << "LimeRFE set to " << modeName[modeSelector] << endl;
    Logger(msgSDR.str());


    // Enable RX or TX channel,Channels are numbered starting at 0
    if (modeSelector == RX_MODE)
    {
        if (LMS_EnableChannel(device, LMS_CH_RX, 0, true) != 0)
        {
            error();
        }
        if (LMS_EnableChannel(device, LMS_CH_TX, 0, false) != 0)
        {
            error();
        }
    }
    else
    {
        if (LMS_EnableChannel(device, LMS_CH_TX, 0, true) != 0)
        {
            error();
        }
        if (LMS_EnableChannel(device, LMS_CH_RX, 0, false) != 0)
        {
            error();
        }
    }

    // Set sample rate
    if (LMS_SetSampleRate(device, (float)sampleRate, 0) != 0)
    {
        error();
    }
    msgSDR.str("");
    msgSDR << "Sample rate: " << sampleRate / 1e6 << " MHz" << endl;
    Logger(msgSDR.str());

    // Set center frequency
    if (modeSelector == RX_MODE)
    {
        if (LMS_SetLOFrequency(device, LMS_CH_RX, 0, frequency) != 0)
        {
            error();
        }
    }
    else
    {
        if (LMS_SetLOFrequency(device, LMS_CH_TX, 0, frequency) != 0)
        {
            error();
        }
    }

    msgSDR.str("");
    msgSDR << "Center frequency: " << frequency / 1e6 << " MHz" << endl;
    Logger(msgSDR.str());

    // select Low TX path for LimeSDR mini --> TX port 2 (misslabed in MINI, correct in USB)
    if (modeSelector == RX_MODE)
    {
        if (LMS_SetAntenna(device, LMS_CH_RX, 0, LMS_PATH_LNAL) != 0)
        {
            error();
        }
    }
    else
    {
        if (LMS_SetAntenna(device, LMS_CH_TX, 0, LMS_PATH_TX2) != 0)
        {
            error();
        }

        // set TX gain
        if (LMS_SetNormalizedGain(device, LMS_CH_TX, 0, normalizedGain) != 0)
        {
            error();
        }
    }


    // calibrate Tx, continue on failure
    if (modeSelector == RX_MODE)
    {
        LMS_Calibrate(device, LMS_CH_RX, 0, sampleRate, 0);
    }
    else
    {
        LMS_Calibrate(device, LMS_CH_TX, 0, sampleRate, 0);
    }

    sleep(2);

    return 0;
}

int error()
{
    msgSDR.str("");
    msgSDR << "ERROR: " << LMS_GetLastErrorMessage();
    Logger(msgSDR.str());
    if (device != NULL)
        LMS_Close(device);
    return -1;
}

```

```
/*****************************************************************************
 * C++ source of RPX-100-BER_simulator
 *
 * File:   RPX-100-TX.h
 * Author: Marek Honek
 *
 * Created on 19 Apr 2022, 18:20
 *****************************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sstream>
#include <syslog.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <cstdio>
#include <ctime>
#include <math.h>
#include <complex.h>
#include <time.h>
#include <chrono>
#include <cstring>
#include <bitset>
#include "stuff/ini.h"
#include "stuff/log.h"
#include <chrono>
#include "liquid/liquid.h"
#include "stuff/ServerSocket.h"
#include "stuff/SocketException.h"
#include <iterator>
#include <signal.h>
#include "stuff/Util.h"
#include "stuff/WebSocketServer.h"
#include <correct.h>
#pragma once

#define SUBCARRIERS 1024
#define DATACARRIERS 480

// print each step for debuggigng
#define PRINT false

int error();
string exec(string command);

//BER simulation
#define MIN_SNR 0
#define MAX_SNR 25
#define SIMULATION_REPETITIONS 1000

void sendFrame(int cyclic_prefix, int phy_mode);
void frameReception(int cyclic_prefix);
int global_cycl_pref_index;
int global_phy_mode;
liquid_float_complex before_channel_buffer[33280]; //buffer for artificial channel
liquid_float_complex after_channel_buffer[33280]; //buffer for artificial channel
int artificial_SNR;
float BER_log[4][15][MAX_SNR+1] = {0}; //[cycl pref index][phy_mode][SNR]
float calculateBER(unsigned int payload_len, string transmitted, unsigned char
*received);
bool callback_invoked = false;
int  exportBER(void);
string message = "";

// Initialize sdr buffers
liquid_float_complex complex_i(0,1);

void frameAssemble(liquid_float_complex *c_frame_buffer, int cyclic_prefix, int
phy_mode); //buffer changed to c_frame_buffer for BER simulation (original
r_frame_buffer)
void subcarrierAllocation (unsigned char *array);
ofdmflexframegen DefineFrameGenerator (int dfg_cycl_pref, int dfg_PHYmode);
int frameSymbols(int cyclic_prefix);
uint complexFrameBufferLength(int cyclic_prefix);
uint complexSymbolBufferLength(int cyclic_prefix);
uint payloadLength(int cyclic_prefix, int phy_mode);
void artificialChannel(int cyclic_prefix);
void printByteByByte(unsigned int payload_len, unsigned char *transmitted, unsigned
char *received);
string alterMessage(int payload_len);

int callbackWhatsReceived(unsigned char *_header,
               int _header_valid,
               unsigned char *_payload,
               unsigned int _payload_len,
               int _payload_valid,
               framesyncstats_s _stats,
               void *_userdata);

int callbackBERCalculation(unsigned char *_header,
               int _header_valid,
               unsigned char *_payload,
               unsigned int _payload_len,
               int _payload_valid,
               framesyncstats_s _stats,
               void *_userdata);

```

```cpp
/*****************************************************************************
 * C++ source of RPX-100-BER_simulator
 *
 * File:   RPX-100-BER_simulator.cpp
 * Author: Marek Honek
 *
 * Created on 19 Apr 2022, 18:20
 *****************************************************************************/

#include "RPX-100-BER_simulator.h"

using namespace std;

int main(void)
{
    cout << "main - program started" << endl;

    for (int repetition = 0; repetition < SIMULATION_REPETITIONS; repetition++) // repetition for higher resolution
    {
        message = alterMessage(8100); //max payload

        for (global_cycl_pref_index = 0; global_cycl_pref_index < 4; global_cycl_pref_index++) // cyclic prefix
        {
            for (global_phy_mode = 1; global_phy_mode <= 14; global_phy_mode++) // PHY mode
            {
                if (PRINT)
                    cout << "main - global_phy_mode: " << global_phy_mode << "; global_cycl_pref_index: " << global_cycl_pref_index << "; cycl_pref: " << (4 << global_cycl_pref_index) << endl;

                frameAssemble(before_channel_buffer, 4 << global_cycl_pref_index, global_phy_mode);
                if (PRINT)
                    cout <<"main - frameAssemble exitted" << endl;

                for (artificial_SNR = MIN_SNR; artificial_SNR<=MAX_SNR; artificial_SNR++) // SNR sweep
                {
                    artificialChannel(4 << global_cycl_pref_index);

                    if (PRINT)
                        cout << "main - artificialChannel exitted" << endl;

                    frameReception(4 << global_cycl_pref_index);
                    if (PRINT)
                        cout <<"main - frameReception exitted" << endl;
                }
            }
            cout << "main - repetition: " << repetition << "; cp: " << (4<<global_cycl_pref_index) << endl;
        }
    }

    // Up to now, BER_log contains sum of results from individual simulations. Following for loop structure divides
    // the value by number of simulations. Thus calculates the average of all simulations.
    for (global_cycl_pref_index = 0; global_cycl_pref_index<4; global_cycl_pref_index++) // cyclic prefix
    {
        for (global_phy_mode = 1; global_phy_mode <=14; global_phy_mode++) // PHY mode
        {
            cout << "main - cyclic prefix: " << (4<<global_cycl_pref_index) << "; PHY mode: " << global_phy_mode << endl;
            for (artificial_SNR = MIN_SNR; artificial_SNR<=MAX_SNR; artificial_SNR++) // SNR sweep
            {
                BER_log[global_cycl_pref_index][global_phy_mode][artificial_SNR]/=SIMULATION_REPETITIONS;
            }
        }
    }

    exportBER();

    return(0);
}


void frameAssemble(liquid_float_complex *c_frame_buffer, int cyclic_prefix, int phy_mode) //buffer changed to c_frame_buffer for BER simulation (original r_frame_buffer)
{
    if (PRINT)
        cout << "frameAssemble - frameAssemble started" << endl;

    liquid_float_complex complex_i(0, 1);

    unsigned int payload_len = payloadLength(cyclic_prefix, phy_mode); //depends on PHY mode and cyclic prefix
    if (PRINT)
    {
        cout << "frameAssemble - payloadLength exitted" << endl;
        cout << "frameAssemble - payload_len: " << payload_len << endl;
    }

    uint c_buffer_len = complexSymbolBufferLength(cyclic_prefix); //depends on cyclic prefix
    if (PRINT)
        cout << "frameAssemble - complexSymbolBufferLength exitted" << endl;

    // create frame generator
    ofdmflexframegen fg = DefineFrameGenerator(cyclic_prefix, phy_mode);
    if (PRINT)
    {
        cout << "frameAssemble - DefineFrameGenerator exitted, frame generator setted" << endl;
        ofdmflexframegen_print(fg);
    }

    // buffers
    liquid_float_complex c_buffer[c_buffer_len]; // time-domain buffer
    unsigned char header[8];                     // header data
    unsigned char payload[payload_len] = {};          // payload data
    if (PRINT)
        cout << "frameAssemble - header and payload buffers initialized" << endl;

    // ... initialize header/payload ...
    strcpy((char *)payload, message.c_str());

    header[0] = '0';
    header[1] = '0';
    header[2] = '0';
    header[3] = '0';
    header[4] = '0';
    header[5] = '0';
    header[6] = '0';
    header[7] = '0';

    if (PRINT)
        cout << "frameAssemble - header and payload written" << endl;

    // cout << "frameAssemble - payload: " << payload << endl;  // prints as text
    if (PRINT)
    {
        cout << "frameAssemble - payload: ";                       // prints as numbers
        for (int i=0; i<payload_len; i++)
        {
            cout << unsigned(payload[i]) << " ";
        }
        cout << endl << endl;
    }


    // assemble frame
    ofdmflexframegen_assemble(fg, header, payload, payload_len);
    if (PRINT)
    {
        cout << "frameAssemble - frame assembled" << endl;
        ofdmflexframegen_print(fg);
    }

    int last_symbol = 0;
    int i = 0;
    int l = 0;

    while (!last_symbol)
    {
        // pthread_mutex_lock(&SDRmutex);

        // generate each OFDM symbol
        last_symbol = ofdmflexframegen_write(fg, c_buffer, c_buffer_len);
        if (PRINT)
        {
            cout << "frameAssemble - symbol " << l+1 << " written" << endl;
            cout << "frameAssemble - c_buffer[0]: " << c_buffer[0] << endl;
            cout << "frameAssemble - last_symbol value: " << last_symbol << endl;
        }


        if (!last_symbol)
        {
            if (PRINT)
                cout << "frameAssemble - starting complex to real buffer conversion" << endl;
            for (i = 0; i < c_buffer_len; i++)
            {
                c_frame_buffer[l*c_buffer_len+i]=c_buffer[i];
            }
        }
        if (PRINT)
        {
            cout << "frameAssemble - c_frame_buffer[0]: " << c_frame_buffer[0] << endl;
            cout << "frameAssemble - exiting complex to real buffer conversion" << endl;
        }

        l++;
    }
    ofdmflexframegen_destroy(fg);
}


void frameReception(int cyclic_prefix)
{
    if (PRINT)
        cout << "frameReception - frameReception started" << endl;

    int c_sync_buffer_len = complexFrameBufferLength(cyclic_prefix); //synchronizer buffer can be  of arbitrary length
    if (PRINT)
        cout << "frameReception - buffers initialized" << endl;

    unsigned char allocation_array[SUBCARRIERS];    // subcarrier allocation array (null/pilot/data)
    subcarrierAllocation(allocation_array);
    if (PRINT)
        cout << "frameReception - subcarrierAllocation exited; allocation_array defined" << endl;

    unsigned int cp_len = (int)SUBCARRIERS / cyclic_prefix; // cyclic prefix length
    unsigned int taper_len = (int)cp_len / 4;          // taper length

    //ofdmflexframesync fs = ofdmflexframesync_create(SUBCARRIERS, cp_len, taper_len, allocation_array, callbackWhatsReceived, NULL);
    ofdmflexframesync fs = ofdmflexframesync_create(SUBCARRIERS, cp_len, taper_len, allocation_array, callbackBERCalculation, NULL);
    if (PRINT)
        cout << "frameReception - frame synchronizer created" << endl;

    // while(1)
    {
        // receive symbol (read samples from buffer)
        ofdmflexframesync_execute(fs, after_channel_buffer, c_sync_buffer_len);
        if (PRINT)
            cout << "frameReception - synchronization ended" << endl;

        if (!callback_invoked)
        {
            if (PRINT)
                cout << "frameReception - callback was not invoked" << endl;
            BER_log[global_cycl_pref_index][global_phy_mode][artificial_SNR] += 1;
        }

        callback_invoked = false;

        ofdmflexframesync_destroy(fs);
        if (PRINT)
            cout << "frameReception - ofdmflexframesync destroyed";
    }

}

ofdmflexframegen DefineFrameGenerator (int dfg_cycl_pref, int dfg_PHYmode)
{
    if (PRINT)
        cout << "DefineFrameGenerator - DefineFrameGenerator started" << endl;

    // initialize frame generator properties
    ofdmflexframegenprops_s fgprops;
    ofdmflexframegenprops_init_default(&fgprops);
    fgprops.check = LIQUID_CRC_NONE;
    fgprops.fec1 = LIQUID_FEC_NONE;

    unsigned int cp_len = (int)SUBCARRIERS / dfg_cycl_pref; // cyclic prefix length
    unsigned int taper_len = (int)cp_len / 4;          // taper length

    switch (dfg_PHYmode)
    {

    case 1:
        fgprops.fec0 = LIQUID_FEC_NONE;
        fgprops.mod_scheme = LIQUID_MODEM_PSK2;
        break;
    case 2:
        fgprops.fec1 = LIQUID_FEC_REP3;
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 3:
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 4:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P23;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 5:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P34;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
         break;
    case 6:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P56;
        fgprops.mod_scheme = LIQUID_MODEM_QPSK;
        break;
    case 7:
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 8:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P23;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 9:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P34;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 10:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P56;
        fgprops.mod_scheme = LIQUID_MODEM_QAM16;
        break;
    case 11:
        fgprops.fec0 = LIQUID_FEC_CONV_V27;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    case 12:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P23;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    case 13:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P34;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    case 14:
        fgprops.fec0 = LIQUID_FEC_CONV_V27P56;
        fgprops.mod_scheme = LIQUID_MODEM_QAM64;
        break;
    default:
        return NULL;
    }

    unsigned char allocation_array[SUBCARRIERS]; // subcarrier allocation array(null/pilot/data)
    subcarrierAllocation(allocation_array);
    if (PRINT)
        cout << "DefineFrameGenerator - subarrierAllocation exitted" << endl;

    return ofdmflexframegen_create(SUBCARRIERS, cp_len, taper_len, allocation_array, &fgprops);
}

void subcarrierAllocation (unsigned char *array)
{
    if (PRINT)
        cout << "subcarrierAllocation - subcarrierAllocation started" << endl;
    for (int i = 0; i < 1024; i++)
    {
        if (i < 232)
            array[i] = 0; // guard band

        if (231 < i && i < 792)
            if (i % 7 == 0)
                array[i] = 1; // every 7th carrier pilot
            else
                array[i] = 2; // rest data

        if (i > 791)
            array[i] = 0; // guard band
    }
}

int frameSymbols(int cyclic_prefix)
{
    if (PRINT)
        cout << "frameSymbols - frameSymbols started" << endl;
    int symbolCnt;
    switch (cyclic_prefix)
    {
    case 4:
        symbolCnt = 22+4;
        break;
    case 8:
        symbolCnt = 24+4;
        break;
    case 16:
        symbolCnt = 26+4;
        break;
    case 32:
        symbolCnt = 27+4;
        break;
    default:
        symbolCnt = 0;
    }

```

```cpp
    if (PRINT)
        cout << "frameSymbols - symbolCnt: " << symbolCnt << endl;
    return symbolCnt;
}

uint complexFrameBufferLength(int cyclic_prefix)
{
    if (PRINT)
        cout << "complexFrameBufferLength - complexFrameBufferLength started" <<
endl;
    return complexSymbolBufferLength(cyclic_prefix)*frameSymbols(cyclic_prefix);
}

uint complexSymbolBufferLength(int cyclic_prefix)
{
    if (PRINT)
        cout << "complexSymbolBufferLength - complexSymbolBufferLength started" <<
endl;
    return (SUBCARRIERS + ((int)SUBCARRIERS / cyclic_prefix));
}

uint payloadLength(int cyclic_prefix, int phy_mode)
{
    if (PRINT)
        cout << "payloadLength - payloadLength started" << endl;
    uint8_t useful_symbols;
    float coding_rate;

    switch (cyclic_prefix)
    {
        case 4:
            useful_symbols = 22;
            break;
        case 8:
            useful_symbols = 24;
            break;
        case 16:
            useful_symbols = 26;
            break;
        case 32:
            useful_symbols = 27;
            break;
        default:
            return 0;
    }


    switch (phy_mode)
    {
        case 1:
            coding_rate = 1;
            break;
        case 2:
            coding_rate = 2.0 / (2.0 * 3.0f);
            break;
        case 3:
            coding_rate = 2.0f / 2.0f;
            break;
        case 4:
            coding_rate = 2.0f * 2.0f / 3.0f;
            break;
        case 5:
            coding_rate = 2.0f * 3.0f / 4.0f;
            break;
        case 6:
            coding_rate = 2.0f * 5.0f / 6.0f;
            break;
        case 7:
            coding_rate = 4.0f / 2.0f;
            break;
        case 8:
            coding_rate = 4.0f * 2.0f / 3.0f;
            break;
        case 9:
            coding_rate = 4.0f * 3.0f / 4.0f;
            break;
        case 10:
            coding_rate = 4.0f * 5.0f / 6.0f;
            break;
        case 11:
            coding_rate = 6.0f / 2.0f;
            break;
        case 12:
            coding_rate = 6.0f * 2.0f / 3.0f;
            break;
        case 13:
            coding_rate = 6.0f * 3.0f / 4.0f;
            break;
        case 14:
            coding_rate = 6.0f * 5.0f / 6.0f;
            break;
        default:
            return 0;
    }

    return (uint)floor(DATACARRIERS * useful_symbols * coding_rate / 8)-1;
}

// callback function
int callbackWhatsReceived(unsigned char *_header,
               int _header_valid,
               unsigned char *_payload,
               unsigned int _payload_len,
               int _payload_valid,
               framesyncstats_s _stats,
               void *_userdata)
{
    cout << endl;
    cout << "***** callback invoked!" << endl << endl;
    if (_header_valid)
    {
        cout << "  header valid" << endl;
    }
    else
    {
        cout << "  header invalid" << endl;
    }

    if (_payload_valid)
    {
        cout << "  payload valid" << endl;
    }
    else
    {
        cout << "  payload invalid" << endl;
    }
    cout << endl;

    unsigned int i;
    if (_header_valid)
    {
        cout << "Received header: "<< endl;
        for (i = 0; i < 8; i++)
        {
            cout << _header[i];
        }
        cout << endl << endl;
    }

    if (_payload_valid)
    {
        cout << "Received payload: " << endl;
        for (i = 0; i < _payload_len; i++)
        {
            if (_payload[i] == 0)
                break;
            cout << _payload[i];
        }
        cout << endl << endl;
    }

    cout << "payload len: " << _payload_len << endl;
    cout << endl;

    return 0;
}

int callbackBERCalculation(unsigned char *_header,
               int _header_valid,
               unsigned char *_payload,
               unsigned int _payload_len,
               int _payload_valid,
               framesyncstats_s _stats,
               void *_userdata)
{
    if (PRINT)
        cout << "callbackBERCalculation invoked - _payload_len: "<< _payload_len << 
endl;

    callback_invoked = true;

    if (_payload_len != 0)
    {
        float BER = calculateBER(_payload_len, message, _payload);
        if (PRINT)
            cout << "callbackBERCalculation - calculateBER exitted; BER: "<< BER << 
endl;
        if (PRINT)
        {
            cout << "calculateBER exitted" << endl;
            cout << "BER: " << BER << endl;
        }
        BER_log[global_cycl_pref_index][global_phy_mode][artificial_SNR] += BER;
    }
    else
        BER_log[global_cycl_pref_index][global_phy_mode][artificial_SNR] += 1;

    return 0;
}

void artificialChannel(int cyclic_prefix)
{
    if (PRINT)
        cout << "artificialChannel - artificialChannel started" << endl;

    int channel_buffer_len = complexFrameBufferLength(cyclic_prefix); //synchronizer 
buffer can be  of arbitrary length
    if (PRINT)
        cout << "artificialChannel - complexFrameBufferLength exitted" << endl;

    // create channel object
    channel_cccf channel = channel_cccf_create();

    // additive white Gauss noise impairment
    float noise_floor   = -60.0f;   // noise floor [dB]
    float SNRdB         =  (float)artificial_SNR;   // signal-to-noise ratio [dB]
    channel_cccf_add_awgn(channel, noise_floor, SNRdB);

    // carrier offset impairments
    float dphi          =   0.00f;  // carrier freq offset [radians/sample]
    float phi           =   0.0f;   // carrier phase offset [radians]
    //channel_cccf_add_carrier_offset(channel, dphi, phi);

    // multipath channel impairments
    liquid_float_complex* hc   = NULL;     // defaults to random coefficients
    unsigned int hc_len = 4;        // number of channel coefficients
    //channel_cccf_add_multipath(channel, hc, hc_len);

    // time-varying shadowing impairments (slow flat fading)
    float sigma         = 1.0f;     // standard deviation for log-normal shadowing
    float fd            = 0.1f;     // relative Doppler frequency
    //channel_cccf_add_shadowing(channel, sigma, fd);

    // print channel internals
    if (PRINT)
        channel_cccf_print(channel);

    // fill buffer and repeat as necessary
    // apply channel to input signal
    channel_cccf_execute_block(channel, before_channel_buffer, channel_buffer_len, after_channel_buffer);

    if (PRINT)
        cout << "artificialChannel - channel executed" << endl;

    // destroy channel
    channel_cccf_destroy(channel);
    if (PRINT)
        cout << "artificialChannel - channel destroyed";
}

float calculateBER(unsigned int payload_len, string transmitted, unsigned char *received)
{
    unsigned int temp_payload_len = payload_len; //    strcpy((char *)u_ch_transmitted, transmitted.c_str()); makes diffilulties
    if (PRINT)
        cout << "calculateBER - calculateBER started; payload_len: "<< payload_len << endl;

    float BER = 0;

    if (PRINT)
        cout << "calculateBER - 1st - payload_len: "<< payload_len<<"; temp_payload_len: "<< temp_payload_len << endl;

    unsigned char u_ch_transmitted[8100] = {};
    if (PRINT)
    {
        cout << "calculateBER - u_ch_transmitted initialized" << endl;
        cout << "calculateBER - 2nd - payload_len: "<< payload_len <<"; temp_payload_len: "<< temp_payload_len << endl;
    }

    strcpy((char *)u_ch_transmitted, transmitted.c_str());

    if (PRINT)
    {
        cout << "calculateBER - message copied" << endl;
        cout << "calculateBER - 3rd - payload_len: "<< payload_len<<"; temp_payload_len: "<< temp_payload_len << endl;
    }

    payload_len = temp_payload_len;

    if (PRINT)
    {
        cout << "calculateBER - 4th - payload_len: "<< payload_len <<"; temp_payload_len: "<< temp_payload_len<< endl;
        printByteByByte(payload_len, u_ch_transmitted, received);
        cout << "calculateBER - printByteByByte exitted" << endl;
    }

    for (int i=0; i<payload_len; i++)
    {
        for (int l=0; l<8; l++)
        {
            BER += ((u_ch_transmitted[i]^received[i])>>l)&1;
        }
    }
    BER /= payload_len*8;

    return BER;
}

void printByteByByte(unsigned int payload_len, unsigned char *transmitted, unsigned char *received)
{
    if (PRINT)
        cout << "printByteByByte - printByteByByte(payload_len = "<<payload_len<<") started" << endl;

    cout << "printByteByByte - transmitted: ";
    for (int i=0; i<payload_len; i++)
```

```cpp
    {
        cout << unsigned(transmitted[i]) << " ";
    }
    cout << endl << endl;


    cout << "printByteByByte - received: ";
    for (int i=0; i<payload_len; i++)
    {
        cout << unsigned(received[i]) << " ";
    }
    cout << endl << endl;
}

string alterMessage(int payload_len)
{
    string s = "";
    char random;
    for (int i = 0; i < payload_len; i++)
    {
        random = rand()%255+1;
        s = s + random;
    }
    return s;
}

int exportBER()
{
    cout << "exportBER - export started" << endl;
    std::ofstream myfile;
    myfile.open ("./BER_simulation_outcome/BER_simulation_" + getCurrentDateTime("now") + ".csv");
    myfile << "Number of simulations: " << SIMULATION_REPETITIONS << endl;
    for (artificial_SNR = MIN_SNR-1; artificial_SNR<=MAX_SNR; artificial_SNR++) // SNR sweep
    {
        if (artificial_SNR == MIN_SNR-1)
            myfile << "SNR";
        else
            myfile << artificial_SNR;
        for (global_phy_mode = 1; global_phy_mode <=14; global_phy_mode++) // PHY mode
        {
            for (global_cycl_pref_index = 0; global_cycl_pref_index<4; global_cycl_pref_index++) // cyclic prefix
            {
                if (artificial_SNR == MIN_SNR-1)
                    myfile << ", PHY mode " << global_phy_mode << "; CP " << (4 << global_cycl_pref_index);
                else
                    myfile << "," << BER_log[global_cycl_pref_index][global_phy_mode][artificial_SNR];
            }
        }
        myfile << endl;
    }
    myfile.close();
    return 0;
}

```