Pedro Miguel Rodrigues Cunha

Licenciado em Ciências da Engenharia Electrotécnica e de Computadores

Implementing the SC-FDMA Transmission Technique Using the GNURadio Platform

Dissertação apresentada para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores, pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Orientadores : Prof. Doutor Luis Bernardo, FCT-UNL
Prof. Doutor Rui Dinis, FCT-UNL

Júri:
Presidente: Prof. Doutor Rodolfo Oliveira
Arguentes: Prof. Doutor João Oliveira
Vogais: Prof. Doutor Rui Dinis

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2014

Implementing the SC-FDMA Transmission Technique Using the GNURadio Platform

To all that are a part of my life.

Acknowledgements

First, I would like to express my gratitude to my supervisor Luis Bernardo for all the guidance, support and patience that he gave me along the realisation of this thesis; to my co-supervisor Rui Dinis for the scientific knowledge that he gave me to complete this project. Without their support, this work could not be developed. Also, to other teachers of the Telecommunications Section for their sympathy and academic support. Last, thank you to FCT/MEC Femtocells (PTDC/EEA- TEL/120666/2010), MANY2COMWIN (EXPL/EEI-TEL/0969/2013) and ADIN (PTDC/EEI-TEL/2990/2012) projects for the financial support.

From the Department of Electrical Engineering, I would also like to give my special thanks to my colleagues and friends Pedro Sardinha, António Furtado, João Silva, Nuno Pereira, Filipe Martins, Carlos Ribeiro, Fernando Rosado, Diogo Rocha and Bruno Ribeiro for their friendship and fellowship during my academic years.

To my closest friends Rui Cabrita, Pedro Anjos, Filipe Alves, Filipe Oliveira, Gonçalo Mendonça and Catarina Branco for their friendship and times together through the years.

A special thanks, because I am very grateful for all the support, love and encouragement my girlfriend Marta gave me during our time together in this last years, I appreciated every moment. Thank you Marta.

I would like to express my gratitude to my grandmother Gertrudes for her love and education she gave me in my first years, and to the rest of my family that support me in my tough times and saw me grow up, specially my uncle José Paulo. Finally, from the deep of my heart I would like to thank my parents for their love, education and support they gave me during all my life.

To all, thank you very much.

Resumo

Com a evolução na área das telecomunicações, foram implementadas várias técnicas de transmissão de dados. Na nova geração de comunicações móveis, Long Term Evolution (LTE), o Orthogonal Frequency Division Multiplexing (OFDM) é usado para as transmissões de dados no downlink e o Single Carrier - Frequency Division Multiple Access (SC-FDMA) é usado para as transmissões de dados no uplink, devido a permitir uma maior eficiência energética na transmissão, com um menor rácio entre a potência de pico e a potência média transmitida. Esta tese foca-se nestas duas técnicas de transmissão e implementa um protótipo para o SC-FDMA utilizando a plataforma GNURadio.

O GNURadio é baseado no conceito Software-Defined Radio (SDR) e usa os equipamentos USRP para fazer a transmissão de sinais. Numa primeira análise, examinou-se o modulador OFDM que já estava implementado na plataforma GNURadio. Para criar o SC-FDMA, modificou-se os blocos modulador e desmodelador do OFDM e implementou-se módulos que trabalhassem com o novo sistema de preâmbulos (Zadoff-Chu), que realizassem novos algoritmos FFT e novas sincronizações.

Com o GNURadio-Companion GRC, testou-se ambas as modulações usando vários tipos de ambientes, alterando a potência de ruído e o desvio da frequência. Por último, realizaram-se experiências com os USRP em diferentes frequências, usando o cabo de loop-back e as antenas.

O protótipo proposto foi implemento com sucesso. Comparando o modelador SC-FDMA com o OFDM, os resultados dos testes mostram que a nova técnicas de transmissão foi mais eficiente em altas frequências.

Palavras Chave: USRP, GNU Radio, OFDM, SC-FDMA.

Abstract

With the evolution in the telecommunication field, several transmission techniques have been implemented. With the new generation of mobile communications, Long Term Evolution (LTE), the Orthogonal Frequency Division Multiplexing (OFDM) is used for the downlink data transmission and the Single Carrier - Frequency Division Multiple Access SC-FDMA) for the uplink data transmission and due to its more efficient energy efficiency, due to the low peak-to-average power ratio. This thesis focuses on these two transmission techniques and implements a SC-FDMA prototype in the GNURadio platform.

GNURadio is based on the Software-Defined Radio (SDR) concept and uses the USRP equipment to do the signal transmission. In a first analysis, we examine the OFDM modulator already implemented in the GNURadio platform. In order to create the SC-FDMA, we have modified the OFDM modulator and demodulator blocks and implemented modules that work with the new preamble system (Zadoff-Chu), to perform new FFT algorithms and new synchronizations.

Using the GNURadio-Companion GRC software, we tested both modulations using several types of environment, while changing the noise power and the frequency offset. Last, we performed experiments using the USRP devices in different frequencies, using the loop-back cable and the antennas.

The proposed prototype was successfully implemented. The tests comparing the SC-FDMA modulator with the SC-FDMA, show that the new transmission technique performed better at higher frequencies.

Keywords: USRP, GNURadio, OFDM, SC-FDMA.

Acronyms

ADC *Analog to Digital Converter*

DAC *Digital to Analog Converter*

FDM *Frequency Division Multiplexing*

FDE *Frequency Domain Equalization*

FFT *Fast Fourier Transform*

FPGA *Field-Programmable Gate Array*

GRC *GNURadio Companion*

IFDMA *Interleaved FDMA*

ICI *Inter-Carrier Interference*

IBI *Inter-Block Interference*

IFFT *Inverse Fast Fourier Transform*

ISI *Inter-Symbol Interference*

LTE *Long Term Evolution*

LFDMA *Localized SC-FDMA*

MC *Multi-Carrier*

MIMO *Multiple-Input and Multiple-Output*

ML *Maximum Likelihood*

MMSE *Minimum Mean Square Error*

OFDM *Orthogonal Frequency Division Multiplexing*

PAPR *Peak-to-Average Power Ratio*

PER *Packet Error Rate*

PSK *Phase Shift Keying*

PN *Pseudorandom Noise*

QAM *Quadrature Amplitude Modulation*

SC *Single Carrier*

SC-FDMA *Single Carrier - Frequency Division Multiple Access*

SDR *Software Defined Radio*

SNR *Signal to Noise Ratio*

USRP *Universal Software Radio Peripheral*

ZF *Zero Forcing*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

For decades, the communications systems have been growing exponentially and over time, filling the radio spectrum where they are implemented. On top of everything, the public demanded for faster and more reliable communications systems. To serve this demand, in 1980s the telecommunication community revisited a method that was conceived by Robert W. Chang in 1966, the Orthogonal Frequency Division Multiplexing (OFDM). Before this method was implemented, the total signal bandwidth was split into N non-overlapping frequency sub-channels, and each sub-channel was modulated with an independent symbol and then the N sub-channels were modulated in the frequency-domain. With the application of Frequency Division Multiplexing (FDM) and subsequently, the OFDM transmission technique, the carriers begun to overlap with each other, but they did not create Inter Carrier Interference (ICI). To obtain this performance the carriers have to be mathematically orthogonal between each other [NL08].

OFDM was the predominant method used until 1993, when H. Sari and his team presented in a conference paper [HSJ94], the advantages and drawbacks of OFDM modulation technique; they also introduced a different transmission technique called Single Carrier - Frequency Division Multiple Access (SC-FDMA). The authors proposed that SC-FDMA could attain the performance of the OFDM transmission while easing the Peak-to-Average Power Ratio (PAPR) and synchronization problems. SC-FDMA merge the characteristics of a Single Carrier (SC) transmission and the multiple access similar to the OFDM trans-

mission technique. SC-FDMA tries to take advantage of the strengths of both techniques [CS10].

The community has been fighting over the years between the two transmission techniques, until a major development put the SC-FDMA transmission technique in the spotlight. This happened when the Third-Generation Partnership Project (3GPP) started its work to define the technical standards for the so-called Beyond 3G systems. At the end of 2008, the release 8 of the 3GPP standard adopted OFDM for the downlink and SC-FDMA for the uplink [NL08]. In the release 11, the 3GPP standard adopted clustered SC-FDMA as the LTE-Advanced uplink access scheme, otherwise known as discrete Fourier transform spread OFDM (DFT-S-OFDM) [Tec11].

Parallel to the development of the new transmission techniques, the Software Defined Radio (SDR) was proposed in 1992. This system uses software instead of hardware to do the signal process and is used mainly for applications that run in multiple bands, reducing the number of hardware components and its weight [Mit95]. One software based on this concept is GNURadio. This environment does an excellent job creating new blocks necessary for the signal processing and testing them in several environments, using the software GNURadio-Companion (GRC). GNURadio already comes with several blocks implemented, including the OFDM modulator and demodulator blocks.

1.2 Objectives and Major Contributions

As previously mention, GNURadio software comes with several signal processing blocks implemented and one of them is the OFDM transmission technique. The main objective of this project is to implementation the SC-FDMA transmission technique in the same platform. The main objectives of this thesis are:

- Introduce the GNURadio software, presenting some of its features;
- Present the theory behind the transmission blocks, that are referred in this project;
- Explain the OFDM modulator and demodulator blocks implemented in GNURadio;
- Describe the implementation done to create the SC-FDMA blocks;

- Test and compare the SC-FDMA and the OFDM modulator and demodulator blocks, and see how they work and the differences between them.

The blocks developed in this project are implemented in GNURadio software, which is open source. This way, the SC-FDMA transmission technique developed can be shared, allowing other users to use or modify it.

1.3 Dissertation Structure

This dissertation is divided in five chapters and four appendices. Chapter 2 explains the SDR concept and the software associated with it, GNURadio. It also, explains which block types can be implemented and which tools exist in GNURadio. After, this chapter describes the theory behind the transmission techniques handled in this thesis.

Chapter 3 provides three examples of the tools from GNURadio. First, the Dial Tone example illustrates the use of the GRC tool; second, an example on how to create new blocks in GNURadio is provided; and third, an example shows the use of the filter design tool. After the examples, this chapter explains the OFDM modulator and demodulator blocks, already implemented in GNURadio. Finally, the chapter explains the main contribution of this dissertation: the implementation of the SC-FDMA blocks.

In chapter 4, the OFDM and SC-FDMA modulator and demodulator blocks are tested. First, we test the transmission blocks and see the outputs of some modules within them, in a noise and frequency distortion free environment. Second, using different noise and frequency distortions, we see how the transmission techniques hold up in the tests and measure the errors that occur. Lastly, we test the blocks using a real channel and see the differences between them.

Chapter 5 summarizes the main analysis done in this project, shows the main contributions of this work and refers to future work.

Appendix A explains the Dial Tone example using the GRC software. Appendix B describes the modifications performed in the new module created in chapter 3.2.1, which computes the square of the inputted signal. Appendix C and appendix D complement the explanation performed in chapters 3.3 and 3.4, showing the main modules that build the OFDM and SC-FDMA modulator and demodulator blocks, respectively.

Chapter 2

Theoretical Concepts

In the telecommunications world there are several modulation types to do data transmission. OFDM is used for that purpose in several systems such as the Digital Video Broadcasting [ser09], Digital Audio Broadcast (DAB), Asymmetric Digital Subscriber Line (ADSL)[Rum08], wireless broadband access technologies IEEE 802.16a/d/e [Bib04][IEE06] and the new generation of the cellular system in the Long Term Evolution has a downlink modulator [3GP06]. Another relevant modulation scheme is SC-FDMA adopted in the Long Term Evolution for the uplink access [LAMRdTM08].

This chapter introduces the theoretical bases necessary to develop the SC-FDMA module on the GNURadio. The first section 2.1 introduces SDR main concepts as well as the hardware used by the system, the software GNURadio, the definition of the block types and the tools used in this environment. The second part 2.2 briefly introduces the Multi Carrier (MC) and Single Carrier (SC) modulations and their properties. The transmission and receiver blocks also are characterized in both modulation types. Subsection 2.2.2 describes the OFDM scheme and 2.2.3 the SC-FDMA scheme.

2.1 Software Defined Radio

In 1992, Joe Mitola introduced the Software Defined Radio (SDR) concept to the world. This new concept uses software instead of hardware to process signals which are sent through antennas, in various frequencies. SDR is specially needed when there are applications that use multiple bands, reducing the number of hardware components and

its weight [Mit95].

This section introduces the main elements of SDR, including USRP, daughterboards (hardware components), GNURadio, block types and some software tools.

2.1.1 SDR principle and analog radios

Before SDR appears, traditional radio components (filters, modulators, mixers, amplifiers, FFT, etc) had their own fixed function and needed to be implemented exclusively via hardware. With SDR that principle was changed, almost all blocks are now implemented through programming and processed by a computer, simplifying the creation of new radio prototypes, as well as testing them and/or changing their configuration.

The signals must be analog to be transmitted through the air but SDR can only process them in the digital domain. To achieve this, SDR converts the analog signals to digital and vice-versa. The figure 2.1 shows that SDR uses an ADC to convert the antenna signals to the digital domain and processes them using software. When SDR needs to send any signals, they are converted through a DAC to the analog domain and are transmitted in the desired radio frequency.

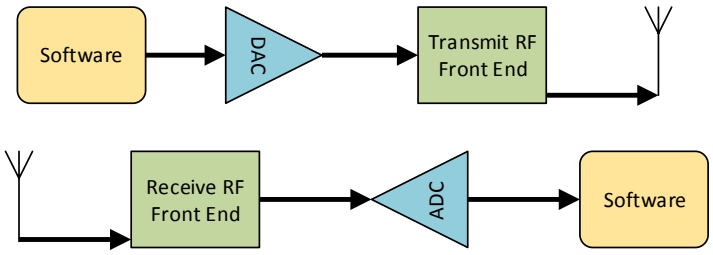


Figure 2.1: SDR sender and receiver module diagram [Mar09].

2.1.2 Universal Software Radio Peripheral (USRP)

USRP is a hardware designed to give general computers the ability to operate with high bandwidth software radios, basically giving them the digital baseband and intermediate frequency (IF) section of a radio communication systems. USRP does all of the waveform-specific processing. The FPGA is an important part on the USRP, because it is in this board where all the DACs and ADCs are connected. The FPGA also does the high

bandwidth math, reducing the data rate to rates compatible with USB 2.0 [Ham08].

USRP is a flexible low-cost platform for SDR developed by Matt Ettus and his company, Ettus Research. They provide a vast variety of USRP models such as the Bus Series, the Networked Series and the Embedded Series. The Bus Series (USRP1 and USRP B100) provides a low-cost RF processing capability and designed for cost-sensitive applications requiring exceptional bandwidth processing capability and dynamic range. This model supports streams up to 8 MS/s and users may implement custom functions in the FPGA fabric. The Networked series (USRP N200 and USRP N210) provides high-bandwidth, high-dynamic range processing capability. It uses a Gigabit Ethernet interface and because of that, the devices can transfer up to 50 MS/s of complex baseband samples. The Networked model uses a dual 14-bit, 100 MS/s ADC and dual 16-bit, 400 MS/s DAC. Also, it is provided a MIMO expansion port which can be used to synchronize two devices, making it the recommended solution for MIMO systems. Finally, the Embedded series (USRP E100 and USRP E110) combine the same functionality of the other USRP devices with an OMAP 3 embedded processor. This devices do not need to be connected to an external PC to operate. The Embedded Series is designed for applications that require stand-alone operation. The tests presented in this thesis were measured using a Bus Series, specifically the USRP B100 [Res13].

2.1.3 Daughterboards

The USRP alone cannot send and receive signals through the antenna. It needs to attach Daughterboards that serve as RF front ends. The daughterboards can belong to one of on three classes: Receivers, Transmitters and Transceivers. The SBX was the chosen to do the tests from a great diversity of daughterboards (BasicRX, BasicTX, SBX, WBX, etc.). This board has a wide bandwidth transceiver, that provides up to 100 mW of output power, and a typical noise figure of 5 dB. Due to the local oscillators for the receive and transmit chains, that operate independently, the board allows dual-band operation. The SBX is MIMO capable and can achieve a 40 MHz of bandwidth in frequencies between 400 MHz and 4,4 GHz. It is widely used in areas as WiFi, WiMax, S-band transceivers and 2.4 GHz ISM band transceivers [Res13].

2.1.4 GNURadio

There are several free software that can be found for SDR, some focused in using only the software to do the signals computation and USRP antennas to make the signals transmission (GNURadio, SDR4all, etc.). Other software, like HPSDR, may use the hardware component to do some of the computation [Mar09].

GNURadio was launched in 2001 by John Gillmore and Eric Blossom and is distributed using the GNU General Public License. It is today one of the most advanced open source project in the SDR area and an easy one to begin with. GNURadio target users are the hobbyists, the academics and the researchers. Nowadays, GNURadio provides examples, reference systems and applications for Global System for Mobile communications (GSM), OFDM, High-definition television (HDTV) and other areas [Mar09].

GNURadio is divided in blocks that do the signal processing, they are written in C++, it also uses flowgraphs that interconnects the blocks and configures them according to our requirements, they are written in Python. GNURadio blocks work with a infinite flow of data of a certain type like complex, short and float [Mar09].

The interface compiler, that allows the integration between C++ and Python language, is called Simplified Wrapper and Interface Generator (SWIG). Figure 2.2 shows the structure of a SDR application including the GNURadio and USRP [AM09].

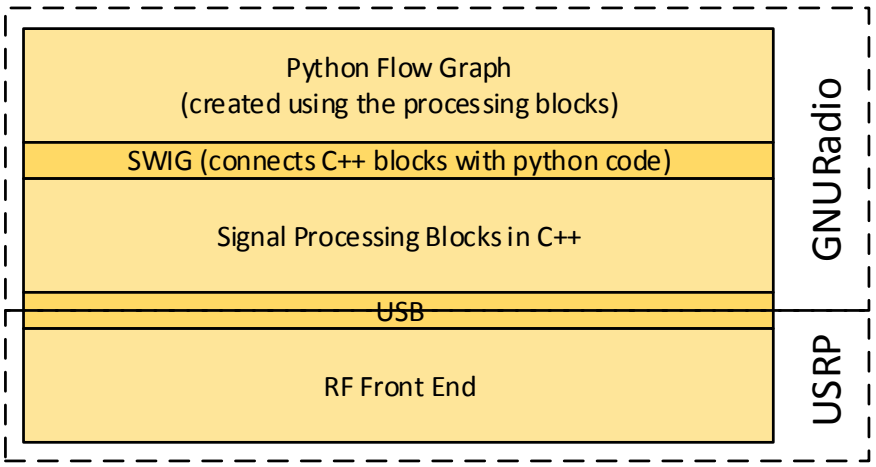


Figure 2.2: GNURadio block connections

GNURadio is updated periodically and the version used in this thesis is 3.6.4.1. The current stable version available is 3.7.0.1.

2.1.5 GNURadio Block Types

GNURadio offers four different types of blocks, implemented using C++: general, synchronous, decimation and interpolation blocks. Using Python, the programmer can create flowgraphs, where the earlier blocks are combined to create a hierarchical block.

A general block has a N:M ratio, which defines the relation between the number of input items and the number of output items; the other blocks are just specializations of this type of block. The main method of a general block is *general_work()*, however, for the other types is *work()*. One of the input parameters is *ninput_items*, which is a vector describing the length of each input buffer; *noutput_items* outputs the same length of data as the input buffer, but this behaviour can be changed using the *forecast()* method.

A synchronous block has a ratio of 1:1, allowing users to write blocks that consume and produce the same number of items per port. Also, it can have any number of input and output ports. This block is used to create source blocks, where there are not any input ports, as well as sink blocks, where there are not output ports. The synchronous block has the same length in items for all input and output buffers.

A decimation block has a ratio of N:1. This means that the number of input items are a fixed multiple N of the number of output items. N is represented by the *decimation* factor, which is a parameter of *gr_sync_decimator* constructor, given by

$$inputitems = noutput\_items * decimation. \tag{2.1}$$

On the other hand, an interpolation block has an 1:M ratio, meaning that the number of output items is a fixed multiple of the number of input items. The *interpolation* factor is provided as a constructor parameter of the *gr_sync_interpolator* and is given by

$$inputitems = \frac{noutput\_items}{interpolation}. \tag{2.2}$$

Finally, the hierarchical blocks are written in Python. This type of block aggregates other blocks and connects them using the *connect* function. The top block is the main data structure of a flowgraph and all blocks are connected under this block. GNURadio generates the top block automatically, when all the connections and configurations are correct.

2.1.6 GNURadio Tools

GNURadio provides tools to help in project development, such as the GNURadio-Companion (GRC), *gr_modtool* and *gr_filter_design*.

GRC is an open-source Visual programming language that uses the GNURadio libraries, providing users an easy way to create GNURadio applications. Figure 2.3 gives a general idea of its interface. The GRC uses drag-and-drop to interact and define connections between the modules. GRC also provides information about the configuration of the system parameters, about its correctness, saving time and avoiding mistakes. When everything is set, GRC builds the python code, that runs the application. Thanks to the graphical and to the user-friendly interface, GRC offers an easy way of inserting and testing new modules in the system. However, GRC has some disadvantages and does not give place for block customization besides the configuration of the parameters, therefore GRC is not recommended for the implementation of a new module.

GRC is an interesting tool in an educational environment because it allows students to create and change GNURadio applications in a very short learning period. But in a research context, which needs more customization, GRC is not recommended because it slows down the research process. Instead, the researchers need to create new custom made blocks and add them to GRC, using the XML file that describes the module.

When creating new modules, it can be very difficult to create, compile and install all the files and folders necessary. For that purpose, GNURadio has a tool that allows an easy way to do it: the *gr_modtool*. In the implementation chapter it is given a short tutorial on how to build new blocks step-by-step.

In the signal processing world, it might exist interference between signals or even noise that destroys the signal. This way, the developers use filters to reduce the interference

and have a cleaner signal. GNURadio provides several filters templates and a tool called *gr_filter_design* and the users can design the filters using the parameters that GNURadio provides. More information about this tool is available in the implementation chapter.

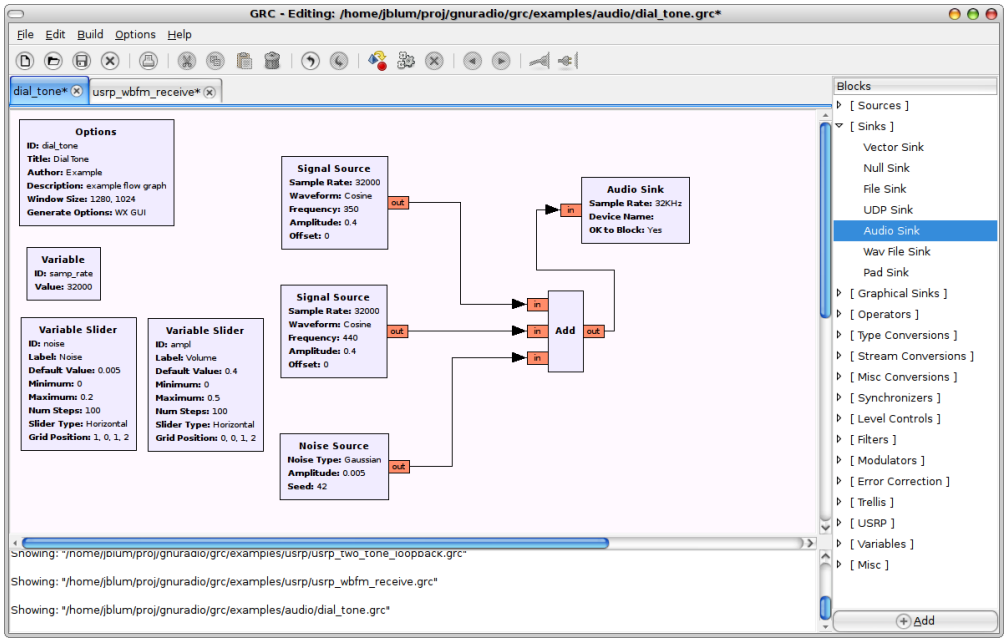


Figure 2.3: GRC interface.

2.2 Block Transmission Techniques

This section starts by providing a brief comparison between Multi-Carrier (MC) and Single Carrier (SC) modulations, which are respectively, the base theoretical concept for the OFDM and SC-FDMA. Afterwards, this section explains each transmission techniques in detail.

2.2.1 Multi-Carrier and Single Carrier Modulations Comparison

The SC transmission is a linear modulation that uses a single carrier with a high symbol rate. This way, the energy for each symbol is divided by the total transmission band. To obtain this linearity, the complex envelope of an N-symbol burst (N have to be even) must be written as

$$s(t) = \sum_{n=0}^{N-1} s_n r(t - nT_s), \tag{2.3}$$

where s_n is the complex coefficient that matches the n^{th} symbol, selected in a chosen constellation (Phase Shift Keying (PSK), or a Quadrature Amplitude Modulation (QAM)); $r(t)$ designates the support pulse with the proper constellation and T_s the symbol duration. The Fourier transform to $s(t)$ is given by,

$$S(f) = \mathfrak{F}\{s(t)\} = \sum_{k=0}^{N-1} s_n R(f) e^{-j2\pi f n T s}. \tag{2.4}$$

The transmission band for each data symbol s_n, is equal to the band occupied by $R(f)$, which is the Fourier transform of $r(t)$.

The MC transmission sends the N symbols in the frequency-domain, each in a different sub-carrier, during the same interval of time (T). The multi-carrier signal assumes the following spectrum,

$$S(f) = \sum_{k=0}^{N-1} S_k R(f - kF), \tag{2.5}$$

where S_k refers the k^{th} frequency-domain symbol, N the number of the used sub-carriers and $F = \frac{1}{T_s}$ the spacing between sub-carriers. Doing the inverse Fourier transform to each side of 2.5, leads to the dual of 2.4,

$$s(t) = \mathfrak{F}^{-1}\{S(f)\} = \sum_{k=0}^{N-1} S_k r(t) e^{-j2\pi k F t}, \tag{2.6}$$

which represents the complex envelope of the corresponding multi-carrier burst. Comparing all the above equations, it is clear that the multi-carrier modulation is dual to single carrier modulation and the same way around.

Frequency Division Multiplexing (FDM), besides being the simpler multi-carrier mod-

ulation method, gives no overlap between different sub-carriers in the spectrum. For the bandwidth of each S_k to be a fraction $\frac{1}{N}$ of the total transmission band, the bandwidth $R(f)$ must be smaller than F (F is the bilateral bandwidth and $\frac{F}{2}$ the unilateral bandwidth), as the figure 2.4 shows.

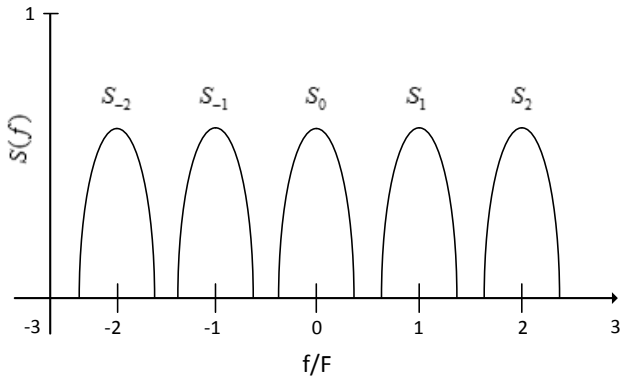


Figure 2.4: Conventional FDM [Sil10]

The Inter-Symbol Interference (ISI) is a problem in this method. To prevent it, the pulse $r(t)$ must verify the next orthogonality condition,

$$\int_{-\infty}^{+\infty} r(t - nT_s)r^*(t - n^{'}T_s)dt = 0, n \neq n^{'}. \tag{2.7}$$

In the frequency-domain, the orthogonality condition between each sub-carriers is

$$\int_{-\infty}^{+\infty} R(f - kF)R^*(f - k^{'}F)df = 0, k \neq k^{'}. \tag{2.8}$$

Using of the Parseval's Theorem in 2.8 we get

$$\int_{-\infty}^{+\infty} |r(t)|^2 \, e^{-j2\pi(k-k^{'})Ft} dt = 0, k \neq k^{'}. \tag{2.9}$$

In the SC modulation case, if we use different pulses using $r(t - nT_s)$ with $n = ..., -1, 0, 1, ...$, it still prevails the orthogonality between the pulses, even if they overlap. For example, using the pulse

$$r(t) = sinc\left(\frac{1}{T_s}\right), \tag{2.10}$$

with $sinc(x) \triangleq \frac{sen(\pi x)}{\pi x}$, the equation 2.7 is verified.

In the MC scheme the behaviour is identical and the orthogonality is still secure between the sub-carriers, even when we use distinct $R(f - kF)$ and they overlap. For instance, the orthogonality between sub-carriers (expressions 2.8 and 2.9) still happens when,

$$R(f) = sinc\left(\frac{1}{F}\right), \tag{2.11}$$

that is equal to have in the time-domain a rectangular pulse $r(t)$, with period $T = \frac{1}{F}$ [Sil10]. This way, the orthogonality condition 2.9 now is

$$\int_{0}^{t_0+T} e^{-j2\pi(k-k^{'})Ft} dt = 0, k \neq k^{'}. \tag{2.12}$$

2.2.2 Orthogonal Frequency Division Multiplexing

OFDM is a MC data transmission technique, where the data is transmitted on N narrowband parallel sub-carriers, each using a portion of the available bandwidth and spaced each other by $F \geqslant \frac{1}{T_B}$ (T_B is the period of an OFDM block). So each block is N times bigger than the symbol period. The difference between OFDM and FDM is that each sub-carrier, in the first scheme, is spaced in frequency by a minimum distance,

fulfilling the orthogonality between them. This way, the complex envelope of an OFDM is given by,

$$s(t) = \sum_{m} \left[\sum_{k=0}^{N-1} S_k^{(m)} e^{j2\pi kFt} \right] r(t - mT_B), \tag{2.13}$$

which characterizes a sum of blocks with the duration $T_B \geqslant T$, where $T = \frac{1}{F}$ (duration of the useful part of the block) and they are transmitted at a rate of $F \geqslant \frac{1}{T_B}$. The N data symbols, $S_k; k = 0, ..., N-1$, are sent during the m^{th} block and the complex sinusoids, $e^{j2\pi kFt}; k = 0, ..., N-1$, denote the sub-carriers.

Consider the m^{th} OFDM block. It can be expressed as

$$s^{(m)}(t) = \sum_{k=0}^{N-1} S_k^{(m)} r(t) e^{j2\pi kFt} = \sum_{k=0}^{N-1} S_k^{(m)} r(t) e^{j2\pi \frac{k}{T} t}, \tag{2.14}$$

with $r(t)$ being the transmitted impulse, where the time interval is bigger than T ($T_B = T + T_G$),

$$r(t) = \begin{cases} 1, [-T_G, T] \\ 0, elsewhere \end{cases} \tag{2.15}$$

the *guard interval* and is larger than 0. Although condition 2.9 is not verified by a pulse defined by 2.15, the orthogonality is still obtained in the time interval between $[0, T]$, which is the effective detection interval. Each sampling instant is given by

$$s^{(m)}(t) = \sum_{k=0}^{N-1} S_k^{(m)} e^{j2\pi kFt}, 0 \leqslant t \leqslant T_B. \tag{2.16}$$

The symbol period must be longer than the delay spread through the time-dispersive radio channel.

Using 2.6, the m^{th} block should take the form

$$s^{(m)}(t) = \sum_{k=0}^{N-1} S_k^{(m)} e^{j2\pi kFt} = \sum_{k=0}^{N-1} S_k^{(m)} e^{j2\pi \frac{k}{T_B} t} = \sum_{k=0}^{N-1} S_k^{(m)} e^{j2\pi kFt}, 0 \leqslant t \leqslant T_B, \quad (2.17)$$

with $e^{j2\pi F_k t}; k = 0, ..., N-1$ representing the sub-carriers, $S_k; k = 0, ..., N-1$ the m^{th} block data symbols, $f_k = \frac{k}{T_B}$ is the centre frequency of the k^{th} sub-carrier and $r(t)$ a rectangular pulse with a bigger duration than $\frac{1}{F}$ and equal to 1 in the interval $[-T_G, T]$. Doing the inverse Fourier transform to both sides of 2.17, we get

$$S(f) = \mathfrak{F}\{s(t)\} = \sum_{k=0}^{N-1} S_k^{(m)} sinc\left[\left(f - \frac{k}{T_B}\right)\right]. \quad (2.18)$$

The duration of each symbol is big enough to insert a guard interval between each OFDM symbols, thus removing the Inter-Block Interference (IBI). To eliminate the Inter-Carrier Interference (ICI) we need to add a cyclic prefix instead of a zero interval [Sil10]. Equation 2.17, defines a periodic function in t with a period T. The guard period is the complex envelope of the final part of the MC block (figure 2.5). Therefore, the final part of the OFDM is copied to the beginning of the transmitted frame, the guard interval, creating a periodic signal and reducing the sensitivity to the time synchronization.

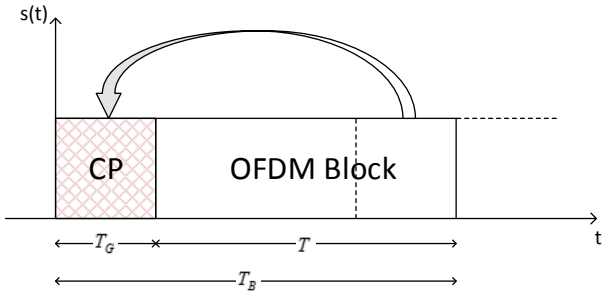


Figure 2.5: MC cyclic prefix [Sil10]

When the symbols enter the OFDM receiver scheme, they need to be synchronized because they might suffer from a frequency and/or time offset, due the channel distortion and delay. There are two types of synchronization: the maximum likelihood (ML) and pseudonoise (PN). In wireless connections we need to add additional pilot tones to perform

the synchronization [vdBMSPOB97].

Assuming that there is a block containing the data samples plus the cyclic prefix, with a size of $N_G + N$, using the ML synchronization, we observe a $2N + N_G$ size window, that certainly contain one complete OFDM $N_G + N$ symbol. By performing a series of operations, the ML gives the estimation of $\hat{\varepsilon}_{ML}$, which is the carrier offset, and $\hat{\theta}_{ML}$ that represents the channel delay estimation [vdBMSPOB97].

PN synchronization searches for a training symbol with two identical halves in the time-domain. The two halves remain identical after passing through the channel, except for phase differences, caused by the carrier frequency offset. The training symbols have the PN sequence in the even frequencies while the odd are zero by default. An accurate estimation of the carrier frequency offset and of the symbol timing. PN allow a very fast and low-overhead synchronization, which is necessary for wireless communications [SC97].

Transmission Scheme

When the data symbols enter in the OFDM transmission block, it is first converted in a serial to parallel converter. Thus, they are transformed into N size data blocks, represented as $S_k; k = 0, ..., N-1$, which are complex data symbols from a chosen constellation (PSK, QAM, etc.). Taking 2.17 and sampling the OFDM symbols with a interval, $T_a = \frac{T}{N}$, we get the samples,

$$s_n \equiv s(t)|_{t=nT_a} = S(t)\delta(t - nT_a) = \sum_{k=0}^{N-1} S_k e^{j2\pi \frac{k}{T} nT_a}, n = 0, 1, ..., N-1, \tag{2.19}$$

where $F = \frac{1}{T}$. An IFFT is applied to the data blocks, to pass them to the time-domain, so 2.19 can be written as

$$s_n = \sum_{N-1}^{k=0} S_k e^{j\frac{2\pi kn}{N}} = IFFT\{S_k\}, n = 0, 1, ..., N-1. \tag{2.20}$$

Then, a cyclic prefix is added, with N_G samples size, which are inserted at the beginning of the data block. The cyclic prefix consists in a time-domain cyclic extension of

the OFDM block. It is bigger than the channel impulse response and is attached between each block, transforming the multipath linear convolution into a circular one. With the cyclic prefix, the data block is $s_n; n = -N_G, ..., N-1$ and the time duration of the OFDM symbol is $N_G + N$ times larger than the symbol of a SC modulation. The cyclic prefix increases the cost and bandwidth, because it adds additional data. After adding the cyclic prefix, the data is converted from parallel to serial, a DAC is applied and the data is sent through a designated channel. The OFDM modulator block is depicted in figure 2.6.

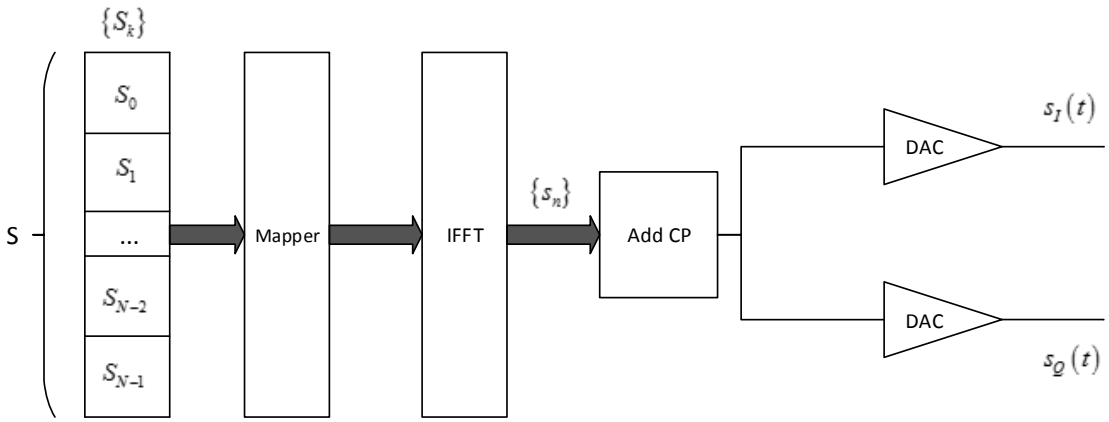


Figure 2.6: OFDM modulator block [Sil10]

Reception Scheme

Figure 2.7 illustrates the basic OFDM receiver block diagram.

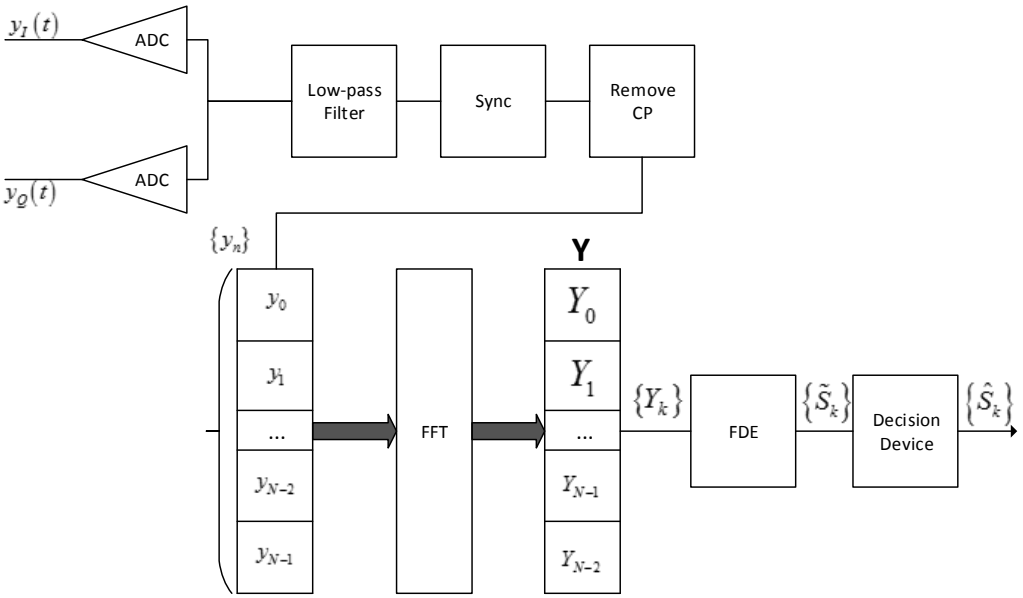


Figure 2.7: OFDM demodulator block [Sil10]

The signal that enters in the reception block, $y(t)$, is the convolution of $s(t)$ with the channel response $h(\tau, t)$ plus the noise signal $n(t)$, and comes in the form of

$$y(t) = \int_{-\infty}^{+\infty} s(t-\tau)h(\tau,t)d\tau + n(t). \tag{2.21}$$

The signal $y(t)$ is then submitted to an ADC, converting it into the sequence y_n, where $n = -N_G, ..., N-1$, which corresponds to the sampled version of the received signal $y(t)$ with a sampling rate $T_a = \frac{T}{N}$. The signal received is then filtered, by a low-pass filter, and enters in the synchronization block, which applies PN or ML synchronization [Ram08].

The received data symbols may overlap, due to multipath propagation, which leads to a loss of orthogonality between sub-carriers. The usage of the cyclic prefix with a duration T_G longer than the channel impulse response avoids the overlapping. The sequence received has $N + N_G$ samples due to the cyclic prefix in the N_G first samples, which is extracted in the next step. After this operation, the resulting samples are converted to the frequency-domain using the FFT algorithm. The frequency-domain block $Y_k; k = 0, ..., N-1$, is represented as

$$Y_k = \sum_{k=0}^{N-1} y_n e^{-j\frac{2\pi kn}{N}}, k = 0, 1, ..., N-1. \tag{2.22}$$

Since IBI is prevented using the cyclic prefix, the receiver works each sub-carrier individually. Considering flat fading on each sub-carrier and null ISI, the symbols are characterized in the frequency-domain by

$$Y_k = H_k S_k + N_k, k = 0, ..., N-1, \tag{2.23}$$

where N_k represents the additive Gaussian channel noise and H_k the overall channel frequency response for the k^{th} sub-carrier.

The OFDM sub-carrier has a narrow bandwidth when the number of sub-carrier is sufficiently large. A constant frequency-selective effect can occur, which is caused by the fading effect due to multipath propagation. In this case, the equalizer has to multiply each sub-carrier by a constant complex number. This equalization is simpler and consumes less computational process if it is done in the frequency-domain rather than in the time-domain, this is why the receiver uses an FFT to convert the N size blocks to the frequency-domain.

When the receiver obtains the Y_k samples, the equalization occurs using a Frequency Domain Equalization (FDE). It may consist in a simple one-tap equalizer under the zero forcing (ZF) criteria, where the samples are outputted as

$$\tilde{S}_k = F_k Y_k. \tag{2.24}$$

$\tilde{S}_k$ denotes the estimated data symbols, calculated from the multiplication of the inputted symbols and the equalization coefficients, $F_k; k = 0, ..., N-1$ defined by

$$F_k = \frac{1}{H_k} = \frac{H_k^*}{|H_k|^2}. \tag{2.25}$$

After the equalization, the decision device works, and based on the constellation, does the demodulation of the samples [Sil10].

2.2.3 Single Carrier - Frequency Division Multiple Access

In the 3^{rd} generation in wireless telecommunication systems, OFDM was the official modulation technique used for transmission, because it could achieve high bit rates. With the evolution to the new generation, Long Term Evolution (LTE), Single Carrier - Frequency Division Multiple Access (SC-FDMA), also referred as precoded-OFDM or DFT-Spread OFDM, takes a step in and is the new modulation technique used for the uplink wireless transmission. This technique uses different sub-carriers to transmit the symbols, but this time, the transmission of the sub-carriers is done sequentially, rather than in parallel, reducing the envelope fluctuations in the transmitted waveform [HGMG06].

The choice for SC-FDMA in LTE comes from the need to reduce the Peak-to-Average Power Ratio (PAPR), which in OFDM is greater than SC-FDMA, making OFDM unfavourable for the uplink transmission [Ahs09]. The wireless systems have severe multipath propagation, making SC-FDMA signals arrive at the base station with inter-symbol interference, but the base station cancels the interference by applying an adaptive FDE.

The distribution of the sub-carriers, in SC-FDMA, can be done in two ways. The first, called the localized SC-FDMA (LFDMA), uses a set of adjacent sub-carriers to transmit the signals, confining the bandwidth to a fraction of the system bandwidth. The second method is the distributed SC-FDMA (also known as the interleaved FDMA (IFDMA)), in which the sub-carriers are distributed over the entire bandwidth, where each sub-carrier is equidistant from one another. Figure 2.8 shows the different sub-carriers distribution [HGMG06].

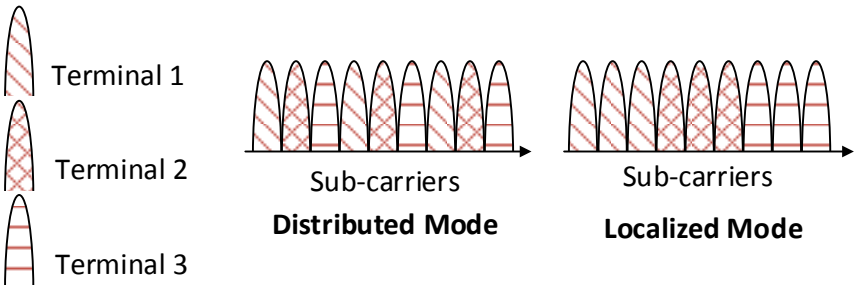


Figure 2.8: Sub-carrier allocation (Distributed mode and Localized mode) [HGMG06]

Transmission Scheme

Figure 2.9 shows the transmitter block, explained in this subsection.

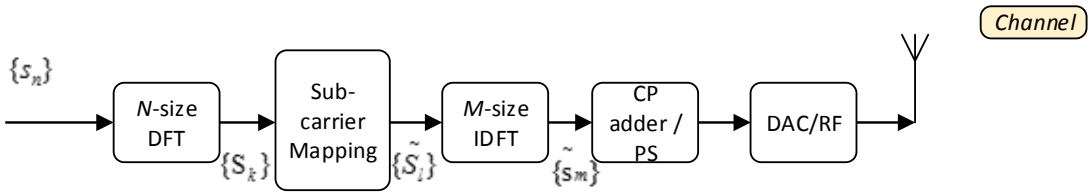


Figure 2.9: SC-FDMA modulator block

Like in the OFDM transmitter, the SC-FDMA transmitter first converts a binary input signal into a multilevel sequence of complex numbers s_n, using the proper constellation (BPSK, QPSK, 8PSK or 16/64QAM), which may differ, to match the current channel conditions. Then the stream of complex numbers enters in a serial to parallel converter. We get, in the end, the same result as the OFDM counterpart in equation 2.19. The passage of the symbols from the time-domain to the frequency-domain (S_k) is done applying a N size Fourier transform to the N symbols of the data blocks. Before going to the next step, the preamble block is inserted between chunks of symbols. The sequence, used for the preamble symbols, is the Zadoff-Chu coefficients, which is more efficiently than the

standard preamble used in OFDM [SB09]. The Chu-sequence used for the LTE systems has the following formula,

$$s_k = e^{\frac{-j\pi Rk(k-1)}{N}}, k = 0, ..., N-1, \tag{2.26}$$

where R is the root of the Zadoff-Chu sequence and N the length. In order to perform a good synchronization, this sequence is interleaved with zeros, like in the OFDM preamble system, in order to perform a good synchronization [3GP08].

From this point on, the modulator performs the sub-carriers mapping, using either IFDMA or LFDMA, mentioned above. The sub-carriers are mapped into blocks, with a size M, where M must be greater than N and $N = \frac{M}{Q}$ (Q is the bandwidth expansion factor of the symbol sequence). The result of this mapping is expressed by $\tilde{S}_l$ (with $l = 0, ..., M-1$).

In the next step, it is performed a $M - size$ inverse Fourier transform (IDFT) to $\tilde{S}_l$, converting the symbols to the time-domain, $\tilde{s}_m$. A chosen frequency carrier is used to carry all the modulated symbols, which are then sent sequentially.

Like in OFDM, the problem with IBI, caused by multipath propagation, is resolved by adding the cyclic prefix, with a size larger than the length of the channel impulse response. The additional symbols, at the beginning of the sequence, are also copied from the end of the block, similarly to the OFDM, which converts a discrete time linear convolution into a discrete time circular convolution. After adding the cyclic prefix, the signal is converted from parallel to serial, it is applied a DAC and the signal is sent through a designated channel [HGMG06].

Reception Scheme

To follow this sub-section, figure 2.10 shows the internal blocks of the demodulation block.

The symbols received are submitted to an analog to digital converter (ADC). After being converted, the symbols are filtered through a root-raised cosine. This filter is used to attenuate the out-of-band signal energy, causing less interference and conserving more

power. Depending upon the rolloff factor of the filter pulse, which convolves with the signal, the filter induces a distortion in the signal, causing an increase in PAPR. However, this does not mean that the PAPR of the SC-FDMA is higher than the OFDM, because even in the worst case scenario, the PAPR for the SC-FDMA is lower then the OFDM signal [Han09].

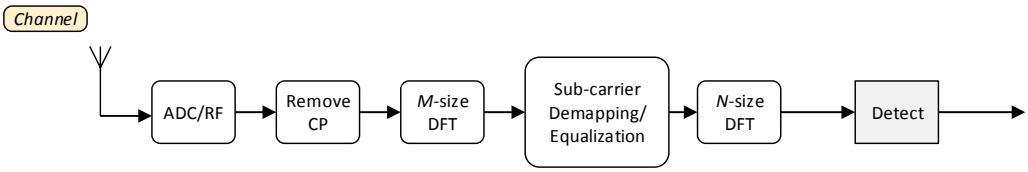


Figure 2.10: SC-FDMA demodulator block

Next, the signal enters in the synchronization block, which transforms the stream of symbols into blocks, that have a $M + N_G$ size (N_G is the guard size). The cyclic prefix is removed, resulting in M size blocks, ready for the equalization, but before this, a M size Fourier transform is applied to the data blocks, converting them from the time-domain to the frequency-domain, to simplify the equalization. The resulting SC-FDMA blocks are defined by the following equation,

$$Y_k = H_k S_k + N_k, k = 0, ..., N-1, \tag{2.27}$$

where H_k is the overall channel frequency response for the k^{th} sub-carrier and N_k represents the additive Gaussian channel noise. ISI can also occur in the SC modulation. Y_k is equalized to compensate this interference. The ZF criteria could be applied, as in OFDM. However, for SC-FDE it is proposed the use of the minimum mean square error (MMSE), which is more efficient because of the robustness against noise [Ahs09]. The coefficients F_k have the following value,

$$F_k = \frac{H_k^*}{\alpha + |H_k|^2}, k = 0, ..., N-1, \tag{2.28}$$

where α represents the inverse of the Signal to Noise Ratio. F_k coefficients multiply the data symbols to equalize them. After the equalization, a N size inverse Fourier transform (IDFT) is used to obtain the time-domain. Finally, the symbols are demodulated by a decision device.

Chapter 3

System Implementation

The GNURadio platform and the theory behind OFDM and SC-FDMA modulators were described in the previous chapter. This chapter focuses on the implementation of the SC-FDMA modulator and demodulator blocks and also shows how the tools in GNURadio work.

This chapter is organized as follows: section 3.1 starts by presenting examples of GRC software; section 3.2 shows how to create new blocks and design the filters, using the *gr_modtool* and *gr_filter_design* tools respectively; section 3.3 reveals how the OFDM technique is implemented in GNURadio, analysing the modulator and demodulator blocks; and section 3.4 explains the SC-FDMA modulator and demodulator blocks implementation.

3.1 Dial Tone Example

This section presents the Dial Tone example implemented using the GNURadio platform. This example produces an audio sound composed by two cosine signal sources and by a noise signal source. The output is converted to an audio format using the audio sink block, allowing the signal to be heard. Figure 3.1 shows the schematic for this example, on the GRC. First, the Signal Source block creates cosine waves; the frequency is adjusted using a WX GUI Slider, a variable slider that ranges from 0 to 1000; the amplitude of the wave is set to 0.1. There are two Signal Sources: one with the default frequency of 350 Hz and another with 450 Hz. The Noise Source block creates a Gaussian noise; this block also has a WX GUI Slider associated with the noise amplitude that ranges from 0 to 0.1

and has a default value of 0.005. The three source blocks are added, using an Add block, and the output enters in the Audio Sink block. To analyse the signal in the time domain, all blocks are connected to the channel plotter block WX GUI Scope Sink, which works like an oscilloscope and shows the inputted signals in the graphical interface.

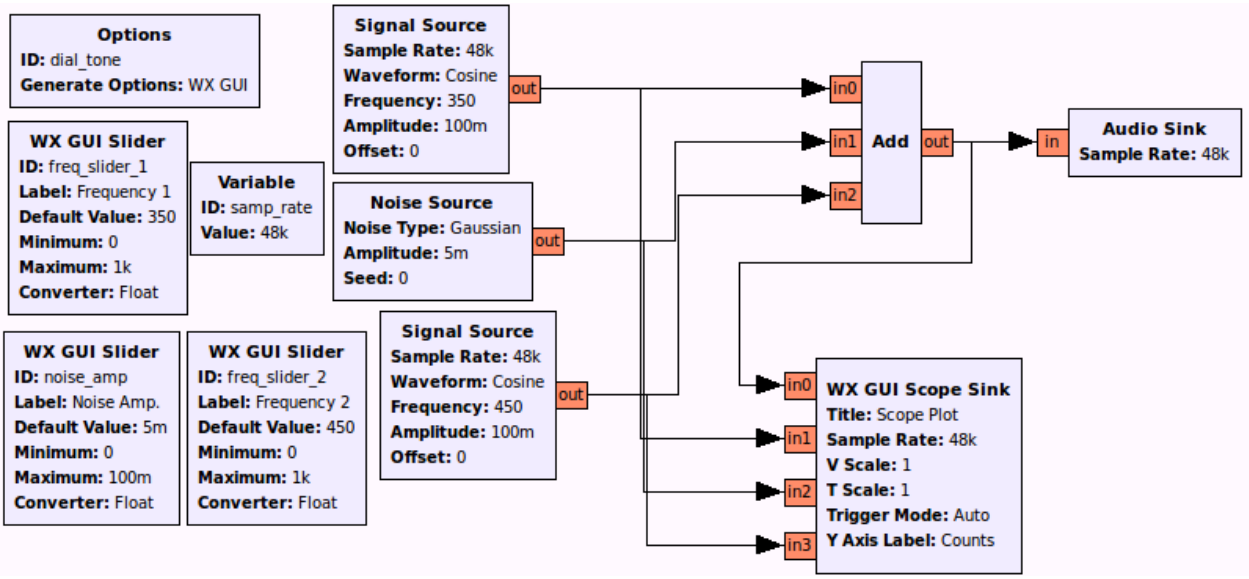


Figure 3.1: Dial Tone example - block diagram.

After connecting all the ports in the GRC schematic, the program generates the python code listed in appendix A. Two experiments were done with the setups from table 3.1, generating the graphs depicted in figure 3.2. They show four channels and each channel represents an input signal. Channel 1 is the final signal, channel 2 is the signal from signal source 1, channel 3 is the noise signal and channel 4 is the second source signal. As observed in figure 3.2, the signal in channel 1 increases as the other two increase and decreases if the other two diminish. The difference between the two graphs is the noise added to the two source signals.

Setup / Parameters	1	2
Signal Source 1 Frequency (Hertz)	450	100
Signal Source 2 Frequency (Hertz)	750	600
Noise Source Amplitude	$5m$	$30m$

Table 3.1: First examples setups.

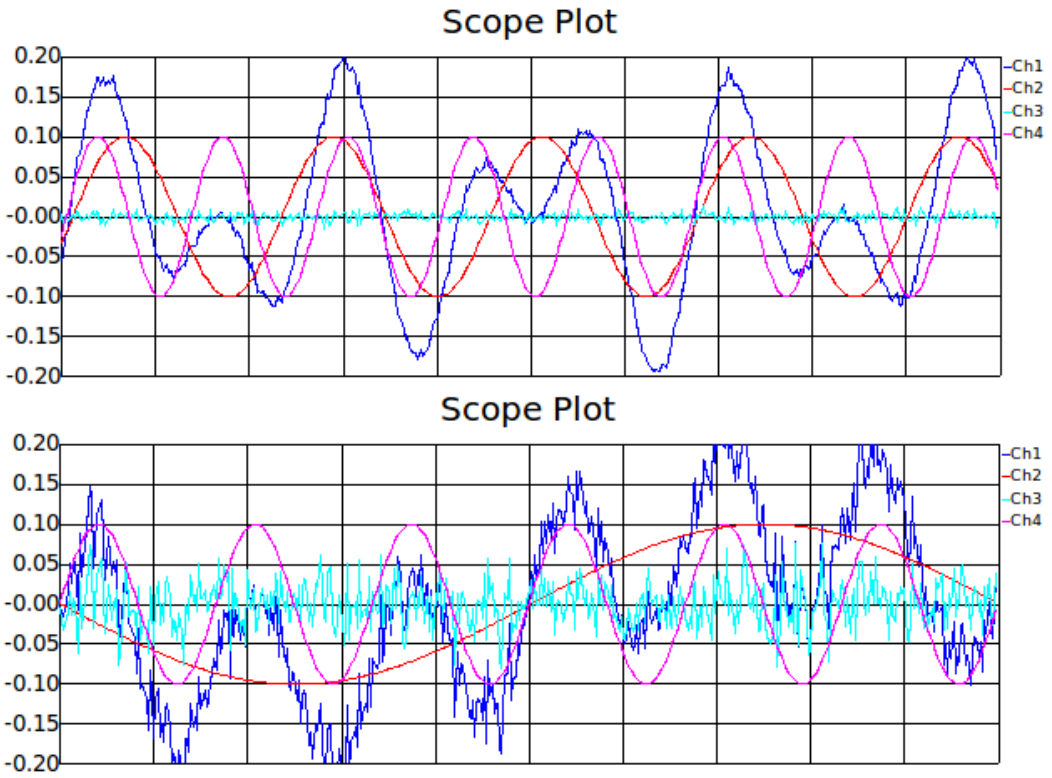


Figure 3.2: First Examples graphs - Upper graph - experiment 1. Bottom graph - experiment 2.

3.2 GNURadio Tools

GNURadio provides tools to assist the signal processing. First, the *gr_modtool* eases the creation of new modules in C++; this section has a step-by-step guide on how to create new modules on GNURadio. The second part shows the *gr_filter_design*, how it works and which filters can be designed by this tool. The operating system used to perform the tests and the implementation is Ubunto 12.04.

3.2.1 Creating New Blocks (*gr_modtool*)

Creating new blocks is a hard process. To smooth it, GNURadio offers a tool called *gr_modtool*, which creates all the folders and the skeleton files for C++.

The first step in the creation of a new block is to create a new module. In the terminal, you may call:

```
1 $ gr_modtool create
```

Then the terminal asks for the module name (let us call it *exp*) and generates a folder with the name *gr-exp*. This folder contains all the necessary sub-folders and files to compile the new blocks the user may add. In order to create new blocks, inside the folder *gr-exp*, you may call:

```
1 $ gr_modtool add
```

Now, the terminal asks which block type the user wants to create (it can be a sink, source, sync, decimator, interpolator, general, hier, noblock). Next, the terminal asks for the block name (let us call it "*square_ff*") and for the input arguments. Finally, the terminal asks if the user wants to define test codes. When the terminal finishes creating all the files and folders, it is time to program them. There are two main files that can be modified to do the signal processing; *square_ff_impl.cc* and *square_ff_impl.h* (both in *lib* folder). There is also an XML file (*exp_square_ff.xml* in the grc folder) that should be modified to prepare the block for the GRC software. Appendix B shows the modification performed in order to create a block that computes the square of the inputted signal.

With the files completely programmed, it is time to compile them; from the terminal in the *gr-exp* folder, you should call:

```
1 $ mkdir build
2 $ cd build
3 $ cmake ../
4 $ make
5 $ sudo make install
6 $ sudo ldconfig
```

This sequence of commands creates a folder named *build* and compiles the files inside it. While running the *make* command, there might exist some errors in the C++ code and the terminal alerts for them. When everything is correct, the user can access the blocks in GRC, and it is also possible to call the new blocks in python flowgraphs.

3.2.2 Filter Design Tool

The filter design tool (*gr_filter_design*) configures the filters available in GNURadio and supports the graphical representation of the characteristics of the filter defined. GNURadio has several types of filters already implemented, such as low pass, band pass, complex band

pass, band notch, high pass, root raised cosine and gaussian filters. To create the filter, there are several window functions such as: Blackman, Blackman-Harris, Hamming, Hann, Rectangular, Kaiser and Equiripple; these functions produce zero values outside of their interval and all that is left is the part where they overlap with the signal.

The application has a friendly user-interface, represented in figure 3.3. On the upper left side, the user can choose from the several filter types and windows functions. The variables below change according to the user choice, and allow the configuration of the filter's and window's parameters. The graph on the right shows the frequency response and time domain representation of the filter as well as the phase and group delay.

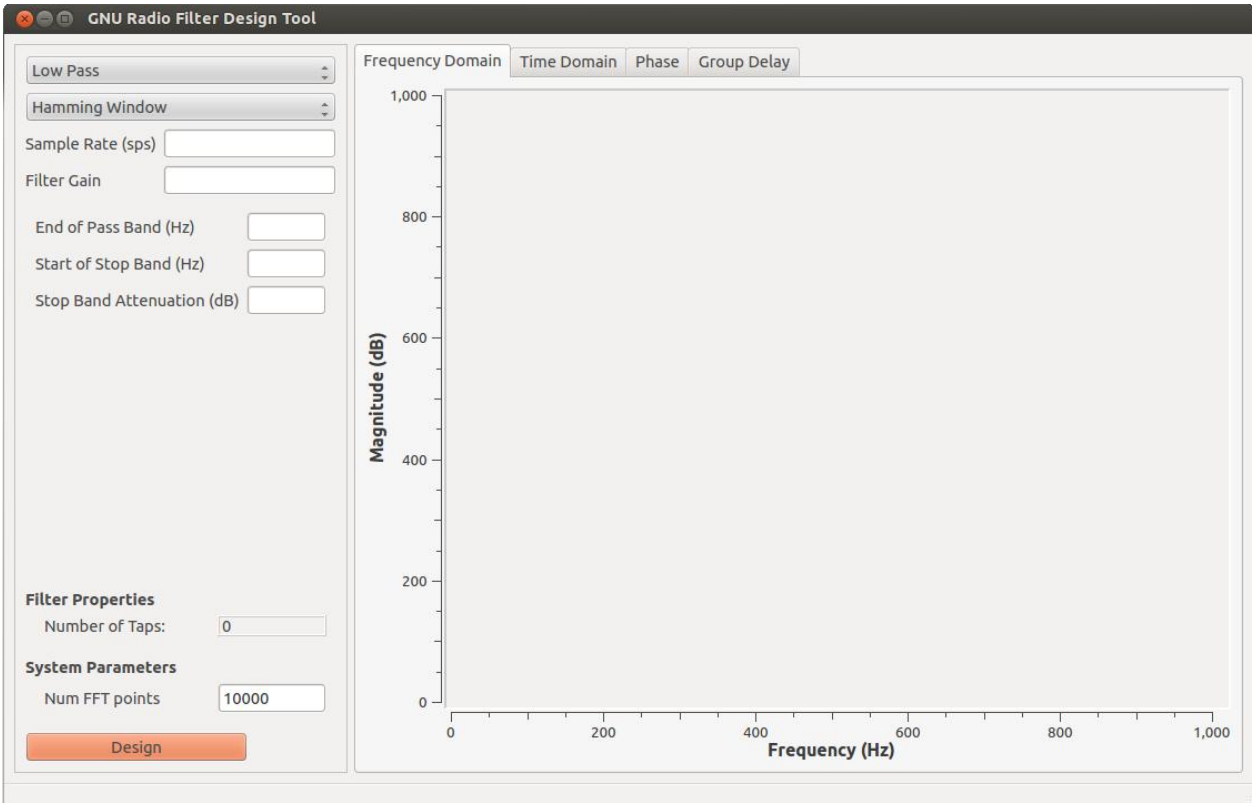


Figure 3.3: GNURadio Filter Design tool interface.

As a second example, we show the root raised cosine filter, used in the implementation of the SC-FDMA demodulator. The filter was implemented using the parameter values in table 3.2. The performance graphs are shown in the figure 3.4, with the following arrangement: on the upper side the left graph is the frequency domain graph and the right one is the time domain graph; bellow on the left is the phase graph and on the right is the time delay.

Root Raised Cosine Parameters	
Sample Rate (sps)	$2M$
Filter Gain	1
Symbol Rate (sps)	$1M$
Roll-off Factor	0.01
Number of Taps	201

Table 3.2: Parameters to design a Root Raised Cosine filter.

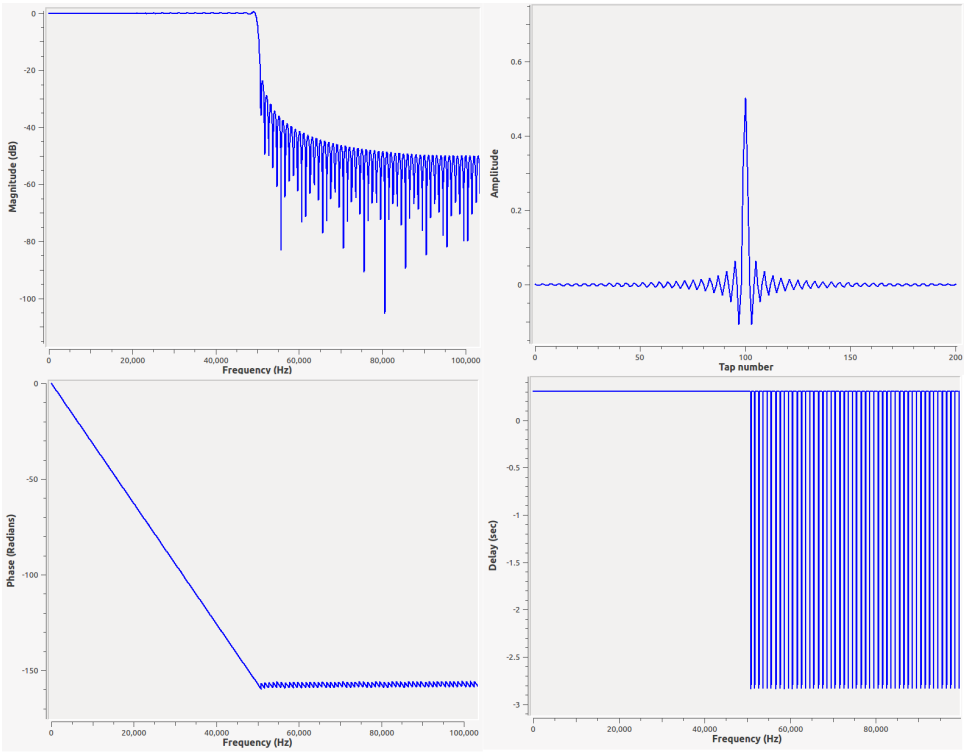


Figure 3.4: Root Raised Cosine filter performance graphs

3.3 OFDM Block

GNURadio package already includes two OFDM blocks; the modulator and demodulator blocks. Both are explained in detail in this section.

For an easy run, GNURadio comes with two benchmark applications: the *benchmark_tx.py* that works with the sender block and the *benchmark_tx.py* that works with the receiver. Both benchmarks are written in python and are located in the GNURadio digital examples folder. The files use the OFDM blocks and are ready to perform simple experiments. The benchmarks have several changeable parameters necessary to perform the experiments, such as: frequency, bandwidth and modulation (BPSK, QPSK, 8PSK

or 8/16/64/256 QAM). These parameters have to be equal in both, sender and receiver. There are other parameters that only belong to *benchmark_tx.py*, like the size and number of packets that are transmitted. When using the benchmarks, the experiments require the use of an USRP. This way, there is another parameter that defines the antenna used by the system. When the user calls the benchmarks in the terminal, he must choose which antenna is used; for the sender the user must write *RX/TX* and for the receiver *RX2*.

The benchmarks are in the top of the OFDM block hierarchy. Bellow them there are other two python files. The *transmit_path.py* file, which creates the schematic using the OFDM modulator block and helps in the definition of some parameters; and the *receive_path.py*, that besides creating the schematic, has a module called probe that detects if there is transmission.

To create the OFDM modulator and demodulator blocks, GNURadio uses the python file called *ofdm.py* (located in the gr-digital/python folder), this file defines two classes: the *ofdm_mod* for the modulator and the *ofdm_demod* for the demodulator. Figure 3.5 shows how the blocks appear in the GRC. The blocks have several parameters such as: modulation, where users may choose a constellation (BPSK, QPSK, 8PSK or 8/16/64/256 QAM); FFT length (M), necessary for the IFFT and FFT algorithms; cyclic prefix length and occupied tones (N).

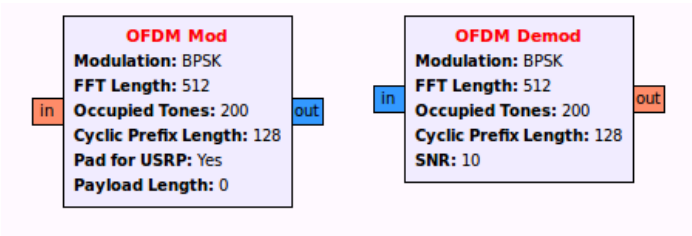


Figure 3.5: GRC OFDM blocks.

The preamble sequence has a big role in the OFDM synchronization. It is generated inside the file *ofdm.py*, using the variable *known_symbols_4512_3*. This variable is a vector with 4512 positions, each occupied with a value of 1 or -1. In the preamble sequence creation, the values are interpolated with zeros. Then, inside other modules, the preamble sequence is inserted in the right positions taking into account the occupied tones. These sequences are used to perform the synchronization.

The next two subsections explain the OFDM modulator and demodulator classes in detail. This way, our implementation of the SC-FDMA blocks can be easier to understand. Appendix C shows the most relevant code of the OFDM blocks.

3.3.1 OFDM Modulator Block

The OFDM modulator block is created in the *ofdm.py* file, at the *ofdm_mod* class. This module defines two main functions: one for changing the options and other to create messages, the *send_pkt* function. Function *send_pkt* creates data blocks with the inputted data using the function *make_packet*, from *ofdm_packet_utils*. The blocks have a header, which is used at the receiver to check its validity, and a body. The created packets are then put in a queue.

Figure 3.6 shows the OFDM modulator block schematic. Each module in the figure represents a member of the *ofdm_mod* class, and underneath them are the file names where the modules are programmed. As we can see, the modulator is composed by the following modules: *pkt_input*, *preambles*, *ifft*, *cp_adder* and *scale*. These modules are connected by one OFDM data stream, except for the first two modules that are connect by an additional data stream.

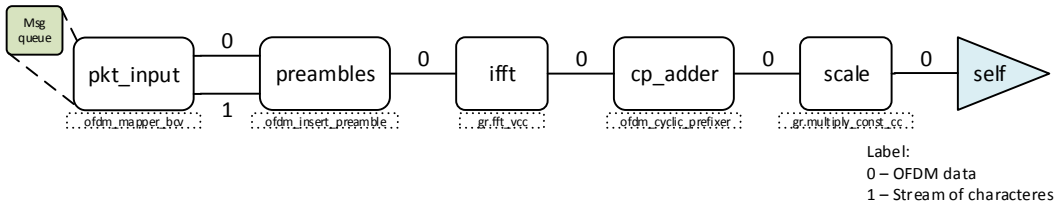


Figure 3.6: Block diagram of the OFDM modulator.

Normally, in a python block where several modules are connected between them, we use the function *connect(self, <module >)*, so that the first module receives the data directly from the block input port. Because the OFDM modulator receives input data from a message queue, the first connection does not start with *self*, but with the first module, *pkt_input*. This module withdraws the messages from the message queue and works with them. We can see how the modules are connected in *ofdm_mod* class in appendix C list C.3.

The *pkt_input* module is defined in the *C++* file *digital_ofdm_mapper_bcv.cc*. This

module receives the messages as data input and has two output ports: one port outputs the data blocks, with M symbols in each block; and the other port outputs a stream of characters delimiting the block, one for each OFDM data symbol outputted. The character outputted is 1 when it is the first symbol of the block and 0 for the remaining symbols. The character stream is consumed by the next module, which uses them to insert the preamble block in the right position. There are two important input parameters in this module:

- *d_occupied_carriers* - sets the size of the *d_subcarrier_map* vector;
- *d_fft_length* - sets the size of the outputted data blocks.

As the number of blocks outputted is higher than the size of *d_subcarrier_map*, the rest of the outputted data block is initialized with zeros. Figure 3.7 shows how the data blocks are filled. We can see that each block has M symbols but only N symbols have useful data, where M represents the FFT length and N the occupied tones length. When the *pkt_input* module is initialized, it calculates the sub-carriers tones and saves them in the *d_subcarrier_map* vector. Later, the module uses the vector to map the data inside the output data blocks.

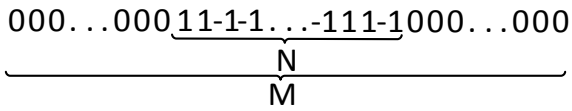


Figure 3.7: Data allocation inside each block.

As stated above, the *pkt_input* module works using messages as inputted data, drawing the messages from the message queue. The module uses the selected constellation (BPSK, QPSK, 8PSK or 8/16/64/256 QAM) and modulates the symbols inside the messages. There are some cases where the data is not enough to complete the data blocks, therefore the module uses the function *randsym* to randomly generate data symbols to fill them.

Next is the *preambles* module, defined in the *C++* file *ofdm_insert_preamble.cc*. The main objective of this module is to add the preamble block to the stream. The preamble is a

known sequence of symbols constructed from a vector of $'1'$ and $'-1'$ previously generated. The sequence enters has an input parameter and it is stored in *d_preamble* vector. The *preambles* module has two data input ports: in the first port enters the data blocks with M symbols; and in the second port enters the stream of characters outputted by the previous module. When the *preambles* module finds the character $'1'$ in the stream, it adds one preamble block and outputs the rest of the data blocks, until it finds the character $'1'$ again. More details on this module are explained in the SC-FDMA modulator subsection, which also uses it. The *preambles* module output ports have the same size as the previous block, but only the port that outputs the data blocks is connected to the next block, the *ifft* module.

The *ifft* module takes the data blocks and does the inverse Fourier transform, converting the symbols from the frequency-domain to the time-domain. IFFT and FFT blocks are defined in the *C++* file *gr_fft_vcc.cc*, which is a GNURadio generic block. The last two modules were specifically developed for the OFDM computation.

In the next step the modulator adds the cyclic prefix. This operation is performed in the *cp_adder* module and is defined in the *C++* file *digital_ofdm_cyclic_prefixer.cc*. This module uses the inputted data blocks and copies the last symbols of the block to the begin. The length of the copied symbols is equal to the *cp_size* parameter, which is an input parameter. Now, the blocks have the length of the inputted data blocks plus the length of the cyclic prefix. In the end, this module converts the blocks into a stream of symbols in order to send them later.

The data symbols are submitted to a inverse Fourier transform, which modifies their amplitude. The *scale* module compensates this deviation, multiplying the symbols by the inverse of the square root of the FFT length. After this last operation, the *scale* module connects to the *self* function, which is the output port for the OFDM modulator block. From here the symbols are transmitted, either to the USRP sink block (hardware) or to the Channel Model block (simulation). Either way, they are demodulated by a OFDM demodulator block, which is described in the next subsection.

3.3.2 OFDM Demodulator Block

The OFDM demodulator block is also created in the *ofdm.py* file, this time in the *ofdm_demod* class. Figure 3.8 illustrates the demodulator block diagram. This block has two modules: *ofdm_recv* and *ofdm_demod*. The demodulator uses the input port *self* to receive the data, which comes in the form of complex numbers. Then, in the *ofdm_recv* module, the demodulator implement the functions (filter, synchronization, de-mapping and equalization) necessary to recover the symbols. Finally, using the *ofdm_demod* module, the block demodulates the data symbols, comparing them with the constellation in use; and possibly recovering the binary messages sent by the modulator.

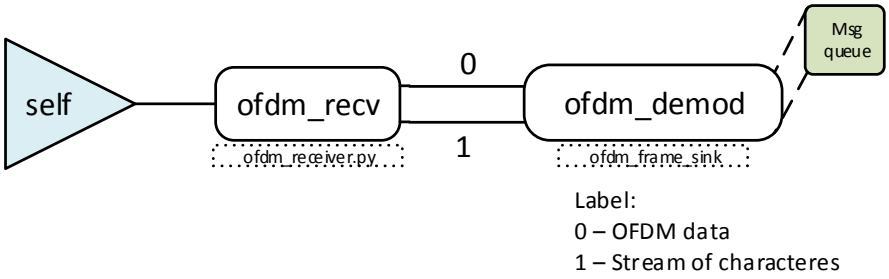


Figure 3.8: Diagram of the OFDM demodulator.

The demodulator block receives the stream of complex data, coming from an USRP sink or from a Channel Model, respectively for a hardware implementation and for a simulation on GRC software. The data enters in the *ofdm_recv* module, which is created in a different python file named *ofdm_receiver.py*. Figure 3.9 shows the block structure in the *ofdm_receiver.py*. Each module in the figure represents its name in *ofdm_receiver.py* and the file name where the module is programmed. The *ofdm_recv* block is composed by the following modules: *chan_filt*, *ofdm_sync*, *nco*, *sigmix*, *sampler*, *fft_demod* and *ofdm_frame_acquisition*. The next lines cover these modules.

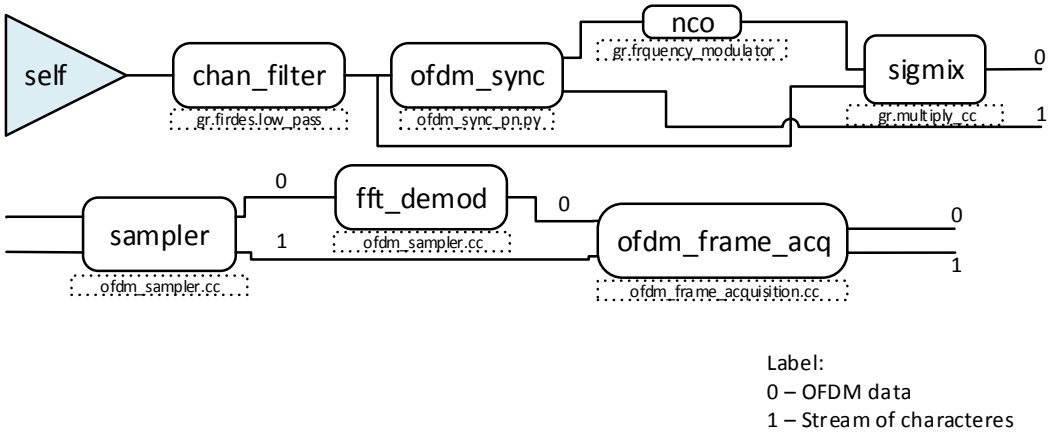


Figure 3.9: Block diagram of *ofdm_receiver.py*.

The first module is *chan_filt*, which is a low-pass filter with a Hamming Window. The filter parameters depend on the FFT size (M) and the occupied tones (N) size. The filter gain and sampling rate are fixed to 1, the cut-off frequency is $BW = \frac{N}{2M}$ and the transition width is $TB = 0.08BW$

After the filter, the data stream enters in the synchronization module *ofdm_sync*. There are several implementations for this module: the maximum likelihood synchronization (*ofdm_sync_ml*), defined in the python code *ofdm_sync_ml.py*; the pseudorandom noise numbers synchronization module (*ofdm_sync_pn*), defined in the python code *ofdm_sync_pn.py*; the enhanced pseudorandom noise synchronization module (*ofdm_sync_pnac*), defined in the python code *ofdm_sync_pnac.py*; and the fixed synchronization module (*ofdm_sync_fixed*), defined in the python code *ofdm_sync_fixed.py*. This OFDM demodulator block uses the *ofdm_sync_pn* module as the default module for the synchronization.

The module *ofdm_sync_pn* is based on the Schmidl and Cox algorithm [SC97]. This algorithm searches for the beginning of the training symbols and calculates the frequency offset. To run this algorithm, the preamble sequence must come with a predefined arrangement. For this reason, when the modulator inserts the preamble block, the symbols were originally put on the even carriers and the zeros on the odd carriers. The modulator's *ifft* module converts the preamble blocks to the time-domain and the resulting block have the symbols distributed in two equal halves, as we see in figure 3.10.

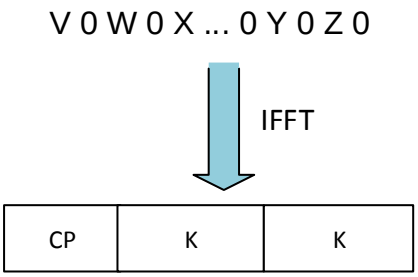


Figure 3.10: Result of the preamble block conversion.

When the *ofdm_sync_pn* module obtains a preamble sequence, the first and second half only differ on a phase shift. If the transmitting channel is constant during an interval T, if we multiply the conjugate of each symbol in the first half with the corresponding one in the second half, we cancel the effect of the channel and get the following phase,

$$\phi = \pi T \Delta f. \tag{3.1}$$

If we consider that half of the preamble block has L symbols, we can sum all the symbols within a window of $2L$ symbols. Resulting in,

$$P(d) = \sum_{k=0}^{L-1} \left(r^*_{d+k} r_{d+k+L} \right), \tag{3.2}$$

where d is the time index. The module calculates the received energy for the second half, using

$$R(d) = \sum_{k=0}^{L-1} \left| r_{d+k+L} \right|^2. \tag{3.3}$$

Last, *ofdm_sync_pn* calculates the timing metric expressed as,

$$M(d) = \frac{|P(d)|^2}{(R(d))^2} \tag{3.4}$$

When the module calculates the timing metric, the result values are bellow 1. Now, the module uses a peak detector to find the maximum value, which detects the starting carrier of the data block. So, in one port the *ofdm_sync_pn* module outputs a stream of $'1'$ and $'0'$ characters, coming from the peak detector, where $'1'$ represents the beginning of the OFDM data blocks and the $'0'$ the rest of the symbols.

Besides giving the start of the preamble block, this module gives the frequency offset estimation ($\hat{\Delta f}$) and uses it to correct the diverted symbols. This way, using 3.1 we can estimate $\hat{\phi}$ as,

$$\hat{\phi} = angle\left(P(d)\right). \tag{3.5}$$

Thus, getting

$$\hat{\Delta f} = \frac{\hat{\phi}}{\pi T}. \tag{3.6}$$

So, from the second output port in the *ofdm_sync_pn* module we get a stream of floats, each representing a frequency offset. This port connects to the *nco* module [Pag06].

The *nco* module, defined in the C++ file *gr_frequency_modulator_fc.cc*, is a frequency modulator that generates a signal proportional to the frequency offset. The outputted complex data multiplies with the outputted signal from the filter, thus making the frequency correction of the signal; this multiplication is done in the *sigmix*, a simple multiplier module.

Later, for the SC-FDMA demodulator we will use the same synchronization module and the *nco* and *sigmix* modules, with a little modification in the peak detector.

After correcting the frequency, the data stream is sampled creating data blocks with the FFT length (M) used in the transmitter. This process is done using the *sampler*

module, defined in the C++ file *ofdm_sampler.cc*. The *sampler* module receives the data stream coming from *sigmix* module and the character stream from the *ofdm_sync* module, removes the cyclic prefix and creates the data blocks. The module has two output ports: the first outputs the data blocks and the second a stream of characters. In this case, the character $'1'$ indicates the preamble block and $'0'$ the data blocks.

The next step is a Fourier transform done by the *fft_demod* module. This module converts the data blocks from the time-domain to the frequency-domain. The outputted data enters in the first port of *ofdm_frame_acquisition*.

The *ofdm_frame_acquisition* module is defined in the C++ file *digital_ofdm_frame_acquisition.cc*. The module searches for the start of the OFDM data block based on the inputted characters. At the same time that the module finds an $'1'$ character, the preamble block enters in the other input port. Using this data block, the module searches for the start of the occupied tones with the *correlate* function, because the carriers may or may not be shifted.

When the data blocks enter in the FDE module, they have the following expression,

$$Y_k = H_k S_k + N_k, k = 0, ..., N-1, \tag{3.7}$$

where Y_k represents the inputted samples, N_k the additive Gaussian channel noise, S_k the known symbols and H_k the overall channel frequency response for the k^{th} sub-carrier. To calculate the frequency response, we use the expression,

$$H_k = \frac{Y_k}{S_k}, k = 0, ..., N-1, \tag{3.8}$$

assuming that we ignore the additive noise. At this instant, when the following data blocks enter in FDE module, we need to multiply them by F_k, to perform the equalization. This way, we use the expression,

$$\tilde{S}_k = F_k Y_k, k = 0, ..., N-1, \tag{3.9}$$

where $F_k = \frac{1}{H_k}$ or $F_k = \frac{S_k}{Y_k}$.

With this in mind, the *ofdm_frame_acquisition* module enters in the function *calculate_equalizer* and using the preamble block, calculates F_k, with the following formula,

$$d\_hestimate = \frac{d\_known\_symbol}{symbol}, \tag{3.10}$$

where *d_hestimate* is F_k, *d_known_symbol* is S_k and *symbol* is Y_k.

After the preamble block, the next blocks have data symbols. Using the equation 3.9, these data blocks suffer a one-tap equalization on all sub-carriers by multiplying the received samples by the values in *d_hestimate*, using the expression

$$out = d\_hestimate * coarse\_freq\_comp * symbol \tag{3.11}$$

where *out* is $\tilde{S}_k$, *d_hestimate* is F_k and *symbol* is Y_k. When there is a carrier offset, the function *coarse_freq_comp* obtains the complex value that restores the samples to the original carrier, otherwise *coarse_freq_comp* will give the value of 1.

Eventually, the module finds a new character $'1'$ and it starts the process again for a new OFDM data block. The *ofdm_frame_acquisition* module only copies the occupied sub-carriers from the data blocks and outputs only the necessary data.

This was the last stage of the *ofdm_recv* module, which results in two output ports: one port outputs the OFDM data blocks and the other port outputs a stream of characters that indicate the start of the OFDM data block. These two output ports connect to the next module, *ofdm_demod*, which is defined in the C++ file *digital_ofdm_frame_sink.cc*. This module makes the demodulation of the data, comparing the symbols with the constellation and works like the state machine, represented in figure 3.11.

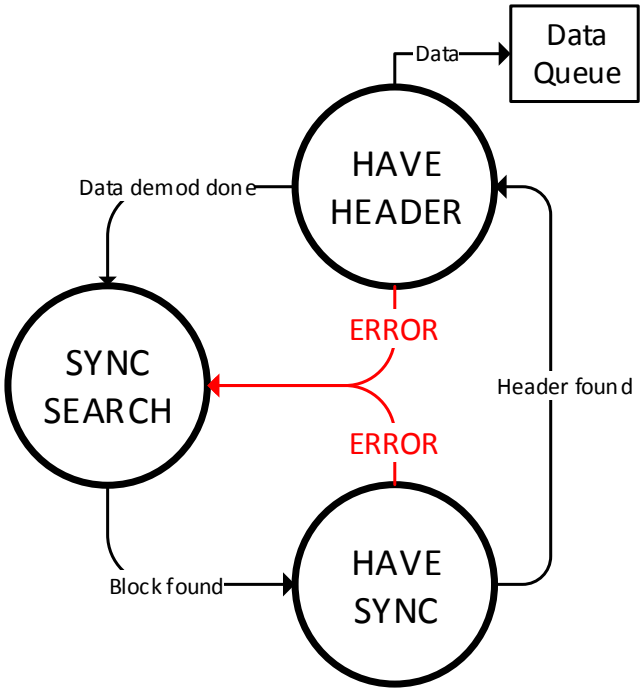


Figure 3.11: State machine in the *ofdm_demod* module.

The state machine in *ofdm_demod* module starts in state SYNC SEARCH. The module waits in this state until a flag 1 is received in the second port, signalling the beginning of an OFDM data block. Following this event, the module enters in the HAVE SYNC state and ignores the preamble data, demodulating only the symbols corresponding to the header. The header is build in a way that the first and the last half have the same data. If the header is found, the module enters in the HAVE HEADER state, demodulates the rest of the OFDM data and creates a new messages, returning in the end to the state SYNC SEARCH. The resulting messages are queued and taken to the upper levels, if the data is valid. Therefore, this module is constantly searching for the start of the next OFDM data block.

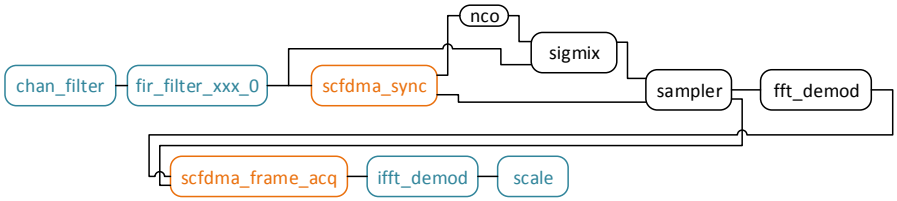
This section presented how the OFDM modulator and demodulator blocks are implemented in GNURadio. The next section explains how the SC-FDMA blocks were implemented in this dissertation.

3.4 SC-FDMA Block

This section focuses on SC-FDMA modulator and demodulator blocks implementation. It explains the modifications performed in the existing OFDM blocks, which new blocks were created and how the OFDM benchmark were adapted to the new modulation. In the previous section, OFDM blocks were described from the upper layers, starting from the benchmarks to the lower layer modules that did the signal processing. The implementation of the SC-FDMA blocks is presented in the reverse order. This section is divided in three subsections: the first subsection explains the modulator; the second subsection the demodulator; and the last one the modification done to the benchmarks and other files (XML).

In the SC-FDMA implementation, we take advantage of some modules already implemented in the OFDM as well as add new ones. Figure 4.16 shows in orange the modules that were modified, in blue the new added modules and in black the remaining modules, which are the same as the OFDM modules.

(a) Modulator Block



(b) Demodulator Block

Figure 3.12: Simplified SC-FDMA schematics.

Figure 3.12(a) shows the modulator block. The *pkt_input* module was changed from the original one and now, it modulates the symbols using the constellation and sends them in data blocks, without inserting them in the occupied tones. We added the *fft* and *scale_vec* modules to convert the symbols to the frequency-domain and changed the *preambles* module to insert the Zadoff-Chu sequence. The new *mapper* module inserts the

samples in the right occupied tones. Finally, the remaining modules *ifft*, *cp_adder* and *scale* are the same as the OFDM modulator.

Figure 3.12(b) shows the demodulator block. The new filter was added in the *chan_filt* and *fir_filter_xxx_0* modules, which now is a root-raised cosine. One part of the code was modified in the peak detector of the *scfdma_sync* module. The *nco*, *sigmix*, *sampler* and *fft_demod* are the same modules used in the OFDM demodulator. The *scfdma_frame_acquisition* module was modified to use the Zadoff-Chu sequence and a new algorithm was implemented to find the beginning of the occupied tones. Last, the *ifft_demod* and *scale* are added to convert the symbols back to the time domain in order to demodulate them.

The python file *scfdma.py* is the main file and has two classes *scfdma_mod* and *scfdma_demod*, which define the modulator and demodulator blocks respectively. In the *scfdma.py* file, the preamble sequence used is a Chu sequence and is calculated inside the modules that need them. Therefore, the code used in *ofdm.py* to store the preamble is not needed here. Appendix D shows the more relevant code implemented in order to create the SC-FDMA modulator and demodulator blocks. Next sub-section describes the modulator.

3.4.1 SC-FDMA Modulator Block

The most important input parameters in the SC-FDMA modulator block are: modulation, occupied tones (N), FFT length (M) and cyclic prefix. The modulation parameter selects the type of constellation used: BPSK, QPSK, 8PSK or 8/16/64/256 QAM. The parameter occupied tones defines the data's length stored in each data block, and defines the size of the *fft* module. The FFT length parameter provides the total size of the transmitted data blocks, and the length of the IFFT module. The cyclic prefix parameter sets the size of the cyclic prefix that is added in the beginning of each data block. Figure 3.13 shows the schematic of the SC-FDMA modulator. As before, inside the boxes is the name of the modules, below them are the names of the files where the modules are implemented. From the figure, we can see that the modulator is composed by the following modules: *pkt_input*, *fft*, *scale_vec*, *preambles*, *mapper*, *ifft*, *cp_adder* and *scale*.

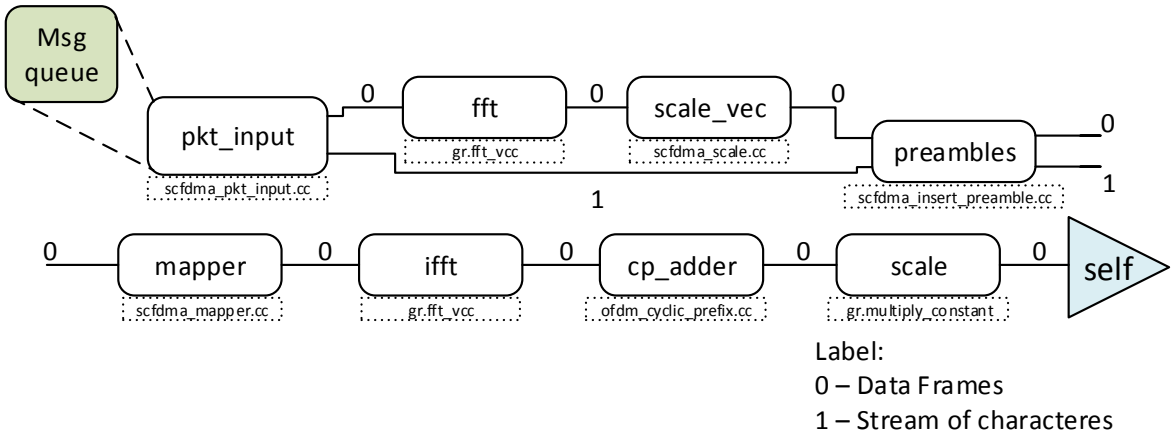


Figure 3.13: Block diagram of the SC-FDMA modulator.

Like the OFDM modulator, this modulator also receives the symbols directly from the source block and wraps them in messages using the function *send_pkt*, this way this function is still intact. The *send_pkt* function takes the inputted symbols, coming from the input port and uses the *make_packet* function, from the *ofdm_packet_utils*, to create the messages. Then, the messages are converted into strings and inserted in the message queue.

The first module of the SC-FDMA modulator is *pkt_input* and is defined in the C++ file *scfdma_pkt_input_impl.cc*. Like in OFDM, the module withdraws the messages from the queue, converts the symbols using the selected constellation and creates data blocks with N symbols. In case there are not enough data to completely fill the block, it randomly selects padding data to send. There are two output ports in this module: the first port outputs the N size data block; and the second port outputs a stream of characters, where 1 represents the first symbol of the block and 0 the remaining symbols. This module is similar to OFDM *pkt_input* however, the later maps the symbols in M size data blocks, so they can fit the IFFT module, instead of outputting all of them in a N size data block.

Before adding the preambles, the modulator needs to perform an FFT using the *fft* module to pass the symbols to the frequency-domain.

To get the original amplitude values, the *scale_vec* module divides the symbols in the block by the square root of the block length (N). The preambles are inserted in the

frequency domain, using the *preambles* module. This module is defined in the C++ file *scfdma_insert_preamble_impl.cc* and has two input ports, that are connected to the output ports from the *pkt_input* module. *Preambles* module works like the corresponding OFDM module and it can be summarized in the state machine diagram, represented in figure 3.14.

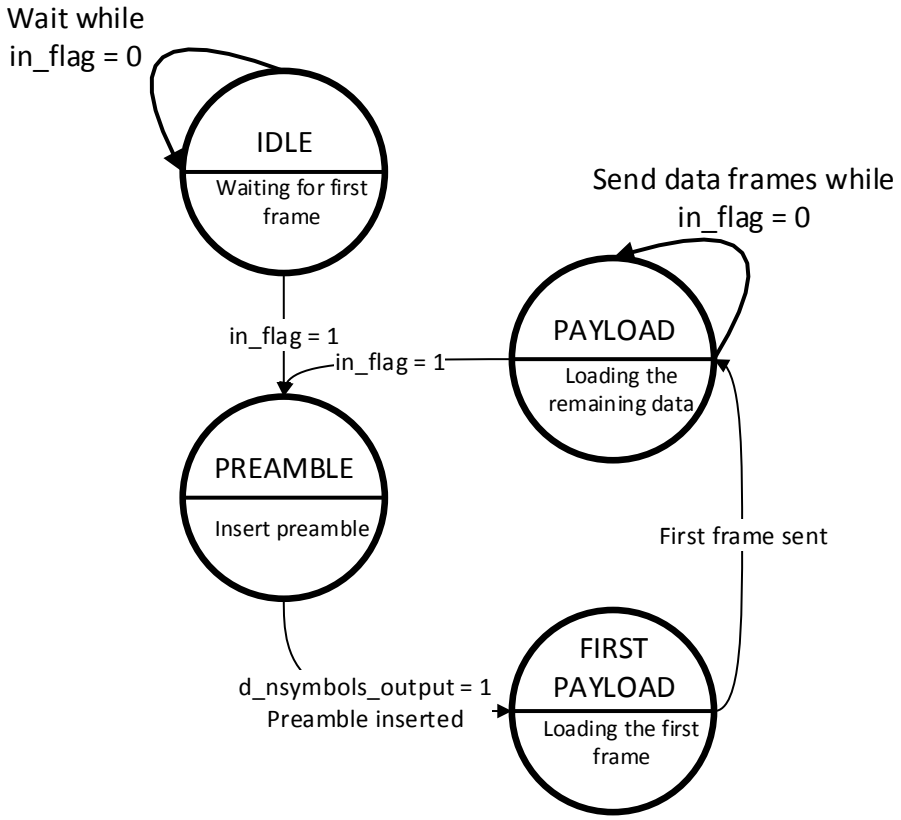


Figure 3.14: State machine in *preambles* block.

The block starts in the *IDLE* state and waits until the *in_flag* (comming from the character stream) has the value 1. The module goes to the *PREAMBLE* state, saves the data block temporarily and starts outputting the preamble block through the output port. When the preamble is fully outputted, the *preambles* module enters in the *FIRST PAYLOAD* state, where it outputs the block that entered earlier. After this, the module reaches the last state, *PAYLOAD*, and outputs the rest of the data blocks that enter, until a new *in_flag* comes with the value 1, going back to the *PREAMBLE* state. Overall, this module and the OFDM counterpart are similar. The difference is in the sequence inserted

in the preamble. This time, the sequence is the Zadoff-Chu sequence calculated when the module is created, instead of being entered as an input parameter. The sequence is still intercalated with zeros to perform the synchronization. The output ports from *preambles* are the same as OFDM, but the only output port used is the one with the SC-FDMA data blocks.

An initial implementation did not inserted the zeros and used all M symbols for data. Using this implementation made us lose some of the data in the modulation/demodulation and introduced problems in the synchronization. Therefore, each data block has a length of N but needs to be mapped to and M size data block in order to be sent, where M denotes the size of the *FFT* length parameter. The *mapper* module is implemented with that purpose in the C++ file *scfdma_mapper_impl.cc*. This module has two initial parameters required for the block initialization (occupied tones and FFT length), and one input port where the data blocks enter. In the initialization, the *mapper* module calculates the position where the data symbols are inserted in the final block, and saves them in the *d_subcarrier_map* vector. The formula and the code used is the same as the one used in the *pkt_input* module from OFDM. With the sub-carriers calculated, the module receives the blocks and inserts each symbol in the sub-carrier of the M size block, earlier initialized with 0. So, each outputted block has every position set to zero except the ones with the data symbols.

From here on, this modulator works the same way as the OFDM modulator. The data blocks have the right size for the IFFT algorithm, and that process is done using the *ifft* module and the cyclic prefix is added with the *cp_adder* module.

We saw in past experiences that the data from the IFFT module comes multiplied by the square root of the blocks length (M). To scale back the data, the modulator uses the *scale* module to divide the symbols by the square root of M. The *scale* module is the last stage of the modulator and is connected to the *self*, which connects to the output port of the SC-FDMA modulator.

Now the data is modulated, mapped, scaled and ready to be sent through the USRP or a Channel Model. These symbols are demodulated at the receiver side, presented in the next subsection.

3.4.2 SC-FDMA Demodulator Block

Scfdma.py holds the class *scfdma_demod* where the demodulator is created. It is composed by two main modules connected: *scfdma_recv* and *scfdma_demod*. The first module performs the synchronization, sampling and equalization; the second module demodulates the data and rearranges it in messages. Figure 3.15 shows how the modules in *scfdma_demod* connect. We can see that the *self* port connects to the *scfdma_recv* module and then, its output ports connect to the *scfdma_demod* module; finally, the last module is linked with a message queue.

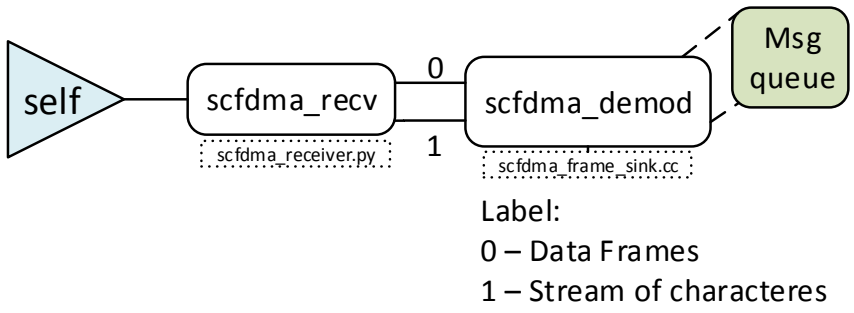


Figure 3.15: Block diagram of the SC-FDMA Demodulator.

The SC-FDMA demodulator block has different input parameters, compared to the OFDM demodulator. Besides the usual parameters (modulation, occupied tones (N), FFT length (M) and cyclic prefix), it also has two new parameters: sample rate and symbol rate. These two new parameters are used to define the root-raised cosine filter. Appendix D in list D.7 shows the code where the modules are connected.

Scfdma_recv Module

Scfdma_recv module is defined in the python file *scfdma_receiver.py*. This blocks is the core of the demodulator, and it is here that almost all the operations occur except the bit demodulation. Figure 3.16 represents the block diagram of *scfdma_recv*. The block is composed by the following modules: *chan_filt*, *fir_filter_xxx_0*, *scfdma_sync*, *nco*, *sigmix*, *sampler*, *fft_demod*, *scfdma_frame_acquisition*, *ifft_demod* and *scale*. The next paragraphs describe these modules.

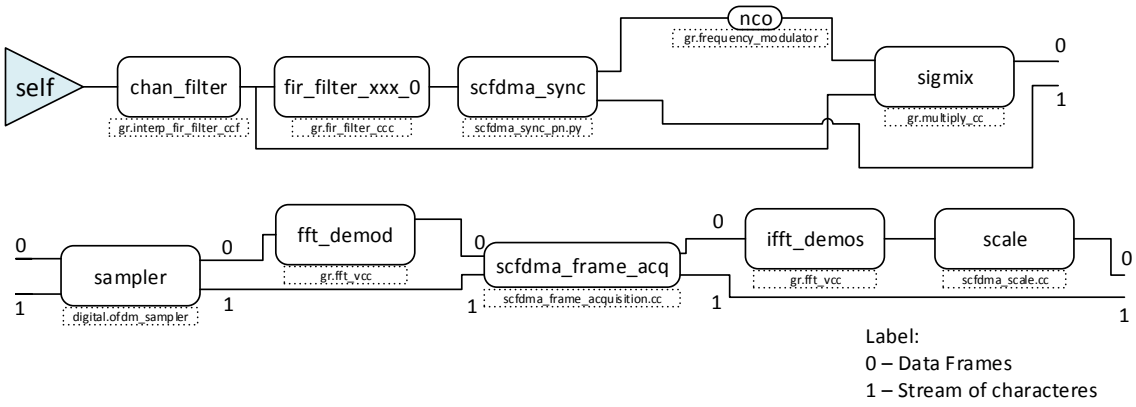


Figure 3.16: Block diagram of *scfdma_recv*.

The first module is the filter, named *chan_filt*, implemented in the object *gr.interp_fir_filter_ccf*. Unlike the OFDM filter, which is a low-pass filter, this demodulator uses a root-raised cosine filter. This module calls the interpolation filter and has two parameters: the first parameter is the number of interpolations, which we left with the value 5; the other parameter is the definition of the root-raised cosine filter, which is the constant *firdes.root_raised_cosine*. Section 3.2.2 explained how this filter works and showed the resulting frequency and time graphs. This filter increases the amount of samples, due to the interpolation. To compensate, we need to decimate the samples and get the real amount of samples. With this purpose, we use a second filter in a module called *fir_filter_xxx_0*. The decimator filter has two parameters as well: the first is the numbers of decimations (equal to the number of interpolations) and the second is "$(1,)$" indicating there is no filter associated.

The filter module is followed by the synchronization module *scfdma_sync*, which uses the PN synchronization created by Schmidl and Cox. One of the components is the peak detector module *pk_detect* and it has one problem; some data come with the value of *Not a Number* (NaN), thus breaking the chain. To resolve this issue an if/else condition was added, giving the value of 1 when the NaN appears.

After the synchronization, one of the output ports from the *scfdma_sync* module enters in the frequency modulator module, *nco*. This module is the same used in the OFDM demodulator, and it creates a signal to compensate the frequency distortion. The data

outputted from *nco* enters into the module *sigmix* to multiply with the data symbols from the filtered signal. With the symbols corrected, they need to be sampled. Just as in the OFDM demodulator, the SC-FDMA demodulator uses the *sampler* module defined in the *digital_ofdm_sampler.cc*. The *sampler* module also removes the cyclic prefix resulting in blocks with M symbols.

The data blocks are now in the time-domain. To equalize them they need to be in the frequency-domain, so the demodulator uses the *fft_demod* module to apply an FFT to convert them.

Using the *scfdma_frame_acq* module, defined in the C++ file *scfdma_frame_acquisition _impl.cc*, the demodulator does the symbols equalization. This module is similar to the *ofdm_frame_acquisition* module used in the OFDM demodulator. One difference between the two modules is that the preamble sequence enters as an input parameter in OFDM; and, in the SC-FDMA, the preamble sequence is calculated inside the module and saved in the *d_known_symbol* variable, when the module is created.*Scfdma_frame _acq* has two data input ports: in one port enters the data blocks and in the other port enters a stream of characters, where 1 represents the preamble block and 0 the rest of the data blocks. With the *scfdma_frame_acq* module running, when it receives the character 1, the module enters in the function *correlate* and uses the preamble block to calculate where the occupied tones begin. To help in the search process, the module creates a vector with N (occupied tones length) samples of 1's intercalated with 0's, let us call it U_k, where,

$$U_k = \begin{cases} 1, k = even \\ 0, k = odd \end{cases}, k = 0, ..., N-1, \tag{3.12}$$

and stores it in *d_symbol*. Also, we need to define where the occupied tones should start if there was no frequency offset, using the following operation,

$$s = \frac{M-N}{2} \tag{3.13}$$

where M is equal to the FFT length parameter and N to the occupied tones parame-

ter. With all this in mind, the *correlate* function calculates the values of the following expression,

$$N(d) = \sum_{k=0}^{N-1} U_k \left| B_{k+d} \right|, d = s - 20, ..., s + 20. \tag{3.14}$$

where B_{k+d} is the data from the preamble block and d is the analysed carrier at that time. The index d that corresponds to the maximum value of $N(d), d = s - 20, ..., s + 20$ is where the occupied tones start until the next preamble block. Now, it is time to calculate the channel estimation to do the data equalization. Using the same principle earlier described in the *ofdm_frame_acq*, the module calls the function *calculate_equalizer* to calculate the values, with the same expression as in 3.10. When the data blocks enter in the *scfdma_frame_acq* module, they suffer a one-tap equalization by multiplying the channel estimation values with the data symbols, getting the final values, like 3.11. With the data blocks equalized they are outputted from the first output port; the second output port transports the characters that symbolize the preamble and the data blocks.

The data blocks outputted from *scfdma_frame_acq* module are in the frequency domain and have N samples, including only the occupied carries. To convert them to the time-domain the demodulator uses the *ifft_demod* module. As before, the *scale* module is used to divide the inputted symbols by the square root of N to compensate the FFT amplitude deviation. The output port from the *scale* module and the second output port from *scfdma_frame_acq* module connect to the next module, *scfdma_demod*.

Scfdma_demod Module

The symbols are prepared to be demodulated by the *scfdma_demod* module, implemented in the C++ file *scfdma_frame_sink_impl.cc*. This module resembles to *ofdm_demod* in the OFDM demodulator, and has the same behaviour defined by the state machine, previously explained in section 3.3.2. The symbols in *scfdma_demod* are stored in messages and the block puts them in the message queue, so they can be later used by the upper layers of software. The only difference between this module and the OFDM counterpart is one code line that gets all the occupied tones instead of ignoring two in the middle, appendix D in list D.11 shows the difference. The next subsection explains the modification

done in benchmarks and XML files, to adapt them to the new SC-FDMA blocks.

3.4.3 Other Files

There are important files that need to be rewritten, to adapted them to the SC-FDMA blocks.

The XML files configure the blocks in the GRC. They are indispensable to run the experiments, to check if the module is running correctly. The XML file for the modulator is similar to the OFDM. The difference is that it calls *scfdma.scfdma_mod* in the $< make ><$ $/make >$ parameter instead of *digital.ofdm_mod*. For the demodulator XML file, besides changing the *scfdma.scfdma_demod*, two new parameters are added: one for the sample rate (*samp_rate*) and another for the symbol rate (*sym_rate*). These two new parameters are used in the filter design. Figure 3.17 shows how the blocks appear in GRC.

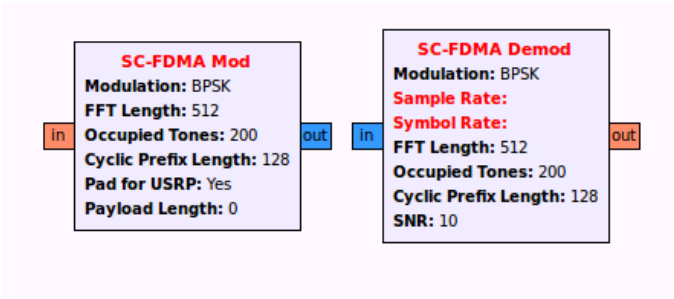


Figure 3.17: GRC SC-FDMA blocks

The benchmarks files (*benchmark_rx.py* and *benchmark_tx.py*) and files related to them (*receive_path.py* and *transmit_path.py*) were also modified. These files now call SC-FDMA blocks and the functions associated to them.

With SC-FDMA blocks implemented, the next step was to test them. Next chapter shows some testing results and compares the performance of SC-FDMA and OFDM.

Chapter 4

Performance Analysis

Last chapter described the implementation of the OFDM and the SC-FDMA transmission techniques. This chapter evaluates the performance of SC-FDMA and compares it with its counterpart, OFDM.

This chapter is composed by three sections: the first section shows the signals present on several points of the OFDM and SC-FDMA modulation techniques and compares the two modulations; the second section tests the two modulation techniques with the channel model block (in the GRC software), with different noise and frequency offset configurations for both modulations; the last section shows a set of performance results (Packet Error Rate (PER)) for both transmission techniques using two USRP nodes connected through a loop-back cable and wirelessly through antennas, for different frequencies.

4.1 Tests on GRC Using a Perfect Channel

This section shows the signals present on several points of the OFDM and SC-FDMA modulations for the same parameters, considering a perfect channel without noise or frequency offset. The purpose of these tests is to measure the data at specific points inside the modulator and demodulator blocks. To compare the two modulation techniques, both use the parameter values represented in table 4.1.

Parameters	
FFT Length	1024
Cycle Prefix Length	100
Constellation	QPSK
Occupied Tones Length	256
Sample Rate	1 MHz

Table 4.1: Parameters used in GRC tests of the modulation techniques.

The inputted signal was also the same for both modulations. We created a file with 10000 random samples of ones and zeros, which was used at all the tests in the simulator; when we get to the real channel the messages sent are created by the *benchmark_tx.py* file. The samples were generated in the GRC software, using a Random Source block and were saved in a file. Figure 4.1 shows the first 100 symbols used to perform the tests.

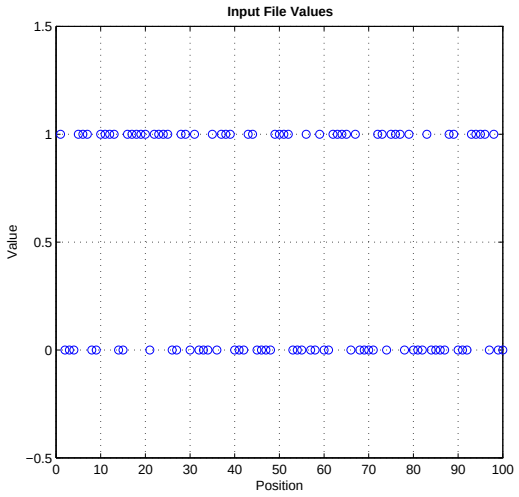


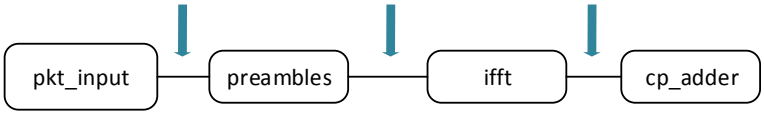
Figure 4.1: First 100 symbols of the inputted file.

This section is divided in two parts: the first part is devoted to OFDM transmission and the second part to SC-FDMA.

4.1.1 OFDM Transmission

This section shows the signals present in the OFDM blocks, when they are tested without noise or frequency offset. This section is divided in two parts: the first shows the modulator of and the second the demodulator.

Figure 4.2 shows using a blue arrow which modules are observed in the modulator and demodulator blocks.



(a) Modulator Block

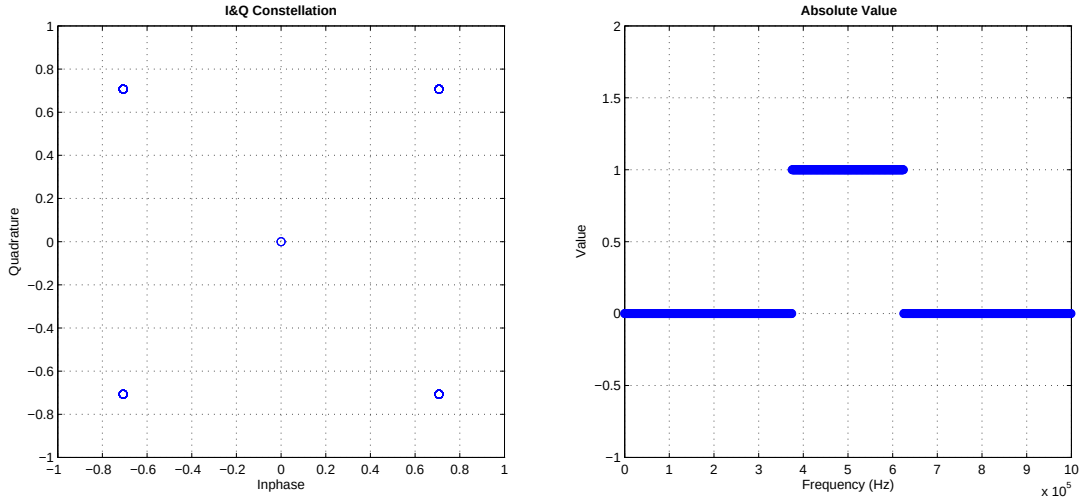
(b) Demodulator Block

Figure 4.2: OFDM schematics indicating which outputs are observed.

As we can see, in the modulator block (figure 4.2(a)) this section examines the outputs from *pkt_input*, *preambles* and *ifft* modules and in the demodulator block (figure 4.2(b)) it examines the outputs from *sampler*, *fft_demod* and *ofdm_frame_acq* modules.

OFDM Transmitter

In the transmitter side, the modules take the inputted data from one file and modulate the symbols using the QPSK constellation, generating the output represented in figure 4.3(a). Because we work with a QPSK there are four points in the constellation. The values on zero represent the unoccupied carriers, which do not have any value. *Pkt_input* module also maps the data symbols in the correct occupied carriers. Figure 4.3(b) shows the absolute value of the data symbols. Since we use a sample rate of $F_s = 1MHz$ and a FFT length of $N = 1024$, each bin occupies $\frac{N}{F_s} = 976Hz$. The data blocks have the band occupied between 375 to 625 kHz with the data samples and the rest of the band with zeros.



(a) Data block constellation

(b) Data block absolute value

Figure 4.3: *Pkt_input* module - data block outputted.

The next module is *preambles*. This module adds the preamble blocks to the stream, which are used in the synchronization and equalization processes later in the demodulator. Figure 4.4 shows the distribution of the vector with the preamble samples, before it occupy the preamble data block. We can see that the samples assume values of 1 or -1 and are intercalated with zeros. This pattern is used later to perform the OFDM synchronization and equalization.

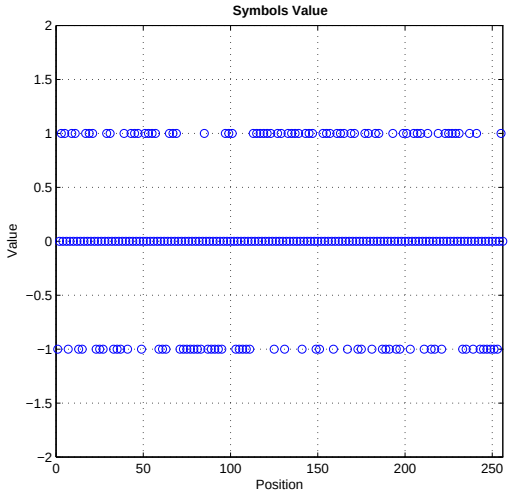


Figure 4.4: Preamble samples vector.

The last module of the modulator is *ifft*. This module converts the data blocks from

the frequency-domain to the time-domain. Figure 4.5 shows the preamble in the time domain. There are two graphs in the figure showing the absolute value of the symbols: the first graph shows the first half of the preamble block and the second graph the other half. We can see that the two halves are equal. This happens because the preamble block has its data intercalated with zeros in the frequency-domain.

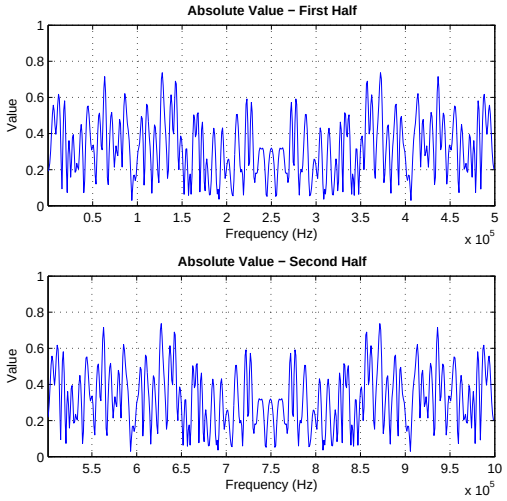


Figure 4.5: The absolute value of the preamble block symbols in the time-domain.

OFDM Receiver

After filtering the signal, and synchronize and correct the frequency offset, the demodulator uses the *sampler* module to sample the blocks and mark the cyclic prefix. The blocks outputted from the *sampler* module are in the time-domain. Figure 4.6 shows the absolute value of the samples in one preamble block outputted from this module, marked as preamble to allow the following module to exclude them. We can see that both halves are equal to each other. This is the same behaviour as in the preamble block outputted from the *ifft* module of the modulator. Therefore, the synchronization was done correctly.

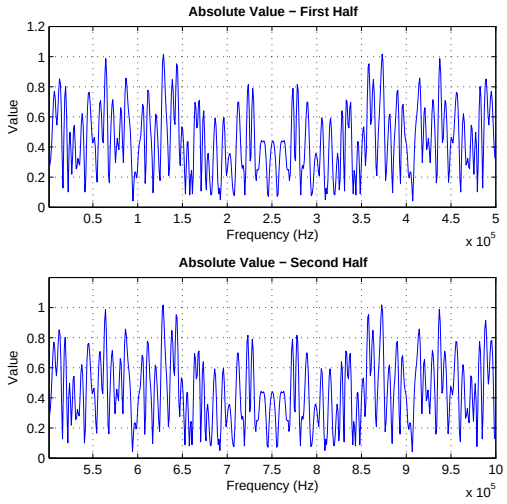
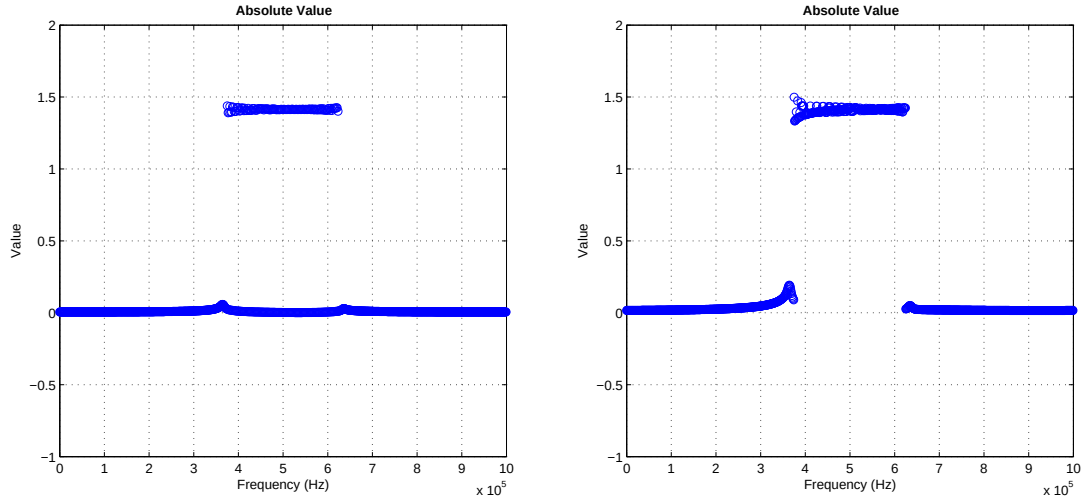


Figure 4.6: The spectrum of the preamble block symbols after the *sampler* module.

The demodulator converts the data blocks from the time-domain to the frequency-domain using the *fft_demod* module. Figure 4.7(a) shows a preamble block and figure 4.7(b) shows a data block. In both figures, only the band between 375 to 625 kHz has data samples with values much higher than zero. Figure 4.8 shows the a data block before the equalization, it has a circular form because the samples are still modulated.



(a) Preamble block (b) Data block

Figure 4.7: Frames after being converted in *fft_demod* module.

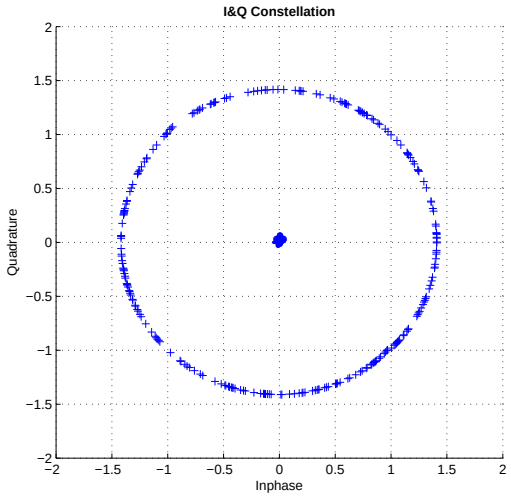
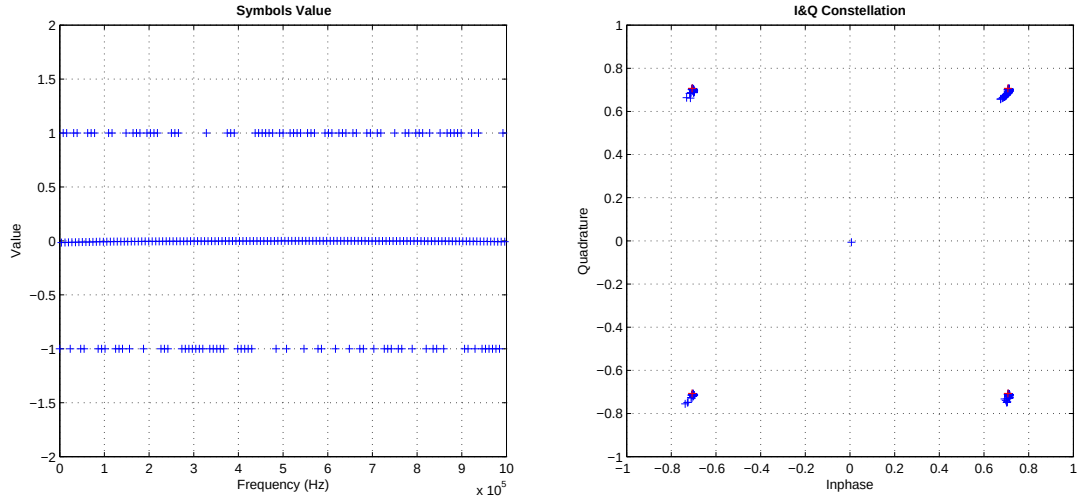


Figure 4.8: Constellation of a data block - QPSK symbols marked in red.

Using the *ofdm_frame_acq* module, the demodulator equalizes the data blocks and removes them from the occupied carries, outputting only the necessary data. Figure 4.9(a) shows the preamble block. We see that the values are 1 or -1 intercalated with zeros. Figure 4.9(b) shows a data block. It can be seen that the symbols are in the right position of the constellation, except for a small variations due to the operations earlier performed (filter, synchronization, etc.). The data blocks occupy the full band between 0 and 1 MHz.



(a) Preamble block

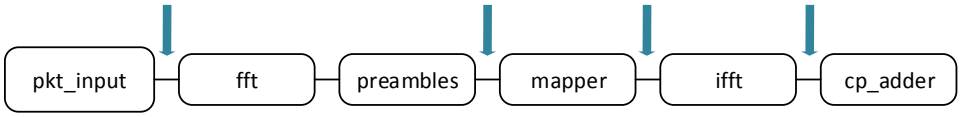
(b) Data block constellation

Figure 4.9: Frames after equalization inside the *ofdm_frame_acq* module.

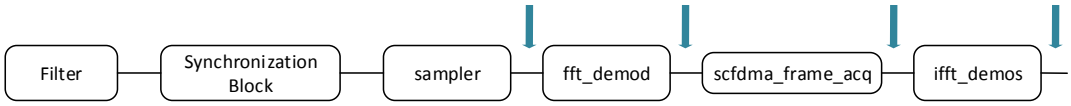
4.1.2 SC-FDMA Transmission

This section shows some samples of the SC-FDMA blocks signals, when they are tested without noise or frequency offset. This section is divided in two parts: the first shows the modulator and the second the demodulator results.

Like in the OFDM technique, not all the blocks can be observed. Figure 4.10 shows which modules are observed in the modulator and demodulator blocks; the blue arrow indicates which outputs are observed in this section.



(a) Modulator Block



(b) Demodulator Block

Figure 4.10: SC-FDMA schematics indicating which outputs are observed.

As we can see, at the modulator block (figure 4.10(a)) only the outputs of *pkt_input*, *preambles*, *mapper* and *ifft* modules are examined, and from the demodulator block (figure 4.10(b)) only the outputs of *sampler*, *fft_demod*, *scfdma_frame_acq* and *ifft_demod* modules are examined.

SC-FDMA Transmitter

The transmitter block starts with the *pkt_input* module. This modules takes the inputted data from a file and modulates the symbols using the constellation. Figure 4.11 shows the points of the QPSK constellation. The outputted data blocks from *pkt_input* module occupy the full band from 0 to 1 MHz. After this module, the modulator converts the data from the time-domain to the frequency-domain using the *fft* module.

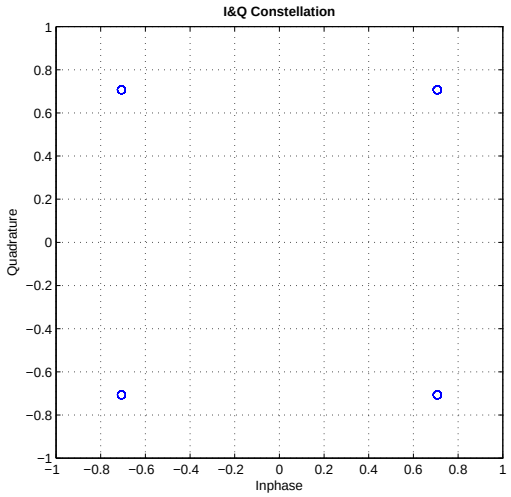


Figure 4.11: Data block constellation - outputted from *pkt_input* module.

The modulator inserts the preamble blocks in the stream, using the *preambles* module. The preamble sequence is based on the Zadoff-Chu sequence (section 2.2.3). Figure 4.12 depicts the preamble block generated. We can we see that this preamble is different than OFDM.

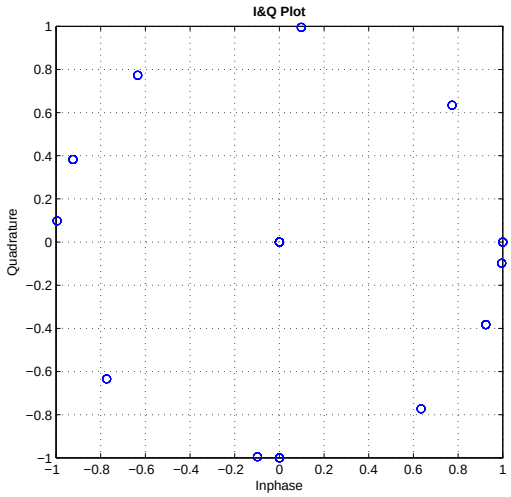


Figure 4.12: Preamble sequence.

After inserting the preamble blocks, the modulator maps the data blocks in the occupied tones, leaving the rest of the block filled with zeros. Figure 4.13 shows a data block after being mapped. The figure shows the absolute value of the symbols and as we can see, only the band between 375 to 625 kHz is occupied with values higher than zero.

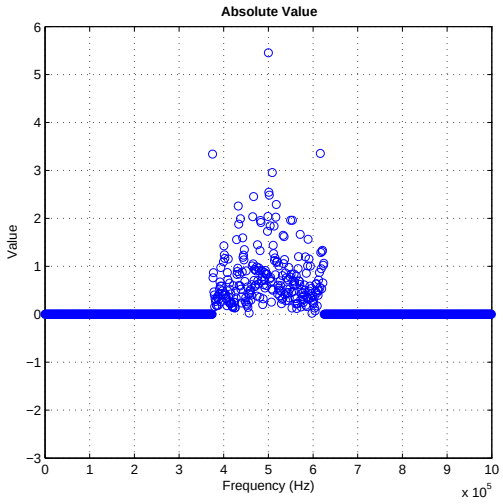


Figure 4.13: Spectrum of the symbols in a data block after mapping.

The last module observed is *ifft*. This module converts the data blocks from the frequency-domain to the time-domain. Figure 4.14 shows the absolute value of a preamble block outputted from this module. There are two graphs in the figure: the first graph shows the first half of the preamble block and the second graph the other half. We can see that the two halves are equal. This happens because the preamble block has its data intercalated with zeros in the frequency-domain. This is the same behaviour as the OFDM technique and it is also used in the synchronization at the receiver.

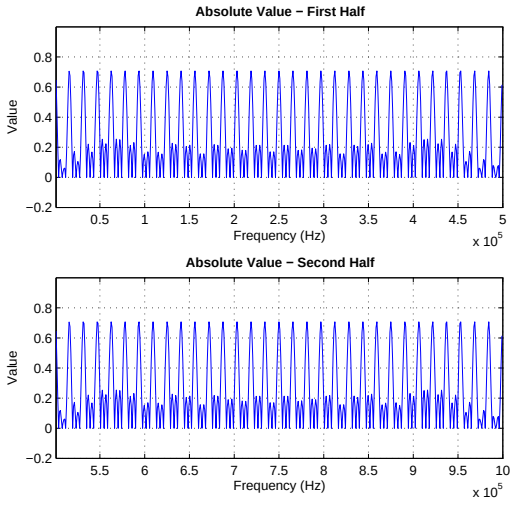


Figure 4.14: Absolute value of the preamble block symbols in the time-domain.

SC-FDMA Receiver

After filtering the signal, synchronize and correct the frequency offset, the demodulator samples the blocks and marks the cyclic prefix, using the *sampler* module. The outputted blocks are still in the time-domain. Figure 4.15 shows one preamble block outputted from the *sampler* module. We can see that both halves have the same behaviour, indicating that the synchronization was correctly done.

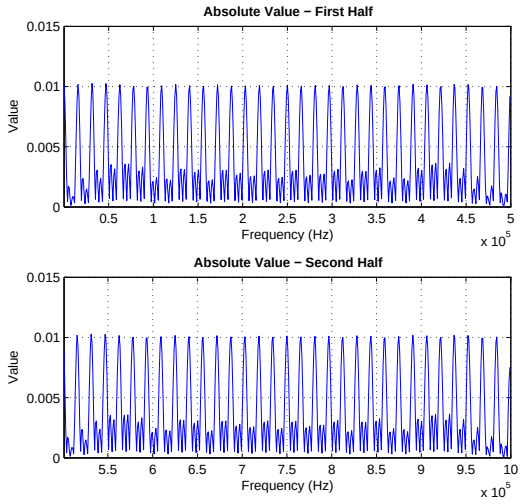
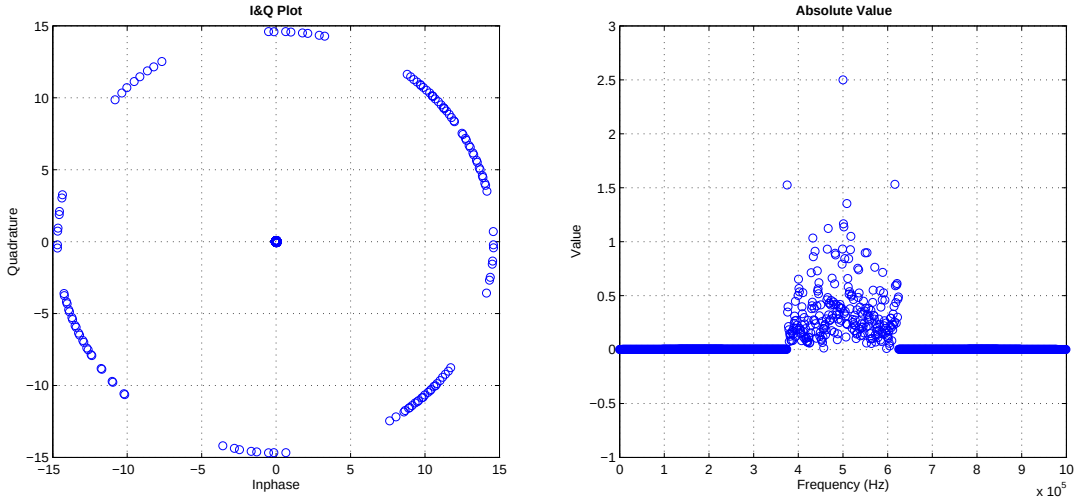


Figure 4.15: The absolute value of the preamble block symbols after the *sampler* module.

The demodulator converts the blocks outputted by the sampler from the time-domain to the frequency-domain using the *fft_demod* module. Figure 4.16(a) shows the I&Q plot of a preamble block before the equalization at the *fft_demod* module. Compared to the figure 4.12, we see that the symbols are not in the same place. Figure 4.16(b) shows the spectrum of a data block, where we can easily mark which positions belong to the occupied tones and which are only zeros.



(a) Preamble block (b) Data block

Figure 4.16: Preamble and data block after the *fft_demod* module.

Using the *scfdma_frame_acq* module, the demodulator equalizes the data blocks and removes the data from the occupied carriers, outputting only the raw data. Figure 4.17 shows a preamble block. Comparing with the figure 4.12, we now see that the symbols are in the right position.

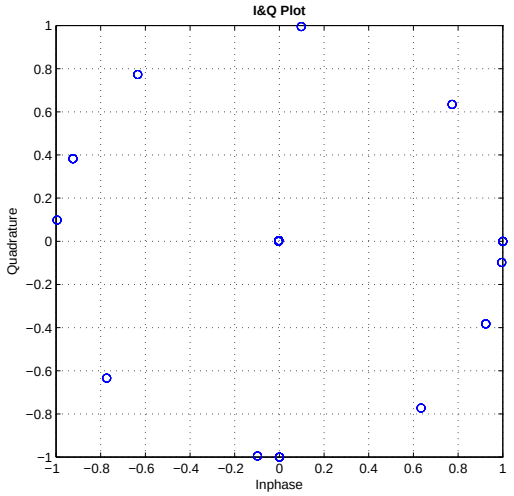


Figure 4.17: Preamble block after the equalization in the *scfdma_frame_acq* module.

The last step is to convert the blocks to the time-domain using the *ifft_demod* module. Figure 4.18 shows a data block after the conversion. We can see that the symbols are in the right position of the constellation (QPSK), marked in red. There are some variations

but this behaviour is normal due to the operations that were earlier performed (filter, synchronization, etc.), just like in the OFDM modulation, although less significant.

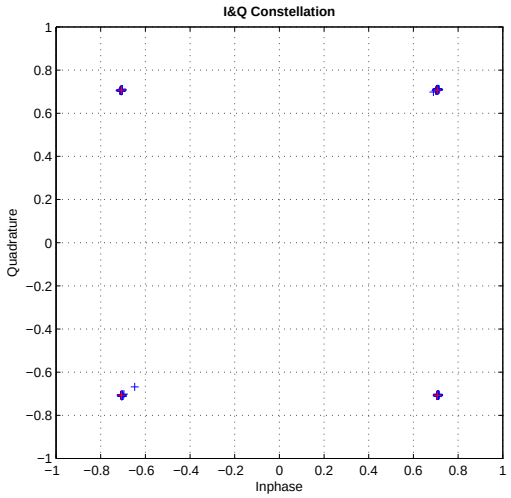


Figure 4.18: Constellation of a data block after being converted to the time-domain - QPSK symbols marked in red.

4.2 Tests on GRC Using Different Noise and Frequency Offsets

In this section, we add noise and frequency offset to the channel while testing the transmissions techniques. These tests compare the overall performance between each technique. This section is divided in two parts: the first part tests the system with different noise powers and the second part tests the system applying frequency offsets.

During the tests, we used the Channel Model block between the modulator and the demodulator blocks. Figure 4.19 shows the Channel Model block and lists its parameters. The two main parameters used in the tests are: the *Noise* (measured in voltages) and the *Frequency Offset* (measured in hertz).

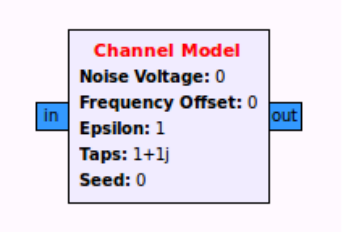
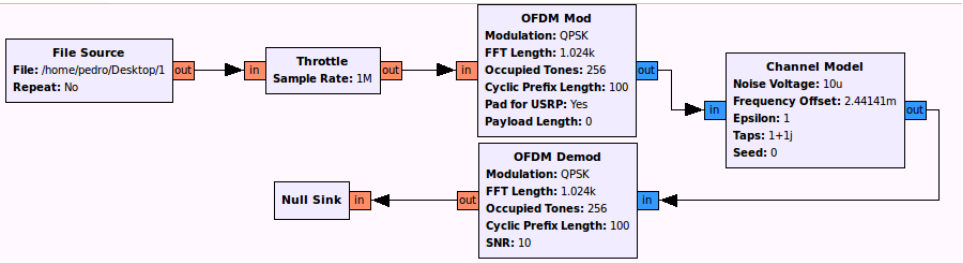
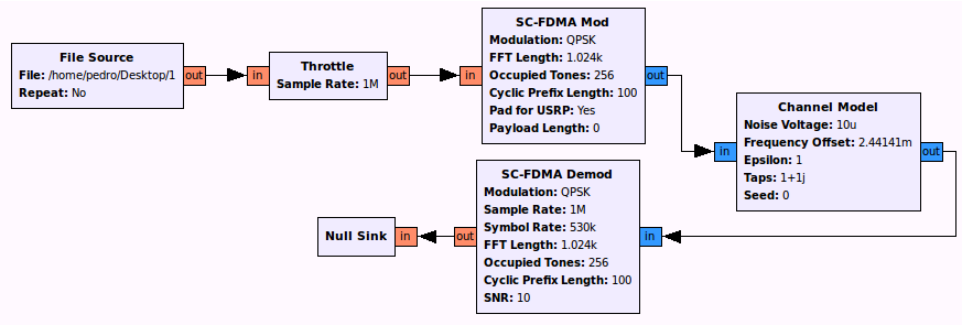


Figure 4.19: Channel model block.

Figure 4.20 shows the connections using the transmission techniques and the Channel Model block. Figure 4.20(a) shows the testing schematic using the OFDM technique and Figure 4.20(b) shows the schematic using the SC-FDMA technique.



(a) OFDM



(b) SC-FDMA

Figure 4.20: Transmission blocks with the Channel Model block in the GRC.

To compare the transmission techniques, we used a file with 10000 random binary samples and the same parameters used in the previous tests, such as: FFT length, occupied tones, constellations and cyclic prefix, with the values presented in table 4.2.

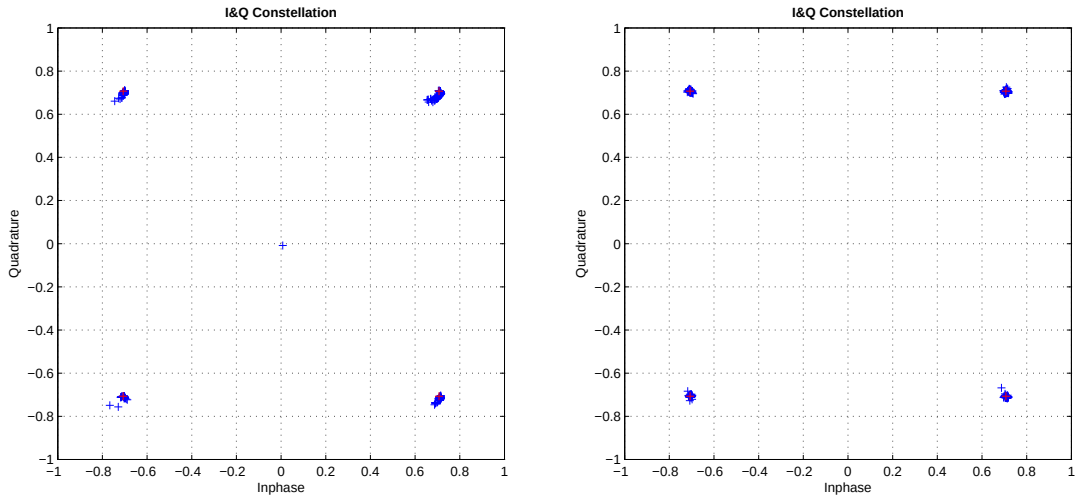
Parameters	
FFT Length	1024
Cycle Prefix Length	100
Constellation	QPSK
Occupied Tones Length	256

Table 4.2: Parameters to test the modulation techniques.

The data blocks have the same length of the FFT length parameter (1024), for this reason the inputted file is divided in 144 data blocks. The bandwidth in these tests is also $F_s = 1MHz$. We will see in the results sections that different noise voltages and frequency offsets disturb the synchronization and subsequently the final results.

4.2.1 Noise Tests Results

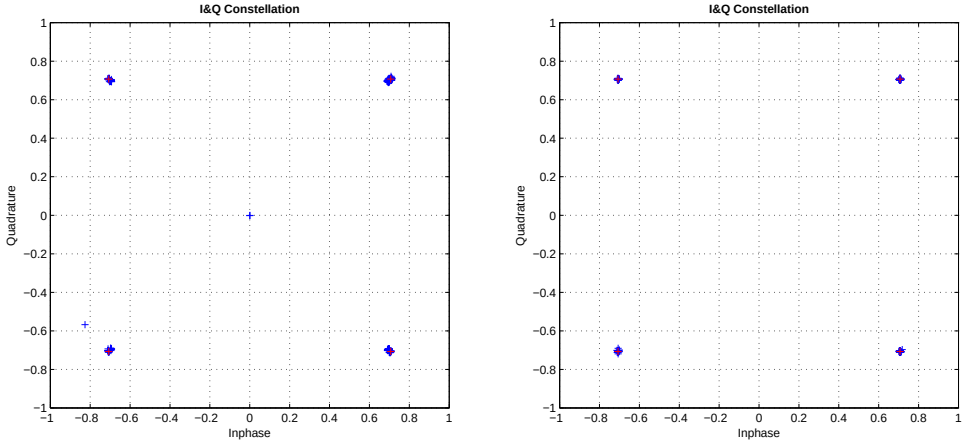
In a environment with a noise voltage of 0.00001 ($0.01\mu W$), the final samples in the constellations do not diverge much from the original ones. Using this voltage, the receiver in both transmissions can recover all the data blocks. Figure 4.21 shows the final constellations of the same data block transmitted, for both transmissions techniques. When we compare 4.21(a) and 4.21(b), we see that the SC-FDMA samples have less noise associated.



(a) OFDM (b) SC-FDMA

Figure 4.21: Data block after the equalization, with a noise voltage of 0.00001 in the channel.

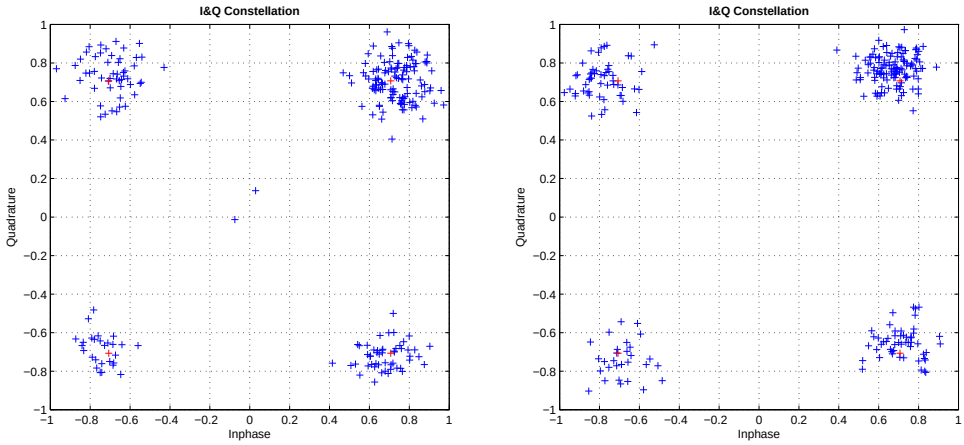
When we increase the noise voltage to 0.001 ($1\mu W$), all the data blocks are received and recovered as well. Thus, we obtained the final constellations, shown in figure 4.22. Like the last test, when we compare 4.22(a) and 4.22(b), we see that the SC-FDMA technique recovers the samples with less noise associated.



(a) OFDM (b) SC-FDMA

Figure 4.22: Data block after the equalization, with a noise voltage of 0.001 in the channel.

In the last test , we increase the noise voltage to 0.1 ($100\mu W$). In this environment, the SC-FDMA does not hold up, because the initial preamble blocks receive too much noise, which in turn disturb the final values, obtaining a PER of 13.2%, whereas in the OFDM transmission, all the data blocks were received.



(a) OFDM (b) SC-FDMA

Figure 4.23: Data block after the equalization, with a noise voltage of 0.1 in the channel.

This way, some adjustments needs to be done in the preamble sequence. Figure 4.23 shows the constellation of the same data blocks received by the two transmission techniques. Figure 4.24 shows one data block that was unsuccessfully equalized in the

SC-FDMA transmission. We can see from the last figure that the final values are shifted from the original values, marked in red, which results in a bad demodulation.

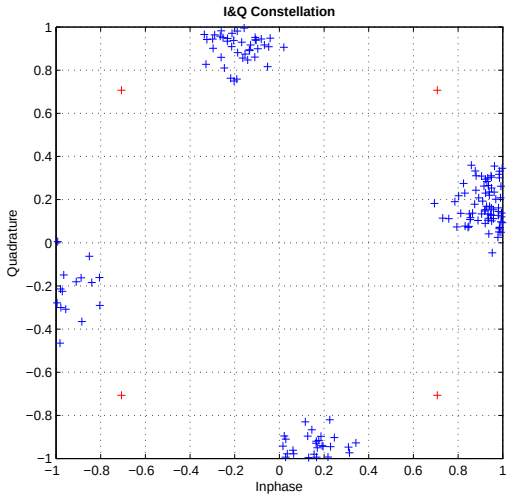


Figure 4.24: Unsuccessfully equalized data block in SC-FDMA.

In the final analysis, if we compare the graphs from the figures 4.21, 4.22 and 4.23, we can see that when we increase the noise voltage, the difference between the samples after the equalization and the original ones also increase.

4.2.2 Frequency Offset Tests Results

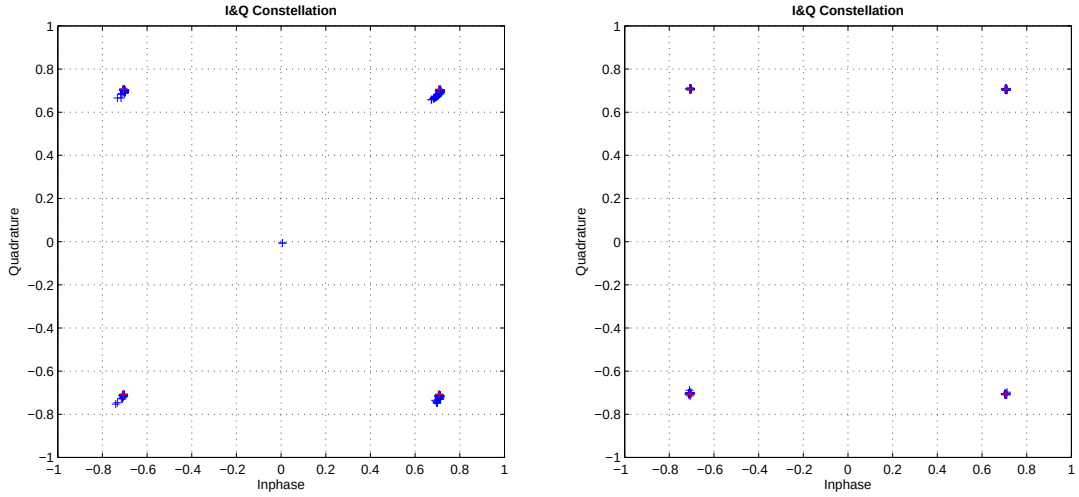
The tests with different frequency offsets considered the noise voltage equal to 0.00001, thus, the noise does not disturb too much the received symbols. The frequency offset changed according to the expression

$$frequency\_offset = \frac{1}{FFTlength} * \alpha. \tag{4.1}$$

where $\alpha > 0$ and $\frac{1}{FFTlength}$ is the size of one sub-carrier. In the tests, we tested a low offset below one sub-carrier, corresponding to $\alpha < 1$ and one higher offset of $\alpha = 2.5$.

For $\alpha = 0.7$, the frequency offset was set to $683\mu Hz$. In this channel, both transmission techniques have problems recovering the first data blocks. But, after those blocks they stabilize and the demodulator is able to recover the remaining data blocks, achieving a PER of 8% for both modulations. The constellations for both transmission techniques are

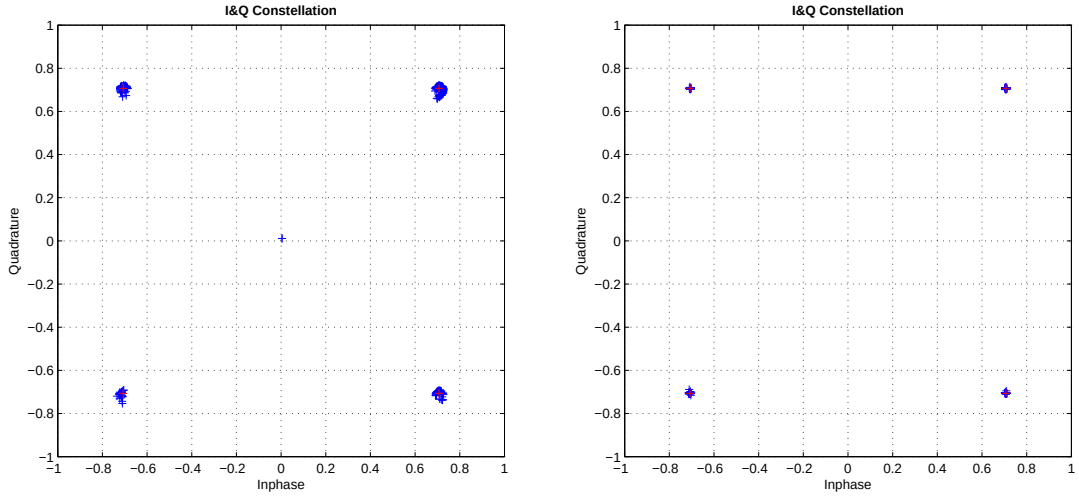
shown in figure 4.25.



(a) OFDM (b) SC-FDMA

Figure 4.25: Data block after the equalization, with a frequency offset of $683\mu Hz$ in the channel.

For $\alpha = 2.5$, the frequency offset is $2.441mHz$. Like in the last tests, both transmission techniques have problems recovering the first data blocks, but after them, the demodulators recover the remaining ones. We also measured a PER of 8% for both modulations. Figure 4.26 shows the constellations for the same data block.



(a) OFDM (b) SC-FDMA

Figure 4.26: Data block after the equalization, with a noise voltage of 0.1 in the channel.

In spite of having lost data blocks in some tests, the overall results for the SC-FDMA transmission technique are similar to the ones with the OFDM transmission technique. Furthermore, the SC-FDMA transmission achieve less noise after the equalization, comparing to the OFDM transmission.

4.3 Tests on USRP Hardware

In this section, we test both modulation techniques on the USRP hardware using the loop-back cable and the antennas. The testing system is composed by two personal computers each connected to one USRP through a USB cable. The USRP device used in the experiments is the B100 system, from the bus series.

We used the *benchmark_tx.py* to setup the transmitter and *benchmark_rx.py* to setup the receiver, and set the parameters for both modulations identical to the last experiments, listed in table 4.3.

Parameters	
FFT Length	1024
Cycle Prefix Length	100
Constellation	QPSK
Occupied Tones Length	256

Table 4.3: Parameters used by both modulation techniques.

SC-FDMA also has two new parameters related to the filter design, *symb_rate* and *samp_rate*. *Symb_rate* needs to be a little higher than the bandwidth value and *samp_rate* is two times higher than the bandwidth. We set the bandwidth with 500 kHz. Table 4.4 shows the remaining parameters for the SC-FDMA modulation.

Parameters	
Samp_rate	1 MHz
Symb_rate	530 kHz

Table 4.4: Some parameters used for the SC-FDMA modulation.

The benchmarks measure how many packets the receptor get and how many were correct. This numbers provide us the value of the PER. We also calculate the Signal to

Noise Ratio SNR at the beginning of the demodulator, using the expression,

$$SNR = 10 * log\left(\frac{P_S}{P_N}\right) \tag{4.2}$$

This section is divided is two parts: the first shows the results using the loop-back cable and the second shows the results using the antennas.

4.3.1 Results Using the Loop-back Cable

In this test, we connect the two USRP devices with a loop-back cable, thus we have less noise and can work with a broader frequency range. Figure 4.27 shows a picture of the setup with the two USRP and the loop-back cable.

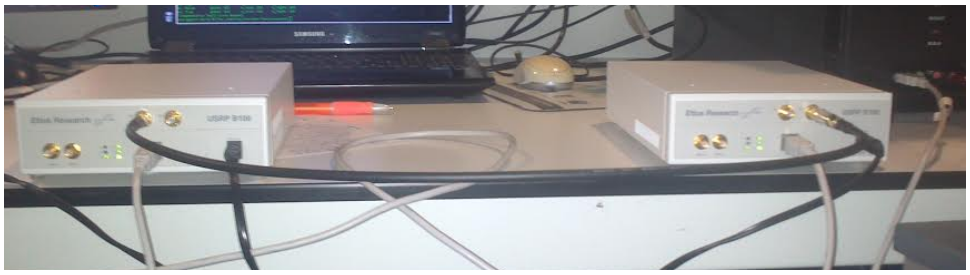


Figure 4.27: USRP setup using the loop-back cable.

We tested the two transmission techniques for three different frequencies (1, 2 and 2.5 GHz), and measured the number of packet received. OFDM and SC-FDMA experiment results are presented in table 4.5 and table 4.6, respectively. The Packets Received column gives the total of received packets and the Packets Right column gives the number of packets that were successfully demodulated.

	Packets Received	Packets Right	SNR (dB)	PER (%)
$F_1 = 1GHz$	89	87	83	3
$F_2 = 2GHz$	125	40	66	68
$F_3 = 2.5GHz$	218	10	58.9	96

Table 4.5: OFDM experiments results using the loop-back cable.

From table 4.5, we see that as we increase the frequency we get less packets right (i.e.

higher PER). Also, higher frequencies results in lower a SNR.

	Packets Received	Packets Right	SNR (dB)	PER (%)
$F_1 = 1GHz$	163	131	83	19.6
$F_2 = 2GHz$	113	91	63	20
$F_3 = 2.5GHz$	138	116	45.4	16

Table 4.6: SC-FDMA experiments results using the loop-back cable.

Similarly to OFDM, in table 4.6 as we increase the frequency, SC-FDMA has lower SNR. But when we compare the SC-FDMA to the OFDM modulation, we see that SC-FDMA receives more data successfully and achieves better results for higher frequencies.

	OFDM	SC-FDMA
$F_1 = 1GHz$	3 %	19.6%
$F_2 = 2GHz$	68 %	20 %
$F_3 = 2.5GHz$	96%	16%

Table 4.7: OFDM and SC-FDMA PER values.

In summary, the OFDM modulation technique achieves a better result in the first frequency, whereas the SC-FDMA modulation technique is more stable for all the frequencies.

4.3.2 Results Using the Antennas

In this tests, we send the data between the two USRPs using antennas. The noise interference is more relevant and the frequency band is more restricted. Figure 4.28 shows the setup with the two USRP and the antennas.

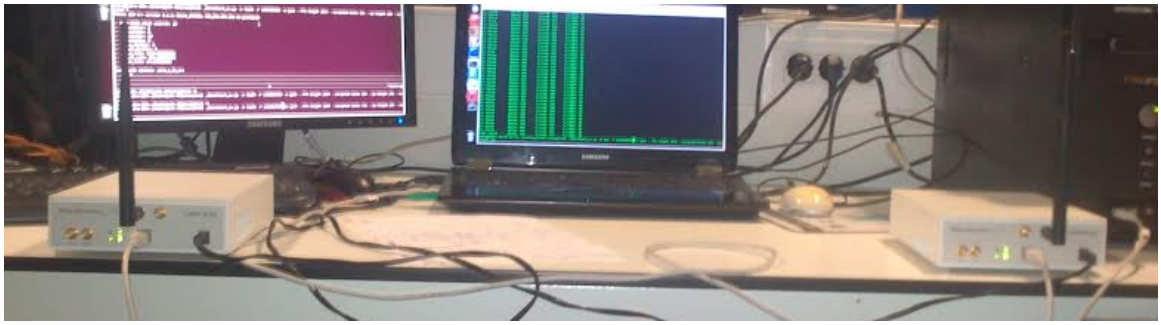


Figure 4.28: USRP setup using the antennas.

We use the VERT2450 antennas, which supports frequency bands of 2.4 to 2.48 GHz and 4.9 to 5.9 GHz. In order to performed the experiments, we set the frequency to 2.48 GHz. OFDM and SC-FDMA experimental results are presented in table 4.8.

	Packets Received	Packets Right	SNR (dB)	PER (%)
OFDM	371	33	44	91.1
SC-FDMA	236	184	30	22.1

Table 4.8: OFDM and SC-FDMA results using the antennas and with a frequency of 2.48 GHz.

In this results we see a lower SNR, due to the higher interference captured by the antennas. Similarly to the loop-back cable experiments, the SC-FDMA results are better than the OFDM for this higher frequency band. We see that the PER in the OFDM modulation is much higher than in the SC-FDMA modulation.

In order to get better results and find a better setup, more tests could be done by changing some parameters, such as: center frequency, FFT length, occupied tones, cyclic prefix, constellation, bandwidth, filter design and preambles. Also, more results could be obtained like the Bit Error Rate (BER) and a better SNR estimation, using a SNR estimator in the demodulator.

In conclusion, although these results result from a small set of experiments using the default parameters, the SC-FDMA implementation shown a acceptable performance, when compared to the OFDM implementation, that came with the GNURadio platform.

Chapter 5

Conclusions

In this chapter, we summarize the work performed along this thesis and deliver some final considerations. We also present future work that is achievable from this implementation or by changing it.

5.1 Final Considerations

This thesis focuses on the study and implementation of the SC-FDMA transmission technique using the GNURadio platform. This objective was successfully executed and now this platform has a new transmission technique.

Chapter 2 introduced the basics of the SDR. We saw that the USRP hardware sends and receives signals through out the spectrum frequency using the antennas and also converts the signals from the analog to the digital domain, while the GNURadio platform works with the digital signals using software, in C++ and Phyton programming languages. Besides introducing the platform, chapter 2 presented the theory behind the MC and SC techniques and the OFDM and SC-FDMA transmissions.

In chapter 3, we implemented basic examples using the GRC, *gr_modtools* and *gr_filter_design* tools, leading to a better understating of the GNURadio platform and its potentials.

The GNURadio platform already came with the OFDM transmission blocks implemented. In order to create the SC-FDMA blocks, we carefully examined which functions each module did inside the OFDM blocks and redesigned them to create the new SC-FDMA blocks. In the modulator, the modules that received modifications were: *pkt_input*,

which now outputs only the data without mapping in the occupied tones and *preamble*, where we used a new preamble sequence, the Zadoff-Chu sequence. In the demodulator, the modules that received modifications were: *scfdma_sync* and *scfdma_frame_acq*, where we created a new algorithm to find the occupied tones and used the new preamble sequence. Besides redesigning some modules, we added new ones, such as: *fft*, *scale_vec* and *mapper* in the modulator and *ifft_demod*, *scale* and a new filter in the demodulator.

After creating the new SC-FDMA blocks, we modified the *benchmark_rx.py* and *benchmark_tx.py* files in order to get the SC-FDMA transmission to work in the USRP devices.

In chapter 4, we tested the two transmission techniques using different environments. In the GRC we simulated the modulators using the Channel Model block. In a noise and frequency offset free environment, both modulators hold up and get the final data without any errors, except for a small distortion due to the synchronization and other operations. When the noise power was increased, we saw that both transmissions could achieve the same results, until we reach a point where the noise voltage distorted the samples too much. Using a low noise power, we tested the effect of the frequency offset for a value bellow or higher than the size of one sub-carrier. Both modulators lose the first preamble block and its data blocks, but after the second preamble block stabilized. From all the tests, when comparing the final constellation for the same data block in the two transmission techniques, we saw that the SC-FDMA seems to achieve better results.

Finally, we tested the transmission techniques using the USRP hardware and the benchmarks, first with the loop-back cable then with the antennas. In the first tests, we saw that the OFDM transmission had better results in a lower frequency (1 GHz) achieving a lower PER (3 %), but as we increase the frequency (to 2.5 GHz), the PER was much higher (to 96%). In contrast, the SC-FDMA transmission had a higher PER (80.4%) in the initial frequency but it almost did not decrease while the frequency increased. With the antennas we got similar results: with a frequency of 2.48 GHz, the OFDM transmission got a PER of 91.1% and the SC-FDMA transmission got 22.1%. Although, these results were achieved in the selected frequencies and initial parameters, the SC-FDMA implementation shown to have an acceptable performance when compared to the OFDM modulation.

As a final analysis, it is possible to tell that the GNURadio platform, an open source

software, and the USRP devices, a low budget platform with high potential, provide a great tool to develop and test new transmission techniques, study other signal processing functions and implement prototypes to get real results measured in the environment. Also, the main objective purposed by this study, implement the SC-FDMA transmission technique was successfully executed and the tests performed with it were successful.

5.2 Future Work

The SC-FDMA transmission modules implemented still have some problems with the synchronization when they are used in a the real channel. They need improvements in order to be more accurate, such as using new types of preambles and other synchronization methods. The implemented transmission needs more tests in the USRP to find the optimal configuration using different parameters, such as: center frequencies, FFT length, occupied tones, cyclic prefix, constellation, bandwidth, filter design and preambles. Also, other results could be measured, like the Bit Error Rate and a better measurement of the SNR could be obtained using an alternative SNR estimator.

For testing purposes, it would be interesting to implement a system that could obtain the data files in a way that it would be easy to manipulate in other programs like MatLab.

Besides improving the SC-FDMA and the GNURadio platform, new ideas could be implemented to improve the overall study in the telecommunication area, such as:

- Implementing the Single-Carrie Frequency-Domain-Equalization transmission technique, by removing the FFT modules from the modulator and redesigning some modules. Eventually implementing a module using the IB-DFE transmission.
- Implement systems that work in other layers and use the GNURadio and the USRP as the physical layer.
- Study the implementation in the encoder and decoders for the error corrections.

The GNURadio software is an open source with a lot of potential and the USRP devices are low budget giving a great tool in the academic environments to develop new ideas.

Bibliography

[3GP06] 3GPP. 3rd generation partnership project; technical specification group radio access network; physical layer aspects for evolved universal terrestrial radio access. *3GPP TR 25.814*, Set. 2006.

[3GP08] 3GPP. Evolved universal terrestrial radio access (e-utra) and evolved universal terrestrial radio access (e-utran); overall description. *ETSI TS 136 300*, Apr. 2008.

[Ahs09] Borko Furht; Syed A. Ahson. *Long Term Evolution: 3GPP LTE Radio and Cellular Technology.* Auerbach Publications, 2009.

[AM09] Norsheila Fisal Sharifah Kamilah Syed Yusof Rozeha A.Rashid Arief Marwanto, Mohd Adib Sarijari. Experimental study of ofdm implementation utilizing gnu radio and usrp - sdr. *Malaysia International Conference on Communications*, 12(2):15–17, 2009.

[Bib04] Ieee standard for local and metropolitan area networks part 16: Air interface for fixed broadband wireless access systems. *IEEE 802.16-2004*, Oct. 2004.

[CS10] C. Ciochina and H. Sari. A review of ofdma and single-carrier fdma and some recent results. *Advances In Electronics And Telecommunications*, 1(1):35 – 40, Apr. 2010.

[Ham08] Firas Abbas Hamza. The usrp under 1.5x magnifying lens! *Proc. IEEE CCNC 2007*, pages 5–10, June 2008.

[Han09] Burcu Hanta. Sc-fdma and lte uplink physical layer design. *Ausgewahlte Kapitel der Nachrichtentechnik, WS 2009/2010*, Dec. 2009.

[HGMG06] Junsung Lim Hyung G. Myung and David J. Goodman. Single carrier fdma for uplink wireless transmission. *IEEE VEHICULAR TECHNOLOGY MAGAZINE*, (1556-6072/06 IEEE):30 – 38, September 2006.

[HSJ94] G. Karam H. Sari and I. Jeanclaude. Channel equalization and carrier synchronization in ofdm systems. *in Int. Tirrenia Workshop on Digital Communications, Tirrenia, Italy, Sep. 1993, in Audio and Video Digital Radio Broadcasting Systems and Techniques, Elsevier Science Publishers*, 1994.

[IEE06] IEEE. Part 16: Air interface for fixed and mobile broadband wireless access systems amendment 2: Physical and medium access control layers for combined fixed and mobile operation in licensed bands and corrigendum 1. *IEEE 802.16-2005*, Feb. 2006.

[LAMRdTM08] Simone Frattasi Luis Angel Maestro Ruiz de Temino, Gilberto Berardinelli and Preben Mogensen. Channel-aware scheduling algorithms for sc-fdma in lte uplink. *978-1-4244-2644-7/08 IEEE*, pages 1 – 6, Set. 2008.

[Mar09] Majó Marcos. Design and implementation of an ofdm-based communication system for the gnu radio platform, 2009.

[Mit95] Joe Mitola. The software radio architecture. *0163-6804/95 IEEE Communication Magazine*, May, 1995.

[NL08] Hazem H. Refai Nick LaSorte, W. Justin Barnes. The history of orthogonal frequency division multiplexing. *978-1-4244-2324-8 IEEE*, 2008.

[Pag06] C. Pagès. A Vectorization of Synchronization Algorithms for OFDM Systems. Master's thesis, Technische Universitat Dresden - Fakultat Elektrotechnik und Informationstechnik, 2006.

[Ram08] Vijaya Chandran Ramasami. Orthogonal frequency division multiplexing. *ETSI TS 136 300*, Apr. 2008.

[Res13] Ettus Research. USRP. USRP from https://www.ettus.com/, 2013.

[Rum08] Moray Rumney. 3gpp lte: Introducing single-carrier fdma. *Agilent Measurement Journal*, Jan. 2008.

[SB09] C. Leung S. Beyme. Efficient computation of dft of zadoff-chu sequences. *IEEE TRANSACTIONS ON COMMUNICATIONS*, pages 1613 – 1621, Apr. 2009.

[SC97] T. M. Schmidl and D. C. Cox. Robust frequency and timing synchronization for ofdm. *IEEE Transactions on Communications*, 45:1613 – 1621, Dec. 1997.

[ser09] European Standard (Telecommunications series). Digital video broadcasting (dvb); framing structure, channel coding and modulation for digital terrestrial television. *ETSI Standard: EN 300 744*, Jan. 2009.

[Sil10] Fabio J. Silva. Design and Performance Evaluation of Turbo FDE Receivers. Master's thesis, FCT-UNL, 2010.

[Tec11] Agilent Technologies. Introducing lte-advanced. March 2011.

[vdBMSPOB97] Jan-Jaap van de Beek; Magnus Sandell; Per Ola Borjesson. Ml estimation of time and frequency offset in ofdm systems. *IEEE TRANSACTIONS ON SIGNAL PROCESSING*, pages 1880–1885, Jul. 1997.

Appendices

Appendix A

Dial Tone Example

This appendix shows the code generated in Dial Tone example (figure A.1). This example creates an audio signal by adding two cosine waves and a noise signal. The first lines define the libraries needed to run the application. From the 19^{tb} line, comes the definition of the *dial_tone* class, which has several modules and variables associated. From the 115^{tb} line, the modules are connected, using the function *connect*. In the last part, the class defines the links between the variables and the appropriated modules.

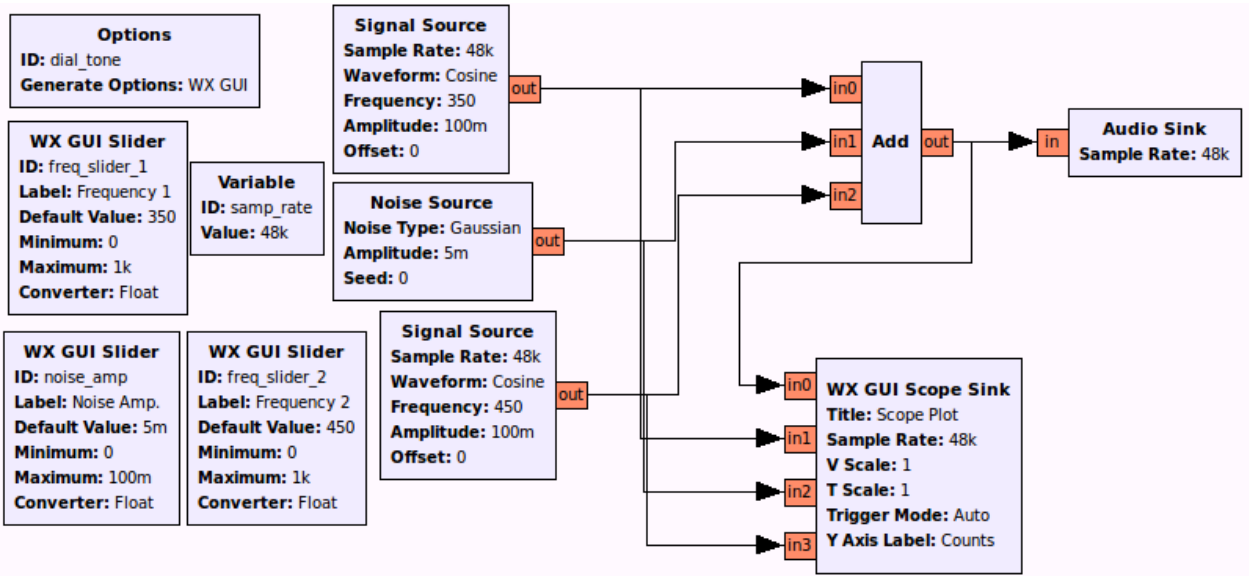


Figure A.1: Dial Tone example - block diagram

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Dial Tone
# Generated: Tue Nov 12 11:41:36 2013
##################################################

from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from gnuradio.wxgui import forms
from gnuradio.wxgui import scopesink2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class dial_tone(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Dial Tone")
        _icon_path = "/usr/share/icons/hicolor/32x32/apps/gnuradio-grc.png"
        self.SetIcon(wx.Icon(_icon_path, wx.BITMAP_TYPE_ANY))

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 48000
        self.noise_amp = noise_amp = .005
        self.freq_slider_2 = freq_slider_2 = 450
        self.freq_slider_1 = freq_slider_1 = 350

        ##################################################
        # Blocks
        ##################################################
```

```
38          _noise_amp_sizer = wx.BoxSizer(wx.VERTICAL)
39          self._noise_amp_text_box = forms.text_box(parent=self.GetWin(),
40              sizer=_noise_amp_sizer,value=self.noise_amp,
41              callback=self.set_noise_amp,label="Noise Amp.",
42              converter=forms.float_converter(),proportion=0,)
43
44          self._noise_amp_slider = forms.slider(parent=self.GetWin(),
45              sizer=_noise_amp_sizer, value=self.noise_amp,
46              callback=self.set_noise_amp, minimum=0, maximum=.1,
47              num_steps=1000, style=wx.SL_HORIZONTAL, cast=float,proportion=1,)
48
49          self.Add(_noise_amp_sizer)_freq_slider_2_sizer = wx.BoxSizer(wx.VERTICAL)
50
51          self._freq_slider_2_text_box = forms.text_box(
52              parent=self.GetWin(),sizer=_freq_slider_2_sizer,
53              value=self.freq_slider_2,callback=self.set_freq_slider_2,
54              label="Frequency 2",converter=forms.float_converter(),proportion=0,)
55
56          self._freq_slider_2_slider = forms.slider(parent=self.GetWin(),
57              sizer=_freq_slider_2_sizer,value=self.freq_slider_2,
58              callback=self.set_freq_slider_2,minimum=0,
59              maximum=1000,num_steps=1000,style=wx.SL_HORIZONTAL,
60              cast=float,proportion=1,)
61
62          self.Add(_freq_slider_2_sizer)_freq_slider_1_sizer = wx.BoxSizer(wx.VERTICAL)
63          self._freq_slider_1_text_box = forms.text_box(parent=self.GetWin(),
64              sizer=_freq_slider_1_sizer,value=self.freq_slider_1,
65              callback=self.set_freq_slider_1,label="Frequency 1",
66              converter=forms.float_converter(),proportion=0,)
67
68          self._freq_slider_1_slider = forms.slider(parent=self.GetWin(),
69              sizer=_freq_slider_1_sizer,value=self.freq_slider_1,
70              callback=self.set_freq_slider_1,minimum=0,aximum=1000,
71              num_steps=1000, style=wx.SL_HORIZONTAL, cast=float,proportion=1,)
72
73          self.Add(_freq_slider_1_sizer)
74
```

```
75          self.wxgui_scopesink2_0 = scopesink2.scope_sink_f(
76              self.GetWin(),title="Scope Plot",sample_rate=samp_rate,
77              v_scale=1,v_offset=0,t_scale=1, ac_couple=False,
78              xy_mode=False,num_inputs=4,trig_mode=gr.gr_TRIG_MODE_AUTO,
79              y_axis_label="Counts",)
80
81          self.Add(self.wxgui_scopesink2_0.win)
82          self.gr_add_xx_0 = gr.add_vff(1)
83          self.blocks_throttle_0 = blocks.throttle(gr.sizeof_float*1, samp_rate)
84          self.analog_sig_source_x_1 = analog.sig_source_f(samp_rate, analog.GR_COS_WAVE, freq_slider_2,
                .1, 0)
85          self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_COS_WAVE, freq_slider_1,
                .1, 0)
86          self.analog_noise_source_x_0 = analog.noise_source_f(analog.GR_GAUSSIAN, noise_amp, 0)
87
88          ##################################################
89          # Connections
90          ##################################################
91          self.connect((self.analog_noise_source_x_0, 0), (self.gr_add_xx_0, 1))
92          self.connect((self.analog_noise_source_x_0, 0), (self.wxgui_scopesink2_0, 2))
93          self.connect((self.analog_sig_source_x_1, 0), (self.wxgui_scopesink2_0, 3))
94          self.connect((self.analog_sig_source_x_0, 0), (self.wxgui_scopesink2_0, 1))
95          self.connect((self.analog_sig_source_x_1, 0), (self.gr_add_xx_0, 2))
96          self.connect((self.analog_sig_source_x_0, 0), (self.gr_add_xx_0, 0))
97          self.connect((self.gr_add_xx_0, 0), (self.blocks_throttle_0, 0))
98          self.connect((self.blocks_throttle_0, 0), (self.wxgui_scopesink2_0, 0))
99
100
101     def get_samp_rate(self):
102         return self.samp_rate
103
104     def set_samp_rate(self, samp_rate):
105         self.samp_rate = samp_rate
106         self.analog_sig_source_x_1.set_sampling_freq(self.samp_rate)
107         self.wxgui_scopesink2_0.set_sample_rate(self.samp_rate)
108         self.blocks_throttle_0.set_sample_rate(self.samp_rate)
109         self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)
```

```
110
111     def get_noise_amp(self):
112         return self.noise_amp
113
114     def set_noise_amp(self, noise_amp):
115         self.noise_amp = noise_amp
116         self._noise_amp_slider.set_value(self.noise_amp)
117         self._noise_amp_text_box.set_value(self.noise_amp)
118         self.analog_noise_source_x_0.set_amplitude(self.noise_amp)
119
120     def get_freq_slider_2(self):
121         return self.freq_slider_2
122
123     def set_freq_slider_2(self, freq_slider_2):
124         self.freq_slider_2 = freq_slider_2
125         self.analog_sig_source_x_1.set_frequency(self.freq_slider_2)
126         self._freq_slider_2_slider.set_value(self.freq_slider_2)
127         self._freq_slider_2_text_box.set_value(self.freq_slider_2)
128
129     def get_freq_slider_1(self):
130         return self.freq_slider_1
131
132     def set_freq_slider_1(self, freq_slider_1):
133         self.freq_slider_1 = freq_slider_1
134         self._freq_slider_1_slider.set_value(self.freq_slider_1)
135         self._freq_slider_1_text_box.set_value(self.freq_slider_1)
136         self.analog_sig_source_x_0.set_frequency(self.freq_slider_1)
137
138 if __name__ == '__main__':
139     parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
140     (options, args) = parser.parse_args()
141     tb = dial_tone()
142     tb.Run(True)
```

Listing A.1: Dial tone python code.

Appendix B

Block Example Code

Section 3.2.1 describes how to create a block using the tool *gr_modtool*. To complement that section, this appendix shows the relevant files generated by the tool and covers the modifications done to them, in order to create a block that computes the square of the inputted signal. In last part, this appendix shows the tests that the block was submitted.

First, let us describe the modification performed in the file *square_ff_impl.cc*. There are several functions inside this file, but as we know, the main function is *general_work*. Here, the inputted items, pointed by the **in* pointer, multiply with themselves and are outputted, pointed by the **out* pointer. Sometimes the header file needs to be adapted, but in this example *square_ff_impl.h* did not needed any change.

```
1  int square_ff_impl::general_work (int noutput_items,
2                                    gr_vector_int &ninput_items,
3                                    gr_vector_const_void_star &input_items,
4                                    gr_vector_void_star &output_items)
5  {
6
7      const float *in = (const float *) input_items[0];
8      float *out = (float *) output_items[0];
9
10     for(int i = 0; i < noutput_items; i++) {
11         out[i] = in[i] * in[i];
12     }
13
14     // Tell runtime system how many input items we consumed on
```

```
15     // each input stream.
16     consume_each (noutput_items);
17
18     // Tell runtime system how many output items we produced.
19     return noutput_items;
20 }
```

Listing B.1: *Square_ff_impl.cc - general_work.*

The other file changed was *exp_square_ff.xml*. This file links the source files and the GRC application, and is automatically created by *gr_modtool*. The only change made was in the $< sink >< /sink >$ and $< source >< /source >$ fields, where we declare the sink and source ports.

```
1  <?xml version="1.0"?>
2  <block>
3    <name>square_ff</name>
4    <key>exp_square_ff</key>
5    <category>exp</category>
6    <import>import exp</import>
7    <make>exp.square_ff()</make>
8    <sink>
9      <name>in</name>
10     <type>float</type>
11   </sink>
12   <source>
13     <name>out</name>
14     <type>float</type>
15   </source>
16 </block>
```

Listing B.2: XML code for block example.

After compiling the code, we test it in GRC. The test consists in squaring a cosine wave and show the result in a WX GUI Scope Sink block. Figure B.1 shows the block diagram and figure B.2 shows the resulted graphs, where channel 1 is the cosine wave and channel 2 the its square.

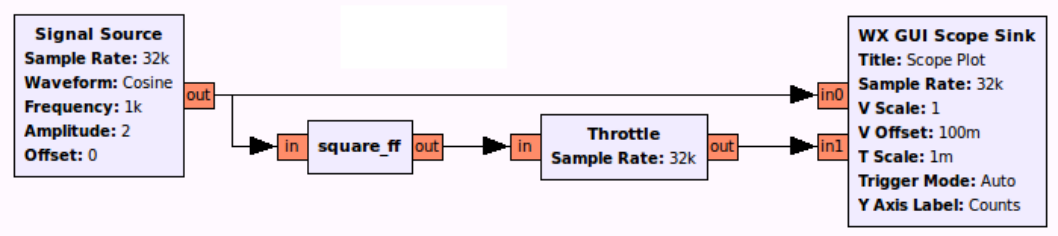


Figure B.1: Test diagram using the new block.

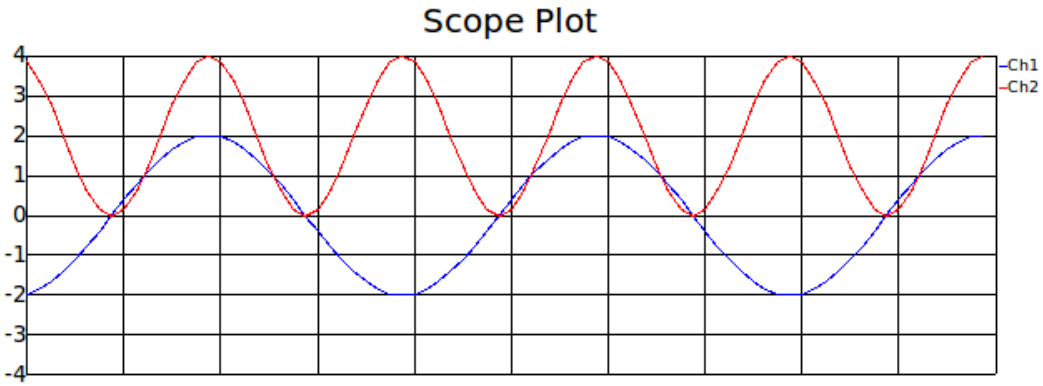


Figure B.2: Cosine wave and its square.

Appendix C

OFDM Block Code

Section 3.3 describes the OFDM modulator and demodulator blocks. This appendix shows the main functions and code lines that are referred in the description.

The first file mention is *ofdm.py*. This file consists in two classes (*ofdm_mod* and *ofdm_demod*) and a vector named *known_symbols_4512_3*, list C.1 shows the two classes and the vector initialization.

```
(....)
# ////////////////////////////////////////////////////////////////////////////
# mod/demod with packets as i/o
# ////////////////////////////////////////////////////////////////////////////
class ofdm_mod(gr.hier_block2):
    """
    Modulates an OFDM stream. Based on the options fft_length, occupied_tones, and
    cp_length, this block creates OFDM symbols using a specified modulation option.
    """
(....)
class ofdm_demod(gr.hier_block2):
    """
    Demodulates a received OFDM stream. Based on the options fft_length, occupied_tones, and
    cp_length, this block performs synchronization, FFT, and demodulation of incoming OFDM
    symbols and passes packets up the a higher layer.
    """
(....)
# Generating known symbols with:
# i = [2*random.randint(0,1)-1 for i in range(4512)]
```

```
20 known_symbols_4512_3 = [−1, −1, 1, −1, 1, 1, −1, −1, 1, −1, 1, 1, −1, 1, −1, −1, 1,(....)]
```

Listing C.1: *Ofdm.py* classes and vector.

C.1 OFDM Modulator Block

In the modulator block, one of the key features is the message queue defined in the function *send_pkt*. This function uses *ofdm_packet_utils.make_packet* to make packets and stores them in the queue.

```
1  def send_pkt(self, payload='', eof=False):
2          """
3          Send the payload.
4
5          @param payload: data to send
6          @type payload: string
7          """
8          if eof:
9              msg = gr.message(1) # tell self._pkt_input we're not sending any more packets
10         else:
11             # print "original_payload =", string_to_hex_list(payload)
12             pkt = digital.ofdm_packet_utils.make_packet(payload, 1, 1,
13                                                         self._pad_for_usrp,
14                                                         whitening=True)
15
16             #print "pkt =", string_to_hex_list(pkt)
17             msg = gr.message_from_string(pkt)
18         self._pkt_input.msgq().insert_tail(msg)
```

Listing C.2: *Send_pkt* function code.

List C.3 shows which modules are created and the connections between them, in order to create the modulator block.

```
1  self._pkt_input = ofdmfic_swig.ofdmfic_mapper_bcv(rotated_const,
2                                                    msgq_limit,
3                                                    options.occupied_tones,
4                                                    options.fft_length)
```

```
5
6  self.preambles = digital.ofdm_insert_preamble(self._fft_length,
7                                                 padded_preambles)
8
9  self.ifft = gr.fft_vcc(self._fft_length, False, win, True)
10 self.cp_adder = digital.ofdm_cyclic_prefixer(self._fft_length,
11                                                symbol_length)
12 self.scale = gr.multiply_const_cc(1.0 / math.sqrt(self._fft_length))
13
14 self.connect((self._pkt_input, 0),(self.preambles, 0))
15 self.connect((self._pkt_input, 1),(self.preambles, 1))
16 self.connect(self.preambles,self.ifft, self.cp_adder, self.scale,self)
```

Listing C.3: Module creation and connections.

During the OFDM modulator description, there are some important code lines to point out. In the *digital_ofdm_mapper_bcv.cc* file, the first part is the method to find the occupied tones and it is implemented in the constructor, and the second part is the main algorithm to fill the data frames, located in the *work* method.

```
1  (....)
2
3  digital_ofdm_mapper_bcv::digital_ofdm_mapper_bcv(
4          const std::vector<gr_complex> &constellation, unsigned int msgq_limit,
5          unsigned int occupied_carriers, unsigned int fft_length) :
6          gr_sync_block("ofdm_mapper_bcv", gr_make_io_signature(0, 0, 0),
7                  gr_make_io_signature2(1, 2, sizeof(gr_complex) * fft_length*2,
8                          sizeof(char))), d_constellation(constellation), d_msgq(
9                  gr_make_msg_queue(msgq_limit)), d_msg_offset(0), d_eof(false), d_occupied_carriers(
10                 occupied_carriers), d_fft_length(fft_length), d_bit_offset(0), d_pending_flag(
11                 0), d_resid(0), d_nresid(0) {
12 (....)
13     // find out how many zeros to pad on the sides; the difference between the fft length and the
           subcarrier
14     // mapping size in chunks of four. This is the number to pack on the left and this number plus any
15     // residual nulls (if odd) will be packed on the right.
16     diff = (d_fft_length / 4 - carriers.length()) / 2;
17
```

```
18    unsigned int i, j, k;
19    for (i = 0; i < carriers.length(); i++) {
20        char c = carriers[i]; // get the current hex character from the string
21        for (j = 0; j < 4; j++) { // walk through all four bits
22            k = (strtol(&c, NULL, 16) >> (3 - j)) & 0x1; // convert to int and extract next bit
23            if (k) { // if bit is a 1,
24                d_subcarrier_map.push_back(4 * (i + diff) + j); // use this subcarrier
25            }
26        }
27    }
28
29 (....)
30 }
31 int digital_ofdm_mapper_bcv::work(int noutput_items,
32        gr_vector_const_void_star &input_items,
33        gr_vector_void_star &output_items) {
34 (....)
35
36 // need new data to process
37        if (d_bit_offset == 0) {
38            d_msgbytes = d_msg->msg()[d_msg_offset];
39            //printf("mod message byte: %x\n", d_msgbytes);
40        }
41        if (d_nresid > 0) {
42            // take the residual bits, fill out nbits with info from the new byte, and put them in the symbol
43            d_resid |= (((1 << d_nresid) - 1) & d_msgbytes)
44                    << (d_nbits - d_nresid);
45            bits = d_resid;
46
47            out[d_subcarrier_map[i]] = d_constellation[bits];
48            i++;
49
50            d_bit_offset += d_nresid;
51            d_nresid = 0;
52            d_resid = 0;
53            //printf("mod bit(r): %x resid: %x nresid: %d bit_offset: %d\n",
54            // bits, d_resid, d_nresid, d_bit_offset);
```

```
55        } else {
56            if ((8 - d_bit_offset) >= d_nbits) { // test to make sure we can fit nbits
57                // take the nbits number of bits at a time from the byte to add to the symbol
58                bits = ((1 << d_nbits) - 1) & (d_msgbytes >> d_bit_offset);
59                d_bit_offset += d_nbits;
60
61                out[d_subcarrier_map[i]] = d_constellation[bits];
62                i++;
63            } else { // if we can't fit nbits, store them for the next
64                // saves d_nresid bits of this message where d_nresid < d_nbits
65                unsigned int extra = 8 - d_bit_offset;
66                d_resid = ((1 << extra) - 1) & (d_msgbytes >> d_bit_offset);
67                d_bit_offset += extra;
68                d_nresid = d_nbits - extra;
69            }
70
71        }
72 (....)
73
74        while (i < d_subcarrier_map.size()) { // finish filling out the symbol
75            out[d_subcarrier_map[i]] = d_constellation[randsym()];
76
77            i++;
78        }
79
80 (....)
81
82    return 1; // produced symbol
83 }
```

Listing C.4: *Digital_ofdm_mapper_bcv.cc* main code lines.

Now, the modulator needs to insert the preambles, this way the demodulator can do the synchronization and the equalization. List C.5 shows the *general_work* method, in the file *digital_ofdm_insert_preamble.cc*, this method finds the right moment, based on the characters inputted, and inserts the preamble frame.

```
1 int
```

```
2  digital_ofdm_insert_preamble::general_work(int noutput_items,
3                          gr_vector_int &ninput_items_v,
4                          gr_vector_const_void_star &input_items,
5                          gr_vector_void_star &output_items)
6  {
7
8  (....)
9
10   while (no < noutput_items && ni < ninput_items){
11     switch(d_state){
12     case ST_IDLE:// this is first symbol of new payload
13    (....)
14       break;
15
16     case ST_PREAMBLE:
17     (....)
18       break;
19
20     case ST_FIRST_PAYLOAD:
21       // copy first payload symbol from input to output
22       memcpy(&out_sym[no * d_fft_length],
23         &in_sym[ni * d_fft_length],
24         d_fft_length * sizeof(gr_complex));
25       (....)
26       break;
27
28     case ST_PAYLOAD:
29    (....)
30       // copy a symbol from input to output
31       memcpy(&out_sym[no * d_fft_length],
32         &in_sym[ni * d_fft_length],
33         d_fft_length * sizeof(gr_complex));
34    (....)
35       break;
36
37     default:
38       std::cerr << "digital_ofdm_insert_preamble: (can't happen) invalid state, resetting\n";
```

```
39          enter_idle();
40        }
41      }
42
43      consume_each(ni);
44      return no;
45  }
```

Listing C.5: *Digital_ofdm_insert_preamble.cc* main code lines.

The last method is related to the cyclic prefix adder, this module is located in the *digital_ofdm_cyclic_prefixer.cc* file and list C.6 shows the *work* method.

```
1  int
2  digital_ofdm_cyclic_prefixer::work (int noutput_items,
3                      gr_vector_const_void_star &input_items,
4                      gr_vector_void_star &output_items)
5  {
6    gr_complex *in = (gr_complex *) input_items[0];
7    gr_complex *out = (gr_complex *) output_items[0];
8    size_t cp_size = d_output_size - d_input_size;
9    unsigned int i=0, j=0;
10
11   j = cp_size;
12   for(i=0; i < d_input_size; i++,j++) {
13     out[j] = in[i];
14   }
15
16   j = d_input_size - cp_size;
17   for(i=0; i < cp_size; i++, j++) {
18     out[i] = in[j];
19   }
20
21   return d_output_size;
22 }
```

Listing C.6: *Digital_ofdm_cyclic_prefixer.cc work* method.

C.2 OFDM Demodulator Block

The OFDM demodulator block is defined in the *ofdm_demod* class, also located in the *ofdm.py* file. This class calls two modules: the *ofdm_recv* module, that executes the filtering, synchronization, sampling and equalization processes; and the *ofdm_demod* module that demodulates the symbols in bits and rebuilds the messages received. List C.7 shows the *ofdm_recv* and *ofdm_demod* modules creation, and the connections between them.

```
class ofdm_demod(gr.hier_block2):
    (....)
        self.ofdm_recv = ofdm_receiver(self._fft_length,
                                       self._cp_length,
                                       self._occupied_tones,
                                       self._snr, preambles,
                                       options.log)

    (....)
    self.ofdm_demod = digital_swig.ofdm_frame_sink(rotated_const, range(arity),
                                                   self._rcvd_pktq,
                                                   self._occupied_tones,
                                                   phgain, frgain)

        self.connect(self, self.ofdm_recv)
        self.connect((self.ofdm_recv, 0), (self.ofdm_demod, 0))
        self.connect((self.ofdm_recv, 1), (self.ofdm_demod, 1))
    (....)
```

Listing C.7: OFDM demodulator block creation and the connections.

The module *ofdm_recv* is built in another python file called *ofdm_receiver.py*. List C.8 shows the main code lines within this file. The first part is the filter implementation, the second is the selection for the synchronization module, where the pseudorandom noise synchronization is selected by default, then comes the creation of the remaining modules and last the connections between them.

```
class ofdm_receiver(gr.hier_block2):
    (....)
```

```
3         bw = (float(occupied_tones) / float(fft_length)) / 2.0
4         tb = bw*0.08
5         chan_coeffs = gr.firdes.low_pass (1.0, # gain
6                                           1.0, # sampling rate
7                                           bw+tb, # midpoint of trans. band
8                                           tb, # width of trans. band
9                                           gr.firdes.WIN_HAMMING) # filter type
10        self.chan_filt = gr.fft_filter_ccc(1, chan_coeffs)# Set up blocks
11
12    (....)
13        SYNC = "pn"
14        if SYNC == "ml":
15            nco_sensitivity = -1.0/fft_length # correct for fine frequency
16            self.ofdm_sync = ofdm_sync_ml(fft_length,
17                                          cp_length,
18                                          snr,
19                                          ks0time,
20                                          logging)
21        elif SYNC == "pn":
22            nco_sensitivity = -2.0/fft_length # correct for fine frequency
23            self.ofdm_sync = ofdm_sync_pn(fft_length,
24                                          cp_length,
25                                          logging)
26    (....)
27        self.nco = gr.frequency_modulator_fc(nco_sensitivity) # generate a signal proportional to
              frequency error of sync block
28        self.sigmix = gr.multiply_cc()
29        self.sampler = digital_swig.ofdm_sampler(fft_length, fft_length+cp_length)
30        self.fft_demod = gr.fft_vcc(fft_length, True, win, True)
31        self.ofdm_frame_acq = digital_swig.ofdm_frame_acquisition(occupied_tones,
32                                                                  fft_length,
33                                                                  cp_length, ks[0])
34
35        self.connect(self, self.chan_filt) # filter the input channel
36        self.connect(self.chan_filt, self.ofdm_sync) # into the synchronization alg.
37        self.connect((self.ofdm_sync,0), self.nco, (self.sigmix,1)) # use sync freq. offset output to
              derotate input signal
```

```
38          self.connect(self.chan_filt, (self.sigmix,0)) # signal to be derotated
39          self.connect(self.sigmix, (self.sampler,0)) # sample off timing signal detected in sync alg
40          self.connect((self.ofdm_sync,1), (self.sampler,1)) # timing signal to sample at
41
42          self.connect((self.sampler,0), self.fft_demod) # send derotated sampled signal to FFT
43          self.connect(self.fft_demod, (self.ofdm_frame_acq,0)) # find frame start and equalize signal
44          self.connect((self.sampler,1), (self.ofdm_frame_acq,1)) # send timing signal to signal frame start
45          self.connect((self.ofdm_frame_acq,0), (self,0)) # finished with fine/coarse freq correction,
46          self.connect((self.ofdm_frame_acq,1), (self,1)) # frame and symbol timing, and equalization
```

Listing C.8: *Ofdm_receiver.py* main code.

Inside *ofdm_recv* there are several modules, but the main module that needs description is the *ofdm_frame_acq*. This module does the equalization of the data frames. In the method *general_work*, when the module finds the character 1 in the *signal_in* variable, the module uses the preamble frame and estimates the shift between the carriers in the *correlate* function; and calculates the one-tap equalization in the *calculate_equalizer* function. When the next frames enter, the symbols in them are multiplied with the one-tap equalization and shifted the necessary carriers, until the module finds the next character 1. List C.9 shows the main code for the *ofdm_frame_acq* module.

```
1  void digital_ofdm_frame_acquisition::correlate(const gr_complex *symbol,
2          int zeros_on_left) {
3      (....)
4      // sweep through all possible/allowed frequency offsets and select the best
5      int index = 0;
6      float max = 0, sum = 0;
7      for (i = zeros_on_left - d_freq_shift_len;i < zeros_on_left + d_freq_shift_len; i++) {
8          sum = 0;
9          for (j = 0; j < d_occupied_carriers; j++) {
10             sum += (d_known_phase_diff[j] * d_symbol_phase_diff[i + j]);
11         }
12         if (sum > max) {
13             max = sum;
14             index = i;
15         }
16     }
```

```
17
18      // set the coarse frequency offset relative to the edge of the occupied tones
19      d_coarse_freq = index - zeros_on_left;
20  }
21
22  void digital_ofdm_frame_acquisition::calculate_equalizer(
23          const gr_complex *symbol, int zeros_on_left) {
24      unsigned int i = 0;
25      d_hestimate[0] = d_known_symbol[0]
26              / (coarse_freq_comp(d_coarse_freq, 1)
27                      * symbol[zeros_on_left + d_coarse_freq]);
28      for (i = 2; i < d_occupied_carriers; i += 2) {
29          d_hestimate[i] = d_known_symbol[i]
30                  / (coarse_freq_comp(d_coarse_freq, 1)
31                          * (symbol[i + zeros_on_left + d_coarse_freq]));
32          d_hestimate[i - 1] = (d_hestimate[i] + d_hestimate[i - 2])
33                  / gr_complex(2.0, 0.0);
34      }
35       (....)
36  }
37
38  int digital_ofdm_frame_acquisition::general_work(int noutput_items,
39          gr_vector_int &ninput_items, gr_vector_const_void_star &input_items,
40          gr_vector_void_star &output_items) {
41      const gr_complex *symbol = (const gr_complex *) input_items[0];
42      const char *signal_in = (const char *) input_items[1];
43
44      (....)
45      if (signal_in[0]) {
46          d_phase_count = 1;
47          correlate(symbol, zeros_on_left);
48          calculate_equalizer(symbol, zeros_on_left);
49          signal_out[0] = 1;
50      } else {
51          signal_out[0] = 0;
52      }
53
```

```
54    for (unsigned int i = 0; i < d_occupied_carriers; i++) {
55        out[i] = d_hestimate[i] * coarse_freq_comp(d_coarse_freq, d_phase_count)
56                * symbol[i + zeros_on_left + d_coarse_freq];
57    }
58
59    (....)
60    return 1;
61 }
```

Listing C.9: *Self.ofdm_frame_acq* block main code.

The last module to be revised is the *ofdm_demod*. This module receives the OFDM data frames and demodulates the symbols, using the constellation in use. The *work* method has the functions described in chapter 3.3.2, they work like a state machine and they are in listed in C.10. The main state is the *STATE_HAVE_HEADER* and it is here that the function *demapper* is called and demodulates the symbols.

```
1  unsigned int digital_ofdm_frame_sink::demapper(const gr_complex *in,
2          unsigned char *out) {
3      (....)
4      //while(i < d_occupied_carriers) {
5      while (i < d_subcarrier_map.size()) {
6          (....)
7          while ((d_byte_offset < 8) && (i < d_subcarrier_map.size())) {
8              (....)
9              if ((8 - d_byte_offset) >= d_nbits) {
10                 d_partial_byte |= bits << (d_byte_offset);
11                 d_byte_offset += d_nbits;
12             } else {
13                 d_nresid = d_nbits - (8 - d_byte_offset);
14                 int mask = ((1 << (8 - d_byte_offset)) - 1);
15                 d_partial_byte |= (bits & mask) << d_byte_offset;
16                 d_resid = bits >> (8 - d_byte_offset);
17                 d_byte_offset += (d_nbits - d_nresid);
18             }
19             printf("demod symbol: %.4f + j%.4f bits: %x partial_byte: %x byte_offset: %d resid: %x nresid:
                       %d\n",
20                     in[i-1].real(), in[i-1].imag(), bits, d_partial_byte, d_byte_offset, d_resid, d_nresid);
```

```
21          }
22          if (d_byte_offset == 8) {
23              //printf("demod byte: %x \n\n", d_partial_byte);
24              out[bytes_produced++] = d_partial_byte;
25              d_byte_offset = 0;
26              d_partial_byte = 0;
27          }
28      }
29      (....)
30  }
31
32  int digital_ofdm_frame_sink::work(int noutput_items,
33          gr_vector_const_void_star &input_items,
34          gr_vector_void_star &output_items) {
35      (....)
36      switch (d_state) {
37      case STATE_SYNC_SEARCH: // Look for flag indicating beginning of pkt
38          (....)
39          break;
40      case STATE_HAVE_SYNC:
41          // only demod after getting the preamble signal; otherwise, the
42          // equalizer taps will screw with the PLL performance
43          bytes = demapper(&in[0], d_bytes_out);
44          (....)
45          break;
46      case STATE_HAVE_HEADER:
47          bytes = demapper(&in[0], d_bytes_out);
48          (....)
49          break;
50      default:
51          assert(0);
52      } // switch
53      return 1;
54  }
```

Listing C.10: *Self.ofdm_frame_acq* block main code.

Appendix D

SC-FDMA Block Code

Section 3.4 focus on the SC-FDMA modulator and demodulator blocks, to assist the description done in that section, this appendix shows the main functions and code lines that are referred in the description.

One of the first file mention is the python file *scfdma.py*, this file consists in two classes (*scfdma_mod* and *scfdma_demod*), and list D.1 shows them.

```
(....)
# ////////////////////////////////////////////////////////////////////////////
# mod/demod with packets as i/o
# ////////////////////////////////////////////////////////////////////////////
class scfdma_mod(gr.hier_block2):
    """
    Modulates an SC-FDMA stream. Based on the options fft_length, occupied_tones, and
    cp_length, this block creates SC-FDMA symbols using a specified modulation option.

    Send packets by calling send_pkt

    This modulator will use parts of the OFDM-mod but without the FFT component, this component
    will go to the demodulator
    """
(....)
class scfdma_demod(gr.hier_block2):
    """
    Demodulates a received SCFDMA stream. Based on the options fft_length and
    cp_length, this block performs synchronization, frequency correlation, FFT, FDE, IFFT and
```

```
        demodulation of incoming SCFDMA
20      symbols and passes packets up the a higher layer.
21
22      The input is complex baseband. When packets are demodulated, they are passed to the
23      app via the callback.
24      """
25  (....)
```

Listing D.1: *Scfdma.py* classes.

D.1 SC-FDMA Modulator Block

The modulator block also work with messages as data input, but the function that creates them is the same as the one in OFDM. List D.2 shows the creation of the modules and the connections between them in the modulator block.

```
1  #Some differences have been made in this block
2  self._pkt_input = scfdma_swig.scfdma_pkt_input(rotated_const,msgq_limit,self._occupied_tones)
3  self.fft = gr.fft_vcc(self._occupied_tones, True, win, True)#Need to add this FFT Block to work like a
       precoded SC-FDMA
4  self.scale_vec = scfdma_swig.scfdma_scale(self._occupied_tones)#Scale up all the signals in the block.
       Input: Block length
5  self.preambles = scfdma_swig.scfdma_insert_preamble(self._occupied_tones)
6  self.mapper = scfdma_swig.scfdma_mapper(rotated_const,msgq_limit,options.occupied_tones,options.
       fft_length)
7  self.ifft = gr.fft_vcc(self._fft_length, False, win, True)
8  self.cp_adder = digital.ofdm_cyclic_prefixer(self._fft_length,symbol_length)
9  self.scale = gr.multiply_const_cc(1.0 / math.sqrt(self._fft_length))
10
11 self.connect((self._pkt_input, 0),self.fft,self.scale_vec,(self.preambles, 0))
12 self.connect((self._pkt_input, 1),(self.preambles, 1))
13 self.connect(self.preambles,self.mapper,self.ifft,self.cp_adder,self.scale,self)
```

Listing D.2: Module creation in the SC-FDMA modulator and the connection between them.

During the SC-FDMA modulator description, there are some important code lines to point out. List D.3 shows the lines related to the file *scfdma_pkt_input_impl.cc*. The first

method is the constructor and it shows that the algorithm to find the occupied tones is not here any more, because this module does not maps the symbols like OFDM. The second method is *work* and is the main method where the module fills the data frames.

```
scfdma_pkt_input_impl::scfdma_pkt_input_impl(
        const std::vector<gr_complex> &constellation, unsigned int msgq_limit,
        unsigned int fft_length) :
        gr_sync_block("scfdma_pkt_input", gr_make_io_signature(0, 0, 0),
                gr_make_io_signature2(1, 2, sizeof(gr_complex) * fft_length,
                        sizeof(char))), d_constellation(constellation), d_msgq(
                gr_make_msg_queue(msgq_limit)), d_msg_offset(0), d_eof(false), d_fft_length(
                fft_length), d_bit_offset(0), d_pending_flag(0), d_resid(0), d_nresid(
                0), d_save_num(0) {
    d_nbits = (unsigned long) ceil(
            log10(float(d_constellation.size())) / log10(2.0));
}

int scfdma_pkt_input_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items) {
    (....)
    while ((d_msg_offset < d_msg->length()) && (i < d_fft_length)) {
        //in the OFDM (i < d_fft_length) was (i < d_subcarrier_map.size()) so it will have the limit of the
        //occupied carries, now the length is the size of the FFT
        // need new data to process
        (....)
        if (d_nresid > 0) {
            (....)
            out[i] = d_constellation[bits];
            //OFDM out[d_subcarrier_map[i]] = d_constellation[bits];
            //aux = out[d_subcarrier_map[i]];
            //now it doesn't have the d_subcarrier_map[i] so it's all straightforward
            (....)

        } else {
            if ((8 - d_bit_offset) >= d_nbits) { // test to make sure we can fit nbits
                (....)
                out[i] = d_constellation[bits];
```

```
35                myfile << (int) bits << endl;
36                //OFDM out[d_subcarrier_map[i]] = d_constellation[bits];
37                //aux = out[d_subcarrier_map[i]];
38                //now it doesn't have the d_subcarrier_map[i] so it's all straightforward
39                (....)
40            }
41        (....)
42        while (i < d_fft_length) { // finish filling out the symbol with the size of the FFT
43            (....)
44            out[i] = d_constellation[rand];
45            //OFDM out[d_subcarrier_map[i]] = d_constellation[randsym()];
46            //aux = out[d_subcarrier_map[i]];
47            //now it doesn't have the d_subcarrier_map[i] so it's all straightforward
48            (....)
49        }
50    (....)
51    return 1; // produced symbol
52 }
```

Listing D.3: Main code lines in the *scfdma_pkt_input_impl.cc* file.

The second module implemented is the module used to scale the symbols, when they suffer a FFT or IFFT transformation. In the constructor method, the module estimates the value which the symbols are multiplied; and in the *general_work* method, the module gets the inputted symbols and multiplies them with the value. List D.4 shows the scale module file.

```
1  scfdma_scale_impl::scfdma_scale_impl(unsigned int mult)
2          : gr_block("scfdma_scale", gr_make_io_signature(1, 1, sizeof(gr_complex) * mult),
3                  gr_make_io_signature(1, 1, sizeof(gr_complex) * mult)), d_mult(
4                  mult) {
5      d_div = 1.0 / sqrt(d_mult);
6  }
7
8  int scfdma_scale_impl::general_work (int noutput_items,
9                      gr_vector_int &ninput_items,
10                     gr_vector_const_void_star &input_items,
11                     gr_vector_void_star &output_items)
```

```
12 {
13 const gr_complex *in = (const gr_complex *) input_items[0];
14 gr_complex *out = (gr_complex *) output_items[0];
15
16 gr_complex aux;
17
18 for (int i = 0; i < d_mult; i++) {
19     aux = in[i];
20     out[i] = gr_complex(aux.real() * d_div, aux.imag() * d_div);
21 }
22 // Do <+signal processing+>
23 // Tell runtime system how many input items we consumed on
24 // each input stream.
25 consume_each(1);
26
27 // Tell runtime system how many output items we produced.
28 return 1;
29 }
```

Listing D.4: Main code lines in the *scfdma_scale_impl.cc* file.

Now, the modulator inserts the preambles using a preamble module. In OFDM, the preambles enter as an input vector, but this time they are estimated inside the constructor method of the module, in the *scfdma_insert_preamble_impl.cc* file. List D.5 shows the *general_work* method, which finds the right moment, based on the characters inputted, and inserts the preamble frame.

```
1  scfdma_insert_preamble_impl::scfdma_insert_preamble_impl(int fft_length) :
2          gr_block("scfdma_insert_preamble",
3                  gr_make_io_signature2(1, 2, sizeof(gr_complex) * fft_length,
4                          sizeof(char)),
5                  gr_make_io_signature2(1, 2, sizeof(gr_complex) * fft_length,
6                          sizeof(char))), d_fft_length(fft_length), d_state(
7                  ST_IDLE), d_nsymbols_output(0), d_pending_flag(0) {
8      /*creates the CHU-sequence*/
9      unsigned int i = 0;
10     d_preamble.resize(d_fft_length);
11
```

```
12      std::fill(d_preamble.begin(), d_preamble.end(), 0.0);
13      do {
14          d_preamble[i] = gr_expj(-M_PI * pow(i, 2) * 7 / d_fft_length);
15          i += 2;
16      } while (i < d_fft_length);
17
18      enter_idle();
19  }
20
21  int scfdma_insert_preamble_impl::general_work(int noutput_items,
22          gr_vector_int &ninput_items_v, gr_vector_const_void_star &input_items,
23          gr_vector_void_star &output_items) {
24
25  (....)
26
27    while (no < noutput_items && ni < ninput_items){
28      switch(d_state){
29      case ST_IDLE:// this is first symbol of new payload
30     (....)
31          break;
32
33      case ST_PREAMBLE:
34      (....)
35          break;
36
37      case ST_FIRST_PAYLOAD:
38          // copy first payload symbol from input to output
39          memcpy(&out_sym[no * d_fft_length],
40              &in_sym[ni * d_fft_length],
41              d_fft_length * sizeof(gr_complex));
42          (....)
43          break;
44
45      case ST_PAYLOAD:
46     (....)
47          // copy a symbol from input to output
48          memcpy(&out_sym[no * d_fft_length],
```

```
            &in_sym[ni * d_fft_length],
            d_fft_length * sizeof(gr_complex));
    (....)
        break;

     default:
        std::cerr << "digital_ofdm_insert_preamble: (can't happen) invalid state, resetting\n";
        enter_idle();
     }
  }

  consume_each(ni);
  return no;
}
```

Listing D.5: Main code lines in the *scfdma_insert_preamble_impl.cc* file.

After the preambles module comes the mapper module. This module maps the symbols from the inputted frames in bigger frames, this way, the modulator can do the IFFT. List D.6 shows the general code lines in the mapper modules.

```
scfdma_mapper_impl::scfdma_mapper_impl(
        const std::vector<gr_complex> &constellation, unsigned int msgq_limit,
        unsigned int occupied_carriers, unsigned int fft_length) :
        gr_block("scfdma_mapper",
                gr_make_io_signature(1, 1,sizeof(gr_complex) * occupied_carriers),
                gr_make_io_signature(1, 1, sizeof(gr_complex) * fft_length)),
                d_constellation(constellation), d_msgq(gr_make_msg_queue(msgq_limit)),
                d_msg_offset(0), d_eof(false), d_occupied_carriers(occupied_carriers),
                d_fft_length(fft_length), d_bit_offset(0), d_pending_flag(0), d_resid(0), d_nresid(0) {
    (....)

    // find out how many zeros to pad on the sides; the difference between the fft length and the
        subcarrier
    // mapping size in chunks of four. This is the number to pack on the left and this number plus any
    // residual nulls (if odd) will be packed on the right.
    diff = (d_fft_length / 4 - carriers.length()) / 2;

    unsigned int i, j, k;
```

```
18    for (i = 0; i < carriers.length(); i++) {
19        char c = carriers[i]; // get the current hex character from the string
20        for (j = 0; j < 4; j++) { // walk through all four bits
21            k = (strtol(&c, NULL, 16) >> (3 - j)) & 0x1; // convert to int and extract next bit
22            if (k) { // if bit is a 1,
23                d_subcarrier_map.push_back(4 * (i + diff) + j); // use this subcarrier
24            }
25        }
26    }
27    (....)
28 }
29 int scfdma_mapper_impl::general_work(int noutput_items,
30        gr_vector_int &ninput_items, gr_vector_const_void_star &input_items,
31        gr_vector_void_star &output_items) {
32    const gr_complex *in = (const gr_complex *) input_items[0];
33    gr_complex *out = (gr_complex *) output_items[0];
34    unsigned int i = 0,j = 0;
35    // Build a single symbol:
36    // Initialize all bins to 0 to set unused carriers
37    memset(out, 0, d_fft_length * sizeof(gr_complex));
38    for(int i = d_subcarrier_map[0];i < d_subcarrier_map[0]+d_occupied_carriers;i++){
39        out[i] = in[i-d_subcarrier_map[0]];
40    }
41    this->consume(0, 1);
42    return 1; // produced symbol
43 }
```

Listing D.6: Main code lines in the *scfdma_mappr_impl.cc* file.

D.2 SC-FDMA Demodulator Block

The SC-FDMA demodulator block is defined in the class *scfdma_demod* inside the *scfdma.py* file. This class calls the module that does the demodulation, *scfdma_recv*; and the block that converts the symbols in bits and rebuilds the messages received, *scfdma_demod*. List D.7 shows the creation of the modules, in the *scfdma_demod* class; and the connections between them.

```
class scfdma_demod(gr.hier_block2):
    (....)
        self.scfdma_recv = scfdma_receiver(self._fft_length,
                                           self._cp_length,
                self._occupied_tones,
                self._samp_rate,
                self._sym_rate,
                                           self._snr,
                                           options.log)
    (....)
         self.scfdma_demod = scfdma_swig.scfdma_frame_sink(rotated_const, range(arity),
                        self._rcvd_pktq,self._occupied_tones,
                        phgain, frgain)

         self.connect(self, self.scfdma_recv)
         self.connect((self.scfdma_recv, 0),(self.scfdma_demod, 0))
         self.connect((self.scfdma_recv, 1),(self.scfdma_demod, 1))
    (....)
```

Listing D.7: Creation of the modules in the SC-FDMA demodulator and the connections between them.

The block *scfdma_recv* is built in another python file, *scfdma_receiver.py*. List D.8 shows the main code lines in this file. First, the definition *__init__* now has two more parameters, *samp_rate* and *sym_rate*, they are used in the filter design. Second, is the synchronization module definition, this time we only use the pseudorandom noise module. Then, comes the creation of the remaining blocks and the connections between the module.

```
class scfdma_receiver(gr.hier_block2):
     (....)
     def __init__(self, fft_length, cp_length, occupied_tones, samp_rate, sym_rate, snr, logging=False):

    (....)
    #Root Raised Cosine Filter, for it has to be manually implmented FIXME: Try to create one
        automatically
         self.chan_filt = gr.interp_fir_filter_ccf(7, firdes.root_raised_cosine(
           1, samp_rate, sym_rate, 0.05,201))
```

```
9   self.fir_filter_xxx_0 = gr.fir_filter_ccc(7, (1, ))
10
11          win = []
12  win2 = [1 for i in range(occupied_tones)
13          if SYNC == "pn":
14              nco_sensitivity = -2.0/fft_length # correct for fine frequency
15              self.scfdma_sync = scfdma_sync_pn(fft_length,
16                                                cp_length,
17                                                logging)
18          # Set up blocks
19          self.nco = gr.frequency_modulator_fc(nco_sensitivity) # generate a signal proportional to
                frequency error of sync block
20          self.sigmix = gr.multiply_cc()
21          self.sampler = digital.ofdm_sampler(fft_length, fft_length+cp_length)
22          self.fft_demod = gr.fft_vcc(fft_length, True, win, True)
23  self.coarse_freq = scfdma_swig.scfdma_coarse_freq(fft_length)
24  self.ifft_demod = gr.fft_vcc(occupied_tones, False, win2, True)
25  self.scale = scfdma_swig.scfdma_scale(occupied_tones)
26  self.scfdma_frame_acq = scfdma_swig.scfdma_frame_acquisition(occupied_tones,
27                                                               fft_length,
28                                                               cp_length,20)
29
30          self.connect(self,self.chan_filt) # filter the input channel
31          self.connect(self.chan_filt,self.fir_filter_xxx_0,self.scfdma_sync) # into the synchronization alg.
32          self.connect((self.scfdma_sync,0),self.nco,(self.sigmix,1)) # use sync freq. offset output to
                derotate input signal
33          self.connect(self.fir_filter_xxx_0, (self.sigmix,0)) # signal to be derotated
34          self.connect(self.sigmix,(self.sampler,0)) # sample off timing signal detected in sync alg
35          self.connect((self.scfdma_sync,1), (self.sampler,1)) # timing signal to sample at
36
37          self.connect((self.sampler,0),self.fft_demod) # send derotated sampled signal to FFT
38          self.connect(self.fft_demod,(self.scfdma_frame_acq,0)) # find frame start and equalize signal
39          self.connect((self.sampler,1),(self.scfdma_frame_acq,1)) # send timing signal to signal frame start
40                                                # do the fine/coarse freq correction,
41          self.connect((self.scfdma_frame_acq,0),self.ifft_demod,self.scale,(self,0)) # finished with IFFT and
                scaling,
42          self.connect((self.scfdma_frame_acq,1),(self,1)) # frame and symbol timing, and equalization
```

```
43    (....)
```

Listing D.8: Main code lines in the *scfdma_receiver.py* file.

The file that has the synchronization implemented, had a problem, one of the *peak_detector_fb* module, sometimes received values as *Not a Number*. To solve the problem, list D.9 shows the code lines that were added to the module.

```
1  int scfdma_peak_detector_fb_impl::work(int noutput_items,
2          gr_vector_const_void_star &input_items,
3          gr_vector_void_star &output_items) {
4          (....)
5              if (iptr[i] > d_avg * d_threshold_factor_rise) {
6                  state = 1;
7
8              } else {
9
10                 if(isnan(iptr[i]) == true){//NEW CODE - problem when the value is a NaN
11                     d_avg = (d_avg_alpha) * 1 + (1 - d_avg_alpha) * d_avg;
12                 }else{
13                     d_avg = (d_avg_alpha) * iptr[i] + (1 - d_avg_alpha) * d_avg;
14                 }
15                 i++;
16             }
17         }
18         (....)
19 }
```

Listing D.9: Code lines added in the *scfdma_peak_detector_fb_impl.cc* file.

Inside the *scfdma_recv* block, the main module that needs a description is module that does the equalization. This module works like the one in the OFDM. The differences between the modulation techniques are: the preamble sequence this time is calculated inside the constructor method, using the Zudoff-Chu sequence; and the algorithm used to find the occupied carriers shift, in the function *correlate*. List D.10 shows the code lines inside the file where this module is implemented.

```
1  scfdma_frame_acquisition_impl::scfdma_frame_acquisition_impl(
2          unsigned occupied_carriers, unsigned int fft_length, unsigned int cplen,
```

```
3         unsigned int max_fft_shift_len) :
4         gr_block("scfdma_frame_acquisition",
5                 gr_make_io_signature2(2, 2, sizeof(gr_complex) * fft_length,
6                         sizeof(char) * fft_length),
7                 gr_make_io_signature2(2, 2,
8                         sizeof(gr_complex) * occupied_carriers, sizeof(char))), d_occupied_carriers(
9                 occupied_carriers), d_fft_length(fft_length), d_cplen(cplen), d_freq_shift_len(
10                max_fft_shift_len), d_coarse_freq(0), d_phase_count(0), d_save_num(
11                0) {
12
13    d_symbol.resize(d_occupied_carriers);
14    d_known_phase_diff.resize(d_occupied_carriers);
15    d_hestimate.resize(d_occupied_carriers);
16    //Creates the Chu-Sequence and saves it
17    unsigned int k = 0;
18    d_known_symbol.resize(d_occupied_carriers);
19    std::fill(d_known_symbol.begin(), d_known_symbol.end(), 0.0);
20    do {
21        d_known_symbol[k] = gr_expj(
22                -M_PI * pow(k, 2) * 7 / d_occupied_carriers);
23        k += 2;
24    } while (k < d_occupied_carriers);
25
26    std::fill(d_symbol.begin(), d_symbol.end(), 0);
27    for (int i = 0; i < d_occupied_carriers; i=i+2) {
28        d_symbol[i] = 1;
29    }
30
31 void scfdma_frame_acquisition_impl::correlate(const gr_complex *symbol,
32        int zeros_on_left) {
33    // sweep through all possible/allowed frequency offsets and select the best
34    int index = 0;
35    float max = 0, sum = 0;
36    for (int i = zeros_on_left - 20; i < zeros_on_left + 20; i++) {
37        sum = 0;
38         for (int j=0;j<d_occupied_carriers;j++){
39            sum = sum + abs(symbol[i+j])*d_symbol[j];
```

```
40          }
41          if (sum>max){
42            max=sum;
43              index = i;
44          }
45      }
46      // set the coarse frequency offset relative to the edge of the occupied tones
47      d_coarse_freq = index - zeros_on_left;
48  }
49
50  void scfdma_frame_acquisition_impl::calculate_equalizer(
51          const gr_complex *symbol, int zeros_on_left) {
52      //Frequency Domain Equalizer
53      unsigned int i = 0;
54
55      gr_complex gr = gr_complex(1.0, 0.0);
56      // Set first tap of equalizer
57      d_hestimate[0] = d_known_symbol[0]
58              / (coarse_freq_comp(d_coarse_freq, 1)*symbol[i + zeros_on_left + d_coarse_freq]);
59
60      // set every even tap based on known symbol
61      // linearly interpolate between set carriers to set zero-filled carriers
62      // FIXME: is this the best way to set this?
63      for (i = 2; i < d_occupied_carriers; i += 2) {
64
65          d_hestimate[i] = d_known_symbol[i]
66                  / (coarse_freq_comp(d_coarse_freq, 1)*symbol[i + zeros_on_left + d_coarse_freq]);
67
68          d_hestimate[i - 1] = (d_hestimate[i] + d_hestimate[i - 2])
69                  / gr_complex(2.0, 0.0);
70
71      }
72      (....)
73  }
74
75  int scfdma_frame_acquisition_impl::general_work(int noutput_items,
76          gr_vector_int &ninput_items, gr_vector_const_void_star &input_items,
```

```
77          gr_vector_void_star &output_items) {
78      const gr_complex *symbol = (const gr_complex *) input_items[0];
79      const char *signal_in = (const char *) input_items[1];
80
81      gr_complex *out = (gr_complex *) output_items[0];
82      char *signal_out = (char *) output_items[1];
83
84      int unoccupied_carriers = d_fft_length - d_occupied_carriers;
85      int zeros_on_left = (int) ceil(unoccupied_carriers / 2.0);
86
87      if (signal_in[0]) { //When it founds a preamble
88          d_phase_count = 1;
89          correlate(symbol, zeros_on_left); //search for the difference in the position
90          calculate_equalizer(symbol, zeros_on_left); //calculates de equalizer
91          signal_out[0] = 1;
92      } else {
93          signal_out[0] = 0;
94      }
95      //gr_complex gr = gr_complex(1.0, 0.0);
96      for (unsigned int i = 0; i < d_occupied_carriers; i++) {
97          out[i] = d_hestimate[i] * symbol[i + zeros_on_left + d_coarse_freq]* coarse_freq_comp(d_coarse_freq,
                    d_phase_count);
98          //applies the FDE to the other symbols
99      }
100     d_phase_count++;
101     if (d_phase_count == MAX_NUM_SYMBOLS) {
102         d_phase_count = 1;
103     }
104     consume_each(1);
105     return 1;
106 }
```

Listing D.10: Main code lines in the *scfdma_frame_acquisition_impl.cc* file.

The last module to be revised is the *scfdma_demod*, the only difference between this module and the one in OFDM, is in the *demapper* function, listed in the list D.11.

```
1 unsigned int scfdma_frame_sink_impl::demapper(const gr_complex *in,
2         unsigned char *out) {
```

```
3
4     (....)
5     while(i < d_occupied_carriers) {
6     //while (i < d_subcarrier_map.size()) {
7         (....)
8         /*this while used to be d_subcarrier_map.size() now it is the d_occupied_carriers,
9          * it causes missing data*/
10        while((d_byte_offset < 8) && (i < d_occupied_carriers)) {
11        //while ((d_byte_offset < 8) && (i < d_subcarrier_map.size())) {
12            (....)
13        }
14        (....)
15    }
16   (....)
17 }
```

Listing D.11: Main code lines in the *scfdma_frame_sink_impl.cc* file.