INSTITUT FÜR
KOMMUNIKATIONSNETZE
UND RECHNERSYSTEME
Prof. Dr.-Ing. Andreas Kirstädter

IKR

Universität Stuttgart

Design and implementation of an OFDM-based communication system for the GNU Radio platform

Master Thesis

Author
Marcos Majó

Submission date
December 18, 2009

Supervisors
Dipl.-Ing. Matthias Kaschub
Dipl.-Ing. Christian Müller
Dipl.-Ing. Magnus Proebster
Dipl.-Ing. Thomas Werthmann
Dipl.-Ing. Martin Schmidt

Institut für Kommunikationsnetze und Rechnersysteme · Universität Stuttgart
Prof. Dr.-Ing. Andreas Kirstädter
Prof. em. Dr.-Ing. Dr. h.c. mult. P. J. Kühn
Pfaffenwaldring 47 · D-70569 Stuttgart · Germany

Abstract

As technologies quickly evolve and computers and devices become more powerful and economical, paths of research appear allowing a new mass of researchers the chance to work on technologies that were only available to few. This is the case for wireless communications technologies. The practical research was very costly in terms of time and also money, sometimes even being necessary to build prototype circuit boards for testing a possible model. Actual commodity computers have become powerful enough to be able to undertake the signal processing tasks that have always been done by dedicated devices. Cheap computers like the ones we use at home are now able to do the necessary computation that these dedicated devices are doing. This is what Software Defined Radio (SDR) is all about. The translation of the signal processing into software run by a regular computer opens up a huge number of possibilities at an affordable price. Now we can access to all the parameters that were embedded and invariable before. Thanks to SDR now we can analyze and change every value of the system.

This project will try to get a grip of the state of the art in both wireless communication technology and SDR projects. With that objective in mind, in this project an implementation of a wireless communications system will be made. For the physical layer of this system Orthogonal Frequency-Division Multiplexing (OFDM) was chosen as the transmission multiplexing method. This choice has been made because of the advantages that OFDM has shown in terms of channel capacity. It has proven its importance by gaining relevance in two of the most important technologies for the 4th generation of wireless communications: WiMAX and LTE. The software toolkit that has been used for the implementation of the prototype has been GNU Radio; an open source project that is being used by many researchers and manufacturers all over the world, and that is growing steadily in source code available and in active members and projects using it.

The implemented prototype communication system has been a prove of concept that has shown that it is possible to run a communication system of a certain complexity all in software with not much more than commodity equipment. The prototype has shown a very good behaviour in some parts of the system such as the synchronization, and has also shown weaknesses in other parts, like the return channel or the equalization. The data rate achieved in one channel has been 95.2 Kbps, which is a positive result given the context it is in.

Kurzfassung

Durch die rasante Weiterentwicklung der Technologie und dadurch dass Computer und Geräte immer leistungsfähiger und billiger werden, ergeben sich heute viele neue Forschungswege die früher nur wengigen Forscher vorbehalten waren. Dies gilt auch im Gebiet der drahtlosen Kommunikationstechnik. Die praktische Forschung in diesem Gebiet war früher sehr teuer und zeitaufwändig. Manchmal war es sogar notwendig neue Hardware-Prototypen zu bauen um neue Technologien zu testen. Gewöhnliche standard PC's, wie die die wir Zuhause benutzen, sind heute in der Lage die komplette Signalverarbeitung zu übernehmen welche früher von speziellen Geräten erledigt wurde. Die Signalverarbeitung in Software wird als Software Defined Radio (SDR) bezeichnet. Durch die Verwendung von standard PC's hierfür eröffnen sich eine große Zahl an neuen preiswerten Möglichkeiten für die Forschung. Dank SDR ist es nun möglich alle Parameter frei zu wählen welche vorher fest in einem Gerät definiert waren, des Weiteren können alle Werte des gesamten Systems analysiert und verändert werden.

In diesem Projekt wird versucht die neuste Technologie in den Bereichen der drahtlosen Kommunikationstechnik sowie von SDR-Projekten aufzunehmen. Mit diesem Ziel vor Augen wurde in diesem Projekt ein drahtloses Kommunikationssystem aufgebaut. Für die physikalische Übertragungsschicht wurde OFDM als Mehrkanalübertragungsmethode gewählt. Diese Auswahl wurde getroffen, da es bei OFDM viele Vorteile hinsichtlich der hohe spektrale Effizienz gibt. In der vierten Generation der drahtlosen Datenübertragungstechniken gibt es zwei maßgebliche Standards wie WiMAX und LTE die sich durchgesetzt haben. Das Software-Toolkit das für die Implementierung des Prototypen verwendet wurde ist GNU Radio, ein Open-Source Projekt das von vielen Entwicklern und Herstellern auf der ganzen Welt verwendet wird und das ständig an Möglichkeiten und aktiven Mitgliedern wächst.

Der entwickelte Prototyp hat gezeigt dass es möglich ist ein komplexes Kommunikationssystem mittels Software zu beschreiben und mit handelsüblichem Equipment zu betreiben. Der Prototyp zeigt ein gutes Verhalten in einigen Bereichen des Systems wie zum Beispiel bei der Synchronisation. Auch wurden Schwachstellen aufgezeigt in anderen Bereichen des Systems wie zum Beispiel beim Rückkanal und der Entzerrungsphase. Die ereichte Datenrate in einem Kanal beträgt 95.2 Kbps was ein positives Ergebnis hinsichtlich der Möglichkeiten darstellt.

Contents

1 Introduction

This chapter gives an overview of the objectives of this project, as well as the different tasks that will be necessary to fulfill these objectives. The chapter is divided in the motivation section 1.1 and the tasks and contents section 1.2. In the motivation I explain the interests that made us decide for this project, introducing also the context in which it has been thought. The tasks and contents section will explain the different proposed tasks for this project and will locate them in the different chapters and sections of the project.

1.1 Motivation

This project has been conceived as the result of putting many ideas in the practice. It started with the objective of covering and gaining experience in some fields of interest in wireless communications such as SDR or the implementation of a communications system based on the OFDM modulation.

The first idea that starts shaping this project is the importance that SDR has acquired in the last years. As of today it has become a tool that allows researchers great freedom at a very moderate price. Some years ago, the leap between the theoretical part of a technology and its practical implementation was very big, both in terms of requirements and specially costs. The proliferation of SDR projects, specially based on open source software, and the interest from the academic and industrial community for its utilization have created a relatively simple and comparatively very affordable solution for the implementation and testing of a very large number of wireless technologies. One of the objectives of this project is to gain knowledge about actual SDR projects, its state of the art and the possibilities it offers.

Another important point for the development of this project, more specifically its practical part, is our interest for the GNU Radio project, the chosen implementation platform and one of the most developed and active toolkits for the development of SDR applications as of today. This thesis will allow us to gain experience and get acquainted with GNU Radio: its structure, its programming, its advantages and its limitations. The implementation of the communication system in the GNU Radio environment is the perfect way of getting to understand this tooklit.

In order to finish shaping this project the content of the implementation should be decided. The most attractive technology for us was OFDM. It has a very good behaviour in terms of spectral efficiency and it is a very hot technology that has found its way into the most important standards for wireless communications in this new generation called 4G, in which the main standards are the IEEE 802.16 (Worldwide Interoperability for Microwave Access (WiMAX)) and the 3GPP's Long Term Evolution (LTE).

Finally, a motivation point that has a slight futuristic component is the inclusion of the implemented prototype in one of the lectures or laboratory courses of out institute, allowing students to experiment with these technologies. The simple mechanics of software radio can give the prototype high learning potential for its versatility and possibilities of costumization.

1.2 Tasks and contents

For this project a communications system will be implemented. This communications system will include a physical layer based on the OFDM multiplexing method, thus being comparable to actual technologies such as WiMAX or LTE. This communications system will be enhanced with a reliable communications channel based on a Forward Error Correction (FEC) mechanism and a retransmission of wrongly received packets protocol (Automatic Repeat reQuest (ARQ)).

The first part of this thesis is the theoretical part, in which the technologies that will be used during the implementation are explained. I will try to project the role that these technologies will play in the implementation throughout the theoretical explanation. These part is included in chapter 2.

Chapter 3 is dedicated to introduce the concepts and ideas behind the chosen software toolkit for the implementation: GNU Radio. This chapter will explain the structure of GNU Radio with the objective of giving the reader a general idea of the possibilities of GNU Radio. This chapter can be specially useful for people interested in starting a SDR project that are considering GNU Radio as their platform.

The implementation chapter 4 details the steps followed for the set up and creation of the prototype. First of all an estimation is made of the possibilities of successfully implementing such a system. Then each of the steps followed for the implementation of the system is explained, with its key points with the results extracted after each step.

In the end, chapter 5 summarizes the work done and provides some suggestions for the future development of this project.

2 Theoretical concepts

This chapter explains the theory behind this project. First of all, in section 2.1 I will explain the basic concepts and ideas behind SDR, as well as some brief information about the state of the art in available SDR platforms. Afterwards, in section 2.2 I will explain the different concepts that OFDM is built upon as well as the reasons that made us choose OFDM as the multiplexing method that will be implemented. Finally, in section 2.3 I will explain the mechanisms used to provide a reliable channel in the communications, which are the FEC protocol and the ARQ protocol.

2.1 Software Defined Radio

SDR is a concept that has been used since the early nineties. Its original purpose was the creation of a device capable of emulating many radios working at different frequencies. However, it has evolved and is still doing so into a tool which has a much broader use. Software defined radio nowadays is a tool that helps the wireless and mobile communications industry in many aspects.

2.1.1 Basic principle and differences to analog radios

The basic principle of SDR is the reduction to the minimum of the hadware dedicated to signal processing parts and its translation into software that should be runnable by an all-purpose commodity PC. The signal should be generated digitally and dealt with in the PC as much as possible, undergoing modulators, filters, FFT blocks and even amplifiers, all of them done in software, until the signal is ready to be sent. Then, the software gives place to the hardware, which has the function of transforming the digital samples in an analog signal and modulating the baseband signal to the desired carrier frequency to be sent. The last stage would be sending it to the antenna. The concept of SDR is very different to the traditional radios that we have been using until now. Traditional radios rely on dedicated hardware for all its functions and each hardware part has a very concrete and fixed function. The same processor in the PC used by SDR will take care of all the signal processing, and it will be the software the one responsible of dictating the function that will be computed. Figure 2.1 show a block diagram of a software defined radio.

Not having dedicated hardware has a very important advantage in relation to traditional radios. All parameters that the radio uses are set in software, and they are all configurable through software. This makes the development and research of new applications a lot easier, faster and cheaper, as we can use software radio as a prototype where we can test all kinds of variations and configurations. We will not need to make or order hardware in order to try new

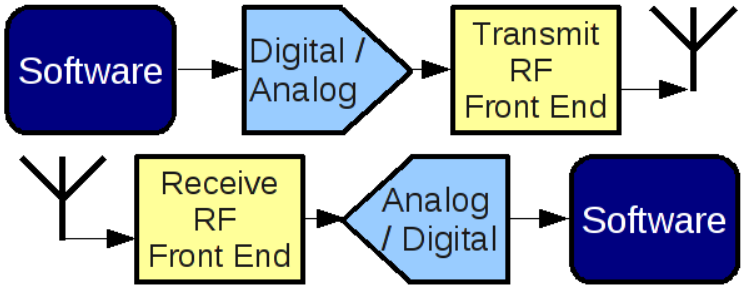

Figure 2.1: Module diagram of a SDR sender and receiver

configurations or variations of any kind. Another advantage is the possibility of having a radio that can work as many radios, as the same device can use different radio technologies without any change in the hardware. This application was one of the first objectives of software radio, but for a number of reasons that I will explain in the next paragraphs, it is still not a very interesting solution.

Software radio has some hardware requirements. It will require some ADC and DAC hardware, as well as RF module dedicated to the modulation to the desired transmission or receiving frequency. These elements and a PC with enough computational power to run the software are the only hardware that we need. If we need to speak in numbers, we should differentiate if the radio will be used for narrow band applications or for wide band. For narrow band applications a regular Pentium PC should have more than enough capacity to meet the requirements. If, however, the application that we want to implement uses up a bigger part of the spectrum for example, a receiver of multiple FM channels at the same time, the requirements get much bigger and we might need powerful PCs in order to process all the data that we are using in the required time [17]. In our application we will be implementing a communications system based on OFDM, which is also a wide band application. We will see in a latter chapter 4 the impact of the computation in the amount of spectrum that can be used.

Not all aspects of software radio are positive. There are also a number of challenges that have been reducing the use of SDR to a limited number of applications. The first of them all is the power consumption of a SDR device. As we have seen in the hardware requirements, we will in many cases need powerful computers to run an application. This means that we will need an amount of power that would never be achievable by a handheld device. The power consumption of SDR devices is not comparable to the power needed by the radios that nowadays work in hardware. Another important drawback is that even if the power needed could be reduced, the size of the hardware needed to process the signal is also much bigger than the dedicated hardware of the traditional radios. This is why the project of having a portable device based on SDR that can be used as many different radios has not evolved very much, and the use of SDR has been reduced to research or applications in the base station, instead of the mobile terminal, where the power requirements are not so critical at the moment.

2.1.2 Different software defined radios available

Many SDR projects can be found in the web. Most of them consist on a free software, most of the time open source that is constantly being modified, plus a hardware part that must be purchased and takes care of the RF part of the transmission. However this hardware is usually not directly related to the software project. Some of the examples that follow this model are GNU Radio [15] or SDR4all [23], which additionally share some of the hardware. Another model is the one followed by the developers of HPSDR [1], which do not rely in the use of a PC for the software radio. These two different approaches will be briefly explained in this section.

The first kind of SDR project that we are interested in is the one which is the most software-based. This orientation focuses on having a robust and well designed modular software that will take care of all the signal processing and the generation of the baseband signal that we want to send, and then the hardware would be taking care of modulating the baseband signal to the desired carrier frequency. In this kind of project the user community will take part in the creation of the software, that will be open source. Each user will be able to create his or her own software applications based on the existing source code, and the users will use a more or less standardized hardware for running their applications. Usually the software contains drivers to run a limited number of RF hardware devices that will be compatible with the software in an easy way. Both GNU Radio and DSR4all use the hardware designed by Ettus Research LLC, which is called [2], and in the case of GNU Radio, it provides drivers for it in the software. The adaptation time to this kind of projects is fairly low, as some examples work practically out of the box and the complexity of the applications can grow slowly according to the level of understanding of the users. GNU Radio is probably the most advanced project in this area and the easiest to start with.

The other completely different kind of SDR project is the one being developed under the name of HPSDR. It is both a hardware and software project, and it parts from a completely different concept. It will not be using a PC for the whole computation, but instead a number of different boards equipped with FPGAs will take care of the different computations that need to be done. This project has started focusing in the development of the basic hardware and encourages its users to develop more hardware boards for more specific operations. Once some of the hardware has been tested now the software part is getting more importance. This project seems to be a bit slow in its development stage and seems to need a fairly big adaptation time as well as some hardware and software knowledge.

2.2 OFDM

2.2.1 Multiple subcarriers and orthogonality

OFDM stands for Orthogonal Frequency Division Multiplexing, and it is the modulation used for the data transmission in the system developed for this project. As its name reveals, OFDM is a multiplexing method, which means that different data channels share the bandwidth available. In the particular case of OFDM the signal is made of independent channels. Each of them will

use a fraction of the available bandwidth. Each of this independent channels is called sub-carrier and transports data that is modulated in the amplitude and phase of the signal. All sub-carriers together form the OFDM carrier. OFDM is called orthogonal because all subcarriers are orthogonal to each other. This orthogonality can be achieved by multiplexing the subcarriers by Frequency-division multiplexing (FDM), which is called multi-carrier transmission or by using Code Division multiplexing (CDM), which is called multi-code transmission. The OFDM transmission system that has been implemented in this project uses multi-carrier transmission. Therefore it uses the same concept as FDM, which assign different frequencies to different signals. Each sub-carrier will use a range of the frequencies available to transmit data, that will travel in the phase of the signal.

This thesis implements a communication system that uses an OFDM multiplexing method for various reasons. First of all, the prototype built for this project had the objective of simulating the lower layers of the ones used by next generation mobile communications (4G), and some of the most important technologies in that field, which are WiMAX and LTE. They both use OFDM as an important part of them [6]. WiMAX stands for Worldwide Interoperability for Microwave Access and is specified in the IEEE 802.16 standard [9]. WiMAX uses OFDM as its multiplexing method in its physical layer. LTE uses OFDM for its downlink, that is from the base station to the terminal, and a precoded version of OFDM called Single Carrier Frequency Division Multiple Access (SC-FDMA). OFDM is also used in many other technologies, like ADSL, digital radio (Digital Audio Broadcasting (DAB)), terrestrial digital TV (Digital Video Broadcasting - Terrestrial (DVB-T)) and terrestrial mobile TV (Digital Video Broadcasting - Handheld (DVB-H)).

The OFDM modulation method relies on the orthogonality between its subcarriers to achieve a good spectral performance. In the case of multi-carrier transmission the chosen frequencies must be orthogonal between each other. This means all frequencies must be multiples of the inverse of the symbol duration. The orthogonality of the frequencies used reduces greatly the cross-talk interference between sub-carriers and increases the spectrum utilisation. It also allows the sender and receiver to be simpler than in the case of FDM. An example of this simplicity is that FDM uses a different filter for each used sub-channel, while OFDM can work with one filter for all subcarriers. The disadvantage of OFDM is that it requires a good frequency synchronisation mechanism, because as the subcarriers are very close to one another small frequency deviations can cause important inter-carrier interference (ICI), i.e. cross-talk between subcarriers. These kind of interferences are mostly caused by Doppler shift due to movement, specially when there are reflexions caused by multipath.

Each of the OFDM subcarriers has a range of frequencies assigned to it, and all of them together fill the spectrum used for the OFDM carrier, that is the bandwidth available. The data in bits will be split among the subcarriers by using a serial to parallel converter and for each subcarrier it is independently modulated using, in most cases, a Quadrature amplitude modulation (QAM) or a Phase-shift keying (PSK) modulation. Then, the OFDM carrier is created with all the modulated subcarriers by using an inverse Fast Fourier Transform (iFFT) module that calculates the time-domain signal with all subcarriers to create a single broad-band

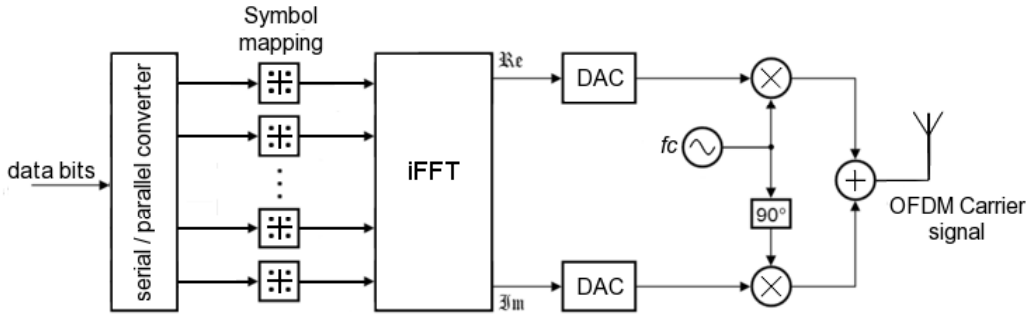

Figure 2.2: OFDM modulator module diagram

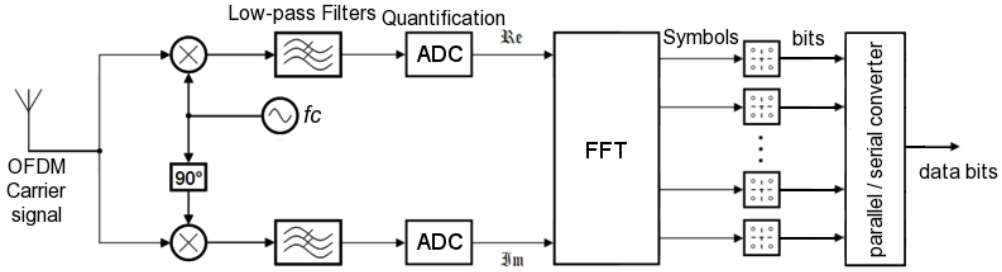

Figure 2.3: OFDM demodulator module diagram

complex signal containing all data belonging to all subcarriers: the OFDM carrier signal. This signal will be used to modulate an Radio Frequency (RF) carrier.

The receiver will separate the received signal in its real and imaginary parts, then they will be low-pass filtered to eliminate mirrored frequencies of the carrier frequency (2fc). Afterwards, they will be quantified with Analog to Digital Converters (ADCs) and then the frequency-domain signals will be calculated with an Fast Fourier Transform (FFT) module. The FFT module will output the different streams corresponding to the subcarriers used and the data in each of them will be independently demodulated using the appropriate symbol detector that corresponds to the modulation used to map the bits into symbols in the modulator module. This functionality has not been implemented in the prototype but is proposed as a future development. Obviously, the sender and the receiver must know the modulation used for each subcarrier in order to demodulate its data correctly. Figure 2.2 and Figure 2.3 show the block diagrams of the OFDM modulator and demodulator.

2.2.2 Delay spread and cyclic prefix

In wireless communication systems the received signal will always be received many times due to the multipath propagation. This effect gives as a result in the receiver a number of signals with different amounts of delay respect the first multipath signal, that usually corresponds with the line of sight path. The difference of delay between the first of the multipath components and the last one is called *delay spread*. The effect of delay spread is specially present in urban environments, in which the number of multipath components is higher than in rural environments, but

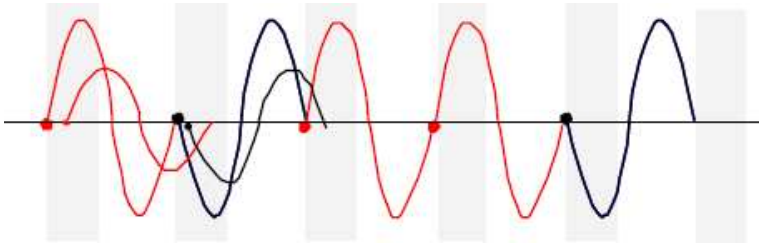

Figure 2.4: Signal in the presence of ISI

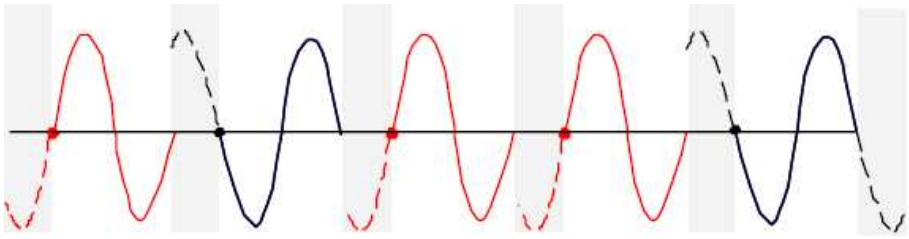

Figure 2.5: The effect of adding a cyclic prefix to a signal

also in environments where sender or receiver are moving at high speeds. This scenario could cause the multipath echoes to have important delays in time, as the target might be moving close to some of the multipath components and far from others. The problems that delay spread can cause are basically two. First of all the different echoes that arrive at different times can come with a different phase in respect to the main signal and they can cause some distortion in the main component. The second problem is the effect that the delayed echoes can cause in the next transmitted symbol. This is called intersymbol interference (ISI). Figure 2.4 shows the effect that multipath echoes can cause on a transmitted signal.

The solution that OFDM proposes to reduce the effect of the multipath propagation in the form of ISI is the addition of some redundancy in the transmitted signal called cyclic prefix (CP). This redundancy is applied to the signal in the time domain and it is meant to create a kind of guard band at the beginning of each symbol that will protect the transmitted symbol from the echoes that cause ISI. The objective of the cyclic prefix is to provide some guard time between symbols that will be discarded by the receiver and will contain the undesired remains of the echoes of the previous symbol. The best way to provide this guard time is to add to the beginning of the symbol a prefix containing the last bits of the same symbol that is being sent. This method is also good for the sender, because it will also have it easier to generate the signal. We have to keep in mind that the oscillators in the sender behave better with a relatively constant rate of bits to send than constantly switching between sending and not sending data. Figure 2.5 shows how the signal looks like with the inclusion of the cyclic prefix. We see that the effect of the ISI is nullified by the CP.

2.2.3 Synchronization and channel equalization

In most wireless systems synchronization plays a very important role in the receiver side. The receiver needs to find the beginning of each symbol correctly. In OFDM systems it includes

finding the right time delay, the frequency deviation of each of the different subcarriers and finally the phase shift of each of the symbols that travel in the subcarriers. All those parameters need to be found by doing some calculations on the incoming signal. More concretely, on some redundancy that is added to the data signal in order to find all these parameters. These redundancy is often referred to as pilots or preamble depending on its situation in the whole transmitted signal.

Pilots are used in many OFDM communication systems and they are known symbols that are sent in some of the subcarriers of the ones used in the transmission. The pilot signals will be known by the receiver, and by looking for them in the received signal all parameters regarding synchronization and equalization will be deduced. The density of pilots in the signal will be proportional to the quality of the synchronization process, but it will also be inversely proportional to the amount of data transmitted. That is the reason why choosing a convenient density of pilots is an important part of the design of the application. Obviously, the quality and variance of the channel will be the key factors that will influence the amount of redundancy dedicated to synchronization in the form of pilots.

Similarly to pilots, preamble redundancy also works in multiple frequencies at the same time. In the case of preambles, the synchronization data will occupy the whole time slot of an OFDM symbol. During the time the preamble is being transmitted all subcarriers will contain preamble data. This method has some advantages and also some weaknesses compared to the pilots one. In this case the information provided by the preamble will give more accurate information regarding all subcarriers. For example, we can find out if there is one subcarrier that fails to send any data or that shows a wrong behaviour. On the other hand, when no preamble data is being sent the system will not be calculating any parameters regarding synchronization, so the updating of the channel data is slower than with the pilots.

The usual procedure for finding the frequency shift is by correlating the signal with itself with a certain delay, and the resulting signal will show a peak from which the frequency and time delay will be extracted. Different algorithms will use slightly different systems, but most of them, including the one used in the implementation, use this correlation method.

The equalization process is usually a simple combination of an amplitude normalizer and a phase corrector. The amplitude in systems that use PSK as modulation for each subcarrier is as simple as amplifying or attenuating all the symbols to the same amplitude. The procedure for finding the phase shift a bit more complicated but it is also similar in most implementations, with some small differences. What the method will do is calculating the expected phase delay between each of the subcarriers. In the case of the synchronization with preambles, the phase difference between subcarriers is extracted from the known symbols of the preamble, and then applied to all the following symbols until the next preamble comes. In the case of using pilots, the difference in phase will be computed by using interpolation between the pilots, which will estimate the expected phase difference between each of the subcarriers. Then the correction is done by rotating the received symbols according to its phase shift.

2.2.4 Phase modulation in the subcarriers

In previous sections we have seen that the information in OFDM systems is transmitted in different frequencies called subcarriers. This section explains how exactly does the information travel in each of these subcarriers.

Each subcarrier is independent from all other subcarriers in terms of how the data of one or another subcarrier is transmitted. Each subcarrier can use a different modulation to transmit the data assigned to it, and the receiver will not depend on any information about the other subcarriers to be able to receive the data belonging to a subcarrier. The sender will be able to modulate the data that goes to one subcarrier with any digital modulation. This decision must be shared by the sender and the receiver, but not by the other subcarriers. As an example, we could use binary phase-shift keying (BPSK) for the transmission of some of the subcarriers and Quadrature Phase-shift keying (QPSK) or QAM for some others, and we would only need to make sure the receiver knows which subcarriers use which modulation. This method helps us optimize each subcarrier by using modulations that fit the subcarrier's SNR. The subcarriers that have better channel conditions could use modulations with more bits per symbol, while subcarriers with lower SNR in the channel could use modulations that are easier to receive, such as BPSK.

2.3 Reliable transmission

One of the objectives of this project is to provide a communication system that offers reliability in the transmission. For that purpose some mechanisms and protocols are necessary. The improvement of the channel that will be implemented to make it reliable has two stages. First of all a method to correct transmission errors will be implemented in order to fix wrongly received packets. It is the FEC mechanism. The information needed for this fix will be contained in the data sent. Of course, as there is a trade-off between redundancy and data rate sent this method will have a limited capacity for fixing errors. The second method is the request of retransmission of frames that couldn't be fixed by the FEC. The union of these two mechanisms is very commonly used in many communication systems and provides reliable communications with good performance of the channel.

2.3.1 Forward Error Correction (FEC)

FEC is one of the most widely used mechanisms to improve the capacity of a channel and thus increase the rate of received data packets in noisy communications, specially in wireless links such as satellite communications, WiMAX, and many other applications. Its basic concept is the addition of redundancy in the transmitted data that will be useful to fix errors produced in the transmission. The correction will not rely on any additional information but the data transmitted itself. This process is known as channel coding [12].

There are two different ways to apply channel coding to a stream of data: Block coding and convolutional coding. Block coding takes small chunks of data of a fixed size (usually up to few

hundreds of bytes) and apply redundancy that contains information about the whole block of data. Therefore, each block is independent from all other blocks. Convolutional coding operates on serial data. It will take data in continuously and the redundancy will use a number of bits (called constraint length and represented with the letter L or K) to generate the output bits. Another important value is the code rate, which shows the relationship between input and output bits. All FEC implementations can be characterized by the parameters n, k and m. The following formulae show the relationship between the constraint length or the code rate and these three basic parameters.

- n = number of output bits
- n = number of output bits
- m = number of memory registers

$$L = k(m - 1) \tag{2.1}$$

$$\text{code rate} = \frac{k}{n} \tag{2.2}$$

Choosing the value of the parameters chosen for the FEC system we want to implement is a very important task in the design of a communications system. There are several issues that must be taken into account. First of all, the characteristics of the channel that we will work on should be known. Then a set of parameters must be chosen in order to find the best performance. The first trade-off that we will face is the threshold SNR of the channel. FEC implementations usually work well in environments with up to a certain SNR. If the SNR falls lower than that threshold the performance of the FEC algorithm is likely to fall down to a point of not performing well at all, outputting data worse than the received in its input in most cases. The second fact that we need to take into account is that adding more redundancy will increase the amount of data to be sent, and that will affect directly the throughput of the channel. Finally, bigger values of m also increase the complexity of the decoder exponentially. To give a general idea of this magnitude, most applications use constraint lengths up to a value of 9 bits (data from 1999) [12].

The FEC implementation of this project will use a convolutional code encoding with a Viterbi decoding mechanism. Viterbi decoding was developed by Andrew J. Viterbi and first published in *IEEE Transactions on Information Theory* in 1967 [8]. Viterbi decoders are one of the two mechanisms to decode data encoded with convolutional codes. The other mechanism is sequential decoding. The main advantage of Viterbi decoders is that the decoding takes a fixed amount of time to conclude. That helps us estimate the feasibility of its inclusion in our system. It is also well suited for hardware implementations.

Viterbi decoding can work on soft bits and hard bits. The data that it works with can be almost analog and the outputted data will be digital. If we use hard bits as input for the Viterbi decoder there must be a decision block before the decoder that takes analog received data in

its input and outputs one of the possible symbols received, which will probably be the one that is closest to the analog data received. When the Viterbi decoder works with soft bits it will quantify the analog data first and then work with the quantified values directly, which usually improves its performance.

Explaining the way the Viterbi decoder works is not a target of this thesis, but I will try to explain very briefly its basic idea in this paragraph. For more information refer to [12]. The main idea of the Viterbi decoding algorithm is to find the stream of data that was sent among all possible streams sent. This is calculated in a rather simple way thanks to the way the data is encoded by the convolutional code algorithm. Each symbol received at the decoder will be followed by another symbol with a fixed known probability. This way, if we receive two symbols that should not come one after the other we detect a possible error. Each time a new symbol comes a path is drawn for all the possible sequences of received symbols and its count of possible errors. In the end the path that accumulated the least amount of errors is the path chosen.

According to the success rate of the received data, the use of FEC algorithms can be seen as an increase of the channel's SNR. As an example, a convolutional code with rate $\frac{1}{2}$ with constraint length of 7 can be seen as an increase of 5 dB in the channel's SNR. This way of analyzing the behaviour of the algorithm can be useful for knowing the increase of the performance that we achieve by using the FEC. Then we can decide on its need or we can modify some of its parameters if necessary.

2.3.2 Retransmission of data

The need of a reliable channel requests the use of some algorithm to make sure all data is received correctly. Therefore a retransmission algorithm is needed in the data link layer, which will be used once the data packets have been received and after they have been validated with a redundancy check mechanism such as *Checksum* or CRC.

There are many different algorithms that can be used for this purpose. The most simple one is the ARQ protocol, which starts a timer in the receiver that waits for the next data packet to be received. If the data packet is received on time the receiver sends an acknowledgement message requesting the next data packet. After a certain time, if the expected packet has not been received the receiver will consider the expected packet lost and will send a message to the sender requesting the same packet that was expected. This is the basic idea. From this simple idea many enhancements have been made in the field of retransmission of lost packets. Some of them are meant to reduce the number of messages sent by inserting the acknowledging messages in the uplink channel, which is called piggybacking.

Another protocol for resending lost packets is the Sliding Window Protocol [11]. It has not been implemented in this project but it is a very broadly used protocol for retransmission that is present is many wireless communication systems. This protocol will acknowledge many received packets at the same time by sending only one acknowledge message. In this case, if the packets have not been received before the time out, the acknowledge message will only acknowledge the

last packet received before the first missing packet. This protocol also saves traffic of acknowledge messages in the system.

3 Architecture of the used platform

This chapter presents the platform used for the implementation of the communication system. This platform is GNU Radio, a free software toolkit that allows the implementation of SDR. It is distributed under the GNU General Public License. GNU Radio's target users are hobbyists, academics and researchers. GNU Radio was started in 2001 by John Gillmore and Eric Blossom, and it has been growing in members and amount of source code. Nowadays it has a large and active community and provides many examples and reference systems and applications for Global System for Mobile communications (GSM), OFDM, High-definition television (HDTV), etc.

In section 3.1 I will explain the concept and basic ideas behind GNU Radio. Then, in section 3.2 I will explain the requirements of all GNU Radio based systems, and I will also explain the hardware peripheral that is most commonly used for the RF part of the system, the Universal Software Radio Peripheral (USRP). After introducing GNU Radio in a theoretical way, and in order to consolidate the previously explained ideas I will present an example application and finally, in section 3.3 I will introduce GNU Radio Companion (GRC), the graphical interface of GNU Radio. Additionally, in appendix A 5 I will give some insight on advanced concepts for developing more complex implementations.

3.1 GNU Radio concept

GNU Radio is a software toolkit designed to allow users to create SDR implementations. It provides mechanisms and tools to create customized and also a great amount of source code modules that can be used as a part of a customized radio as well as stand-alone examples. GNU Radio is used by a large community of hobbyists, academic researchers and commercial companies, specially for prototyping and testing possible implementations.

The software in GNU Radio is divided in modules. These modules can be divided in two large groups. The modules that take care of the signal processing needed in the system are programmed in C++. These signal processing modules can be signal filters, equalizers, FFT modules and so on. The other group of modules include the software needed to interconnect these signal processing modules and configure them according to out needs. These last ones are programmed in Python and they act as some kind of glue that makes the whole system one unit [14]. The possibilities that Python provides for configuration without the need of compiling every time a change is made in a parameter are very convenient for quick reconfigurations and tests. GNU Radio modules are able to operate with infinite streams of data of a certain type. The most common types of data that we will use are complex, short and float. Many GNU Radio applications keep running forever, when the streams of data are infinite. If the stream

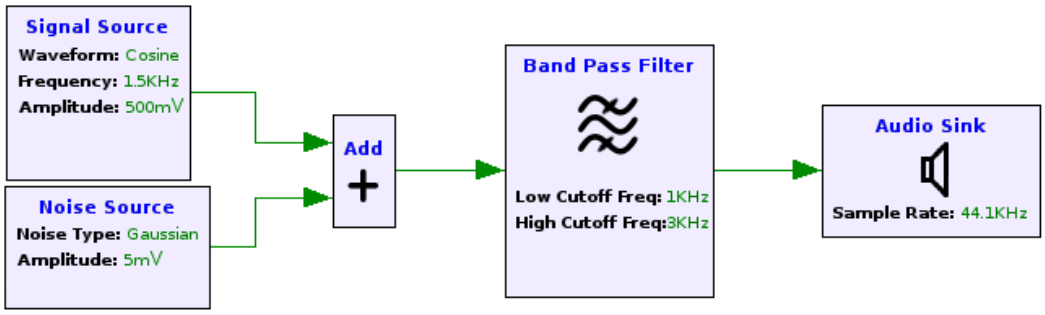

Figure 3.1: Graph representation of a GNU Radio application

has a finite number of bytes the application will finish once the data has been consumed out of the system.

In a GNU Radio application the software modules can also be differentiated in three classes: Source modules, which provide a stream or signal into the system. Examples of source modules are file sources, that get data from a file and insert in into the system, or random data generators, that generate and output a stream of data according to some criteria established by us. The second group of modules are sink modules. They are the contrary of source modules. They receive an a stream of data and consume it. Some examples for these modules are the trivial *null sink*, which consumes the signal without doing anything, file sinks that insert the data from the stream into a file or even graphical spectrum analyzers, that convert the signal received into spectral data and present it to the user in a graphical way as a real spectrum analyzer would do. The last kind of modules are the modules with both input and output ports. They receive streams from their input ports, convert it into a different stream by applying a conversion or a filter and output the result through their output ports. Examples of these modules are band pass filters, Fourier transformation modules, operators or data converters.

We are able to create GNU Radio applications by joining blocks. The union of these blocks creates a graph. We can see a simple example of one of these graphs in Figure 3.1. In the figure we can see various modules with their parameters and the connections between them. The graph representation is important to understand how GNU Radio executes each module in the system. All GNU Radio applications have one thing in common. All applications have a *top block* class that contains an initialization method that will create the instances on the needed modules, build the graph and initialize the parameters of these modules. This initialization method will be called from the main method when we run the application and only after it finishes its execution the system created will be ready to be used. When the application is running GNU Radio will execute the code of the blocks sequentially according to the graph. In the case of the application in Figure 3.1 we would need a Python script to create instances of all these modules, initialize all the parameters that we can see and join them with the *connect* to build the graph.

3.2 Requirements of GNU Radio

Software Defined Radio applications are meant to substitute most of the signal processing in the system with software that will run in commodity computers, but they still need interfaces able to send and/or receive data that travels wirelessly through the air. This section explains what we can do with GNU Radio, and why we need to do it. First of all, if we have an application running infinitely this means that a regular processor will have to deal with the signal or signals in real time. Each application will have a certain degree of complexity and of signal processing to be done that will be translated in requirements. That is why requirements are a critical factor in SDR and, therefore in GNU Radio. SDR has gained popularity during the last ten years due to the fast increase in computation speed in commodity computers.

The requirements of a simple GNU Radio application like the one we have seen in Figure 3.1 are basically any computer, as there are no complicated operations to be done, such as Fourier transformations, and a sound card that will be the responsible of acting as the RF interface. However more complicated applications, such as the implementation made for this project, have much higher computational requirements, because they use many signals and algorithms at the same time and are they must work in real time. The transmissions at higher frequencies also requires special hardware. GNU Radio is meant to work with additional hardware that takes care of translating the software generated digital signal into an analog signal centred in the desired frequency. For a small number of applications working at frequencies up to a few KHz a sound card would be enough, but for most other applications an external device is needed for this function. There is not a very large market for such devices, as there is not too many people working with SDR. However, GNU Radio implements software modules for one of these devices, the USRP. With a USRP we will be able to implement radios operating on frequencies up to some GHz. I will explain the characteristics and possibilities of the USRP in the next subsection.

Another issue related to the computational requirements of GNU Radio applications is the duality of using Python generated code and C++. There is a fraction of the GNU Radio users that find the fact of using both languages at the same time a cause of performace loss. According to the developers of GNU Radio Python should only be used to create and configure the graph, and then the rest of the work should be done by the C++ modules. However there is an ongoing initiative meant to allow GNU Radio applications to run only in C++. This can speed up certain applications, but apparently, it would not represent a very big improvement to well programmed applications that use Python and C++.

3.2.1 The USRP2 peripheral

The USRP is a device designed by Ettus Research with the objective of providing the conversion between the digital baseband signal that is processed in the host computer and the analog intermediate frequency (IF) signal when necessary. It is also responsible of modulating and demodulating the signal to and from the carrier frequency for sending and receiving [10]. The USRP has been specifically developed for SDR and it is highly compatible with GNU Radio. It

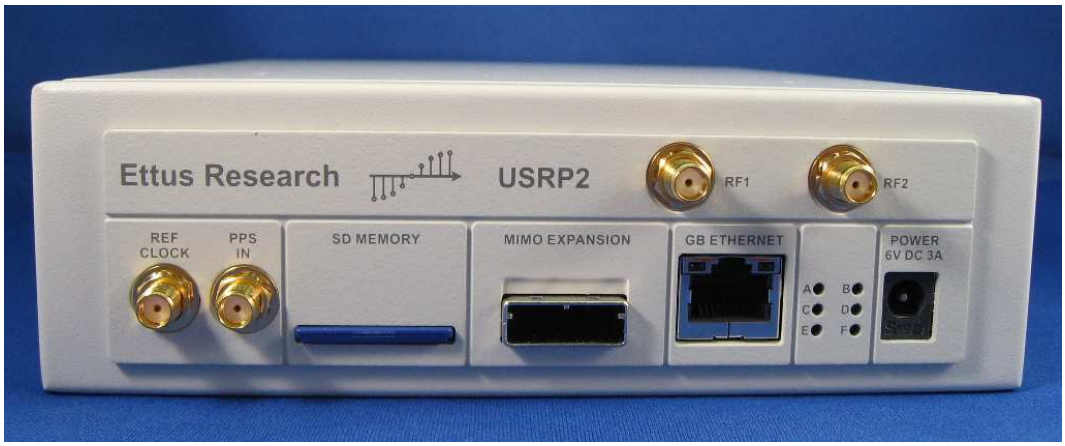

Figure 3.2: Front view of a USRP2 device

is also used in other SDR projects such as SDR4All [23]. Actually there are two different versions of the USRP in the market: the USRP and the USRP2. For this project two USRP2 devices have been used.

The USRP consists of a motherboard, which contains an FPGA that takes care of the high bandwidth math and some ADCs and Digital to Analog Converters (DACs) able to provide bandwidths of around 30 MHz. The FPGA's software can be programmed and stored in an SD card that is inserted in the slot on the front panel on the USRP2. The front view of the USRP2 can be seen in Figure 3.2. The motherboard will be connected to a daughterboard, that will take care of the analog part of the transmission and reception; the closest stage to the antenna. There are several daughterboards available for the USRP family [3]. Each one of them is able to operate at a certain range of frequencies. Actually, we can find daughterboards that operate in frequencies that span from the 0 Hz to the 5.85 GHz. All USRPs' daughterboards include a power supply and one or various antenna connectors, depending on its characteristics. The daughterboard used for this project in both USRP2 devices is the RFX2400 that operates in the industrial, scientific and medical (ISM) band.

The FPGA in the motherboard of the USRP contains digital down converters (DDC) and digital up converters (DUC) that converts the IF signal to the base band and vice versa and decimates it to achieve data rates that are suitable for the PC. This achievable data rate is one of the main differences between the USRP and USRP2. The first version is connected to the PC through a USB 2.0 interface. This connection has proven to be a bottleneck in applications that require large data rates, and that is why the USRP2 has substituted this USB 2.0 connection with a Gigabit Ethernet interface. The USRP2 also includes higher speed and higher precision ADCs and DACs that allow the simultaneous sending and receiving of bandwidths up to 50 MHz. Finally the FPGA in USRP2 is also more powerful [2], which allow the use of the USRP2 as a standalone system that can be used without a host computer in some cases.

Figure 3.3 shows the look of the USRP2 devices together with the host computers that they use in this project's implementation.

Figure 3.3: USRP2 devices with their host PCs

3.3 GNU Radio's graphical interface

A very useful extension that GNU Radio provides is called GRC and it provides a graphical interface that allows its users to easily create GNU Radio applications. GRC has a list of available modules that can be inserted in the application by only clicking on them. These modules can also be configured, and GRC even tells you if the configured parameters are correct. Then, the modules can be connected together also very easily. Afterwards, GRC will build the needed python code that will run the application.

The use of GRC has both advantages and disadvantages, but if it is used correctly it can be a good tool to save a lot of time and to avoid unnecessary mistakes. The main advantages it offers are the simple setup of a system, thanks to its graphical and easy to understand interface, the real time verification of the configured parameters, as it shows the parameters that are not correctly configured and where they are, making it very easy to locate configuration problems. Another advantage is the easy way of inserting and testing new modules in the system; New modules can be added to the system very easily in order to test their behaviour. However GRC also has some disadvantages. GRC only works with the modules and parameters and offers almost no place for customization of a module out of the configuration of the parameters. During the implementation of a module GRC is not recommended, as every time the module is modified one should take care of also modifying the files that allow this module to be controlled from GRC.

We can say that GRC is a very interesting tool in some environments but it should be left out in other environments. In an educational environment GRC would allow students to control and create GNU Radio applications with a very short learning period, and with very successful results, as students would be able to understand all the stages of the system, control the different parameters and see the results in a very short period of time. However, in a research context, where the researcher is adding and customizing modules in all levels GRC would not be recommended, as it adds overhead time to the research process, specially when the researcher already knows GNU Radio well. GRC would be only useful to build test applications and then use the Python code automatically generated by GRC as the starting point for the development

of more complex applications. Custom made modules can be added to GRC by creating an XML file that describes the module.

In the next section we will see a simple example GNU Radio application built with GRC, and in Figure 3.4 we will see how the GRC interface looks like for the sender in that application.

3.4 Basic example application: moving peak

In order to get familiarized with the structure of a GNU Radio application, in this section I explain the first steps that we have to do with GNU Radio to set up a running application. Before starting I want to explain the equipment and software that has been used for this test application. A GNU/Linux PC with the Ubuntu 8.10 (intrepid ibex) distribution. The PC used has 2Gb of RAM memory with a double core Intel Core 2 Duo processor at 2,33 GHz (each processor). Two Gigabit Ethernet network interfaces connect two USRP2 devices with one daughterboard each: the RFX2400 2250 - 2900 MHz Transceiver. This daughterboard allows us to work in the ISM band between 2.4 GHz and 2.5 GHz. With this equipment, the PC works with an average occupation of memory below 50% and with both processors working at around 60%.

The installation of GNU Radio can be done in various manners. Some Linux distributions include GNU Radio in its repositories. Ubuntu is one of them. The GNUradio version that can be found in Ubuntu's repository is relatively old and it lacks some of the GNU Radio modules. However, instead of using the distribution's repositories we use the GNU Radio repository, which hosts the last and complete version of the software. This repository can be found by using the software *subversion*. Then, the installation process is very simple. After all the required applications are available, the common installation commands suffice: ./configuration, make, make install.

The application that I will show in this example will generate a simple sinusoidal signal that will have a time-varying frequency. This signal will then be sent to the USRP2, and another USRP2 will then act as a spectrum analyzer showing the spectrum where we will try to identify our sinusoidal signal. This signal will be a vector of 256 values that represent the spectre that I want to see in the receiver side. This vector will consist (in this example) of only one "1" value and the rest 0 values. This is meant to create a spectrum that consists of only one peak at a given frequency in an instant. The "1" symbol will occupy different positions in the vector as time goes by, so we will try to see this peak in the receiver side move according to the position of the "1" symbol in the inputted vector.

The first considerations that we have to make are related to the hardware limitations that we have. The most important of all is that as we only have a daughterboard for the USRP2 that works at 2.250 to 2900 MHz. So the frequencies that I will be using will be around 2.4 GHz.

The input data is taken from a file. This file contains values of type short (1 byte each) that will be inputted to an iFFT module. To do that first of all the raw short data stream will have to be converted into a vector of 256 complex values, which is the expected input of the iFFT module. For that purpose we will use a module to convert short data into float data, another

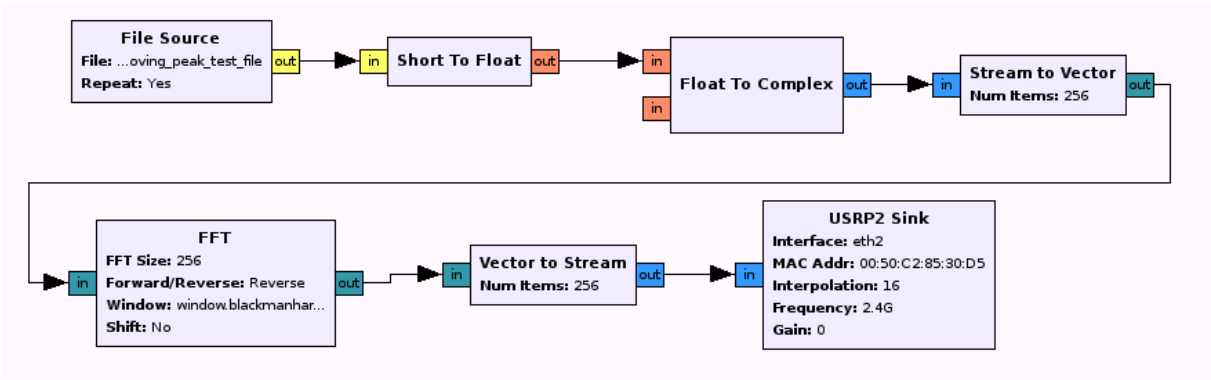

Figure 3.4: Module layout of the sender built with GRC

module will convert the stream of float data into a stream of complex data, and finally another module will take the stream of complex values and group them into groups of 256 values, thus creating the vector that will be used as the input of the iFFT module. The file source module has two important parameters that must be configured, as is the *filename*, the kind of data that we take from it (short: 1 byte, int: 2 bytes, float: 4 bytes, complex: 8 bytes) and *repeat*, which outputs the contents of the file repeatedly, giving the application infinite duration. The type conversion modules that convert the short data stream to the complex data stream are as simple as their names indicate. The module that transforms a stream of data into vectors is also very simple and you can only set the size of the outputted vector. The FFT module is more complex than the ones previously seen. It can compute the FFT and iFFT with the desired number of samples. The input can be float (only allowing forward FFT) or complex (allowing both forward and reverse FFT). Then there is also a configurable *window* parameter which is a Blackman-Harris 1024 by default. The last parameter that can be edited is the *shift* parameter, that will move the correspondence of the 1st value of the vector or the middle value to the reference frequency.

The resulting timely signal that comes from the iFFT module will be sent to the USRP2 to be sent to the air. For this purpose the USRP2's parameters must be configured correctly. In order to find the USRP2 in the system (Gigabit Ethernet interface), the first parameters that must be configured are the *interface* and the *MAC* address of the USRP2 board. Usually the interface name will suffice to identify the board, but both can be set. In order to use this interface the application that we run in GNU Radio will need root permissions. The next parameter is the kind of data that we are going to send to the USRP2. It can be a complex or short value. Then there is an interpolation rate, that modifies the behaviour of the FPGA of the USRP2 and will limit the usable bandwidth and the data rate. The usable bandwidth that we will output are 100 MHz, that is set by the divided by the interpolation rate, which in this case is 16, so the resulting bandwidth will be 6.25 MHz. In Figure 3.4 we can see all these modules as they appear in GRC's interface with it's parameters.

In order to see the resulting signal we will use one of the many sample applications that come with GNU Radio, which is a graphical software spectrum analyzer. We can see in the resulting spectrum in Figure 3.5. In order to show the total bandwidth available in the picture, Figure 3.5

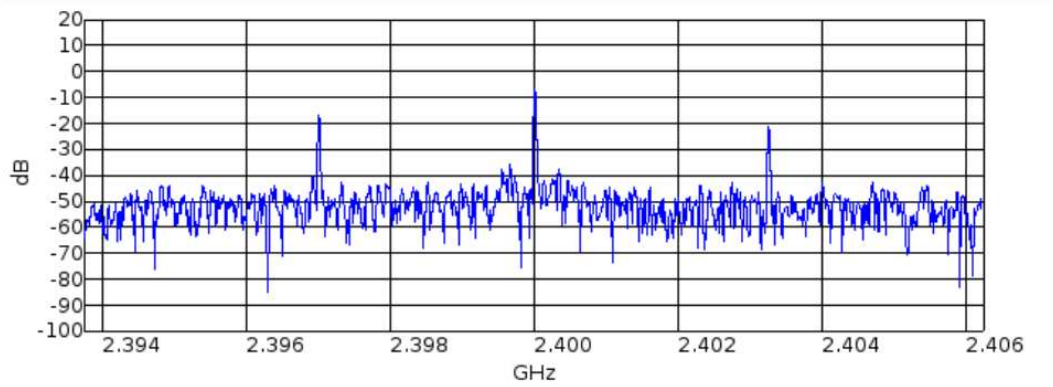

Figure 3.5: Resulting signal of the moving peak example

shows the peak as it moves from the last position to the first one. During some samples we can see the peak disappearing through one of the ends of the use band and appearing in the other end. The picture shows the maximum available bandwidth of 6.25 MHz. The carrier frequency also appears in the centre of the spectrum.

4 Implementation

This chapter describes the implemented communications system in detail. First of all in section 4.1 the objectives of the implementation will be explained. Then the requirements of these objectives will be studied and compared to the resources available. This will set some limitations in the systems that can be built with our resources. After that, the rest of the sections explain the software implementation of the different parts of the communication system: in section 4.2 the OFDM modulator and demodulator will be described. Section 4.3 explains the used modules that implement the synchronization protocol. Section 4.4 explains the implementation of a forward error correction mechanism in the data transmission chain, and Section 4.5 describes the simple protocol used to resend wrongly received packets. Finally, in section 4.6 the performance of the system that we want to implement will be studied and compared with a theoretical measure of the maximum performance achievable. Some physical and mathematical measures will be made of the performance that our system can achieve given our used resources.

4.1 Objectives, resources and limitations

The main idea for the implementation of this project is that the implemented communication system resembles the lower OSI layers of WiMAX. The reason for it is that WiMAX is one of the most important technologies used in the next generation of mobile communications (4G) [6]. Therefore, the implementation is focused on a communication system based on the OFDM multiplexing mechanism, that is the chosen multiplexing mechanism in WiMAX and other 4G technologies.

The WiMAX parameters that are set in the standard [9] are related to many different fields. Some of them are standardized sets of parameters that have been chosen to allow a limited number of standardized system configurations. Depending on the application one of these configurations or another will be chosen. Table 4.1 shows some of these parameters that have been a reference for the chosen parameters of the implemented system. Other WiMAX parameters are requirements that must be fulfilled to grant a correct behaviour of the system in the presence of certain problems for the signal transmission. These include, for example, the update rate needed for the channel estimation and equalization when in the presence of Doppler [13] effect due to a fast moving terminal. These kind of values are less important in the implementation, as we can not reproduce these special channel conditions and the focus of the project is not the simulation of all possible case scenarios.

The resources that SDR requires vary greatly with the application that we want to implement. The reason that SDR has not become a major area of study until around year 2000 [21] lies in part in the fact that SDR requires for many applications a very big amount of computational power,

Table 4.1: Standard sets of parameters for various OFDM configurations [19]

Parameters	Values				
System bandwidth (MHz)	1.25	2.5	5	10	20
Sampling frequency (F_s, MHz)	1.429	2.857	5.714	11.429	22.857
Sample time ($\frac{1}{F_s}$, nsec)	700	350	175	88	44
FFT size (N_{FFT})	128	256	512	1024	2048
Subcarrier frequency spacing	100.8 µs				
Useful symbol time ($T_b = \frac{1}{f}$)	11.16071429 kHz				
Guard time ($T_g = \frac{T_b}{8}$)	11.2 µs				
OFDM symbol time ($T_s = T_b + T_g$)	100.8 µs				

which also includes an important expense of power consumption. The computational power requirements have also played an important role in the implementation of the radio prototype for this project.

The computational resources available were two Pentium Core 2 Personal Computers (PCs) which ran the GNU Radio software and two USRP2 [2] devices. One of the PC+USRP2 systems ran the sender's software and the other ran the receiver's software. The computational power of the two PCs has been a factor that has influenced the OFDM parameters used in the implementation. The computational resources allow different configurations for OFDM implementations, but they also have some boundaries. The computational power available limits the amount of FFT that can be computed in a limited time or some other operations that require a lot of resources in a short time. The USRP2 devices [2] are equipped with an RFX2400 daughterboard [3] that allows us to work in the frequency range from 2.3 to 2.9GHz. The connection between the PC and the USRP is done with a Gigabit Ethernet connection. The observed behaviour of the USRPs and the PCs together shows that the limit in out application is always set by the computational resources in the PC. As a matter of fact, only a fraction of the total bandwidth provided by the USRPs is used, and the Gigabit Ethernet is also not fully used.

4.2 OFDM

4.2.1 GNU Radio's OFDM modules

GNU Radio includes some software modules dedicated to the OFDM modulation. In order to show their basic use and as a kind of documentation for customised OFDM implementations and variations, an example application is available in the gnuradio's examples directory. This example application (`benchmark_ofdm.py`) is a simple OFDM system with transmitter and receiver that runs entirely in software and simulates the whole transmission chain, including the channel. This example has been used as the base source code for the implemented OFDM communication system. In this section I will detail the different modules that intervene in the OFDM communication and will explain the setup of the sender and the receiver, and the modifications needed to convert the software example into a real wireless OFDM system transmitting through an air

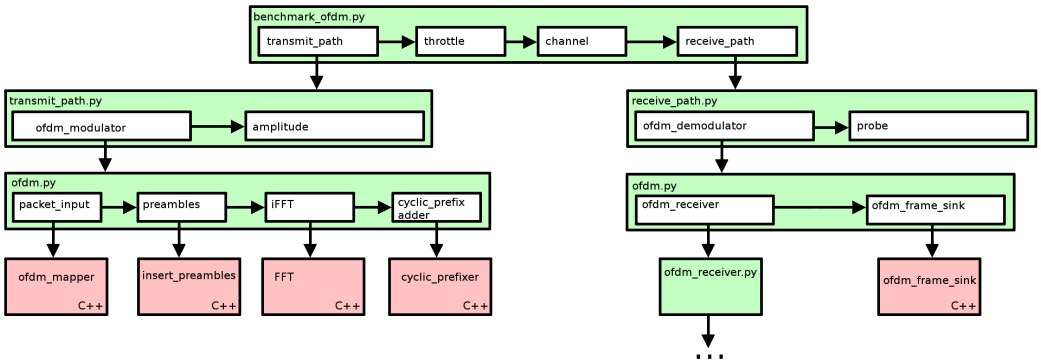

Figure 4.1: OFDM System's module hierarchy

interface between the two USRPs. First of all I will explain the GNU Radio's OFDM modules, and afterwards, I will explain the process of modulation and demodulation implemented in GNU Radio, giving some insight in its capabilities and limitations.

Taking a look at `benchmark_ofdm.py` we can see that it has a lot less lines of code than the expected for the software implementation of an OFDM system. Of course, the source code is divided in many files, that are hierarchically sorted in different abstraction levels. The uppermost levels will set the most general parameters like the total amount of data to be sent or the size of the packets, while as we go deep in the hierarchy more specific paremeters can be set such as USRP-related parameters or the digital modulation that will carry the data in the subcarriers. Figure 4.1 shows the most important blocks and connections in the hierarchy that I will explain in this section. The coloured boxes represent source code files. The ones painted green are written in Python, whereas the ones painted pink are written in C++. The white boxes are instances of modules that are used in the file. The next paragraphs will describe the modules that appear in the figure, both its functionality and its relationships with the rest of the modules in the system.

The uppermost level of the hierarchy shown in Figure 4.1 corresponds to the Python file `benchmark_ofdm.py`. This is the executable file that needs to be run to start the whole system. In the picture we see that it contains the instances of a transmit path and a receive path. This corresponds to the original source code that doesn't transmit through the air, and thus also doesn't use the USRPs. Instead of that, it uses a simulated channel that we see in the module called 'channel'. The 'throttle' module controls the speed of the transmission by configuring the maximum sample rate available.

The whole communication system can be seen as abstraction defined in `benchmark_ofdm.py`. This means that somewhere in the file the information to transmit must be acquired, somewhere it must be sent, received and finally validated or outputted. The first stage, the acquisition of the information to send will happen directly within the python code of `benchmark_ofdm.py`. Test data will be created to fill the packets that will be transmitted in the system. This operation doesn't require a lot of computational power, so a dedicated C++ module for it is not necessary.

The creation of the test data is made in python and inserted in the module chain through a send function.

The channel that the implementation will use is the air, as the signal is going to be sent and received by the USRPs antennae. The room of the laboratory where the USRPs are placed will provide a channel with multipath echoes that will have a very short time delay. Additionally, the USRPs will be static, so the Doppler effect will also be non existant. The most relevant cause of interferences in the channel will be the shared frequency band with the laboratory's wireless Local Area Network (LAN). The knowledge of the channel and some notion of the values of its parameters is very useful for setting the parameters of the system, for example, the length of the cyclic prefix. In the original version of `benchmark_ofdm.py` there is a module called *channel* that simulates a channel with some parameters such as SNR, frequency deviation and phase deviation.

The parameters that can be configured in the uppermost level of the file hierarchy are related to the modules existing in `benchmark_ofdm.py`. Related to the formation of the data packets is packet size, the throttle can be configured with its maximum sample rate, and the channel with its signal to noise ratio (SNR), frequency offset that it introduces and the clock difference between the transmitter and receiver. A part from that, there is also a parameter that sets the total number of Megabytes that are to be transmitted. This parameter belongs to the global execution of the programme and not directly to any module. The rest of the configurable parameters of the system are defined in deeper levels of the hierarchy. At the moment of the execution of the programme by running `benchmark_ofdm.py`, these other parameters that belong to other modules will also be set, and they will be sent through the hierarchy until they reach their target module. This makes the execution a lot easier, as we can set all the configurable parameters at the same time and they will find its destination automatically.

The next level in the hierarchy is also consisting entirely of python files. In this level, the two most relevant files are `transmit_path.py` and `receive_path.py`. Both files are independent of the used channel. This way if we want to change the channel we only need to modify the uppermost level. In `transmit_path.py` we can see that it is simply defining an OFDM modulator and giving it some new parameters. One of them is the number of messages that the modulator should buffer and the other one is called 'padding for USRP', which makes sure that the data that is sent to the USRP is multiple of 128 samples, which the USRP needs to process the data correctly. The `receive_path.py` file is very similar to `transmit_path.py`. It defines an OFDM demodulator and a module called 'probe' that is used for detecting if there is a transmission going on. The modulator and demodulator in this level will be explained in the two next subsections.

The separation of the one file `benchmark_ofdm.py` into two files is a simple process, but it has some small issues that should be taken into account. The objective of this process is to remove the module *channel* and in its place have a sender that includes *transmit_path* connected to an output module that should be the Universal Software Radio Peripheral 2 (USRP2) software module dedicated to sending data through the USRP2. Similar to the sender, the receiver should contain the USRP2 software module dedicated to receiving data from the USRP2 and

then *receive_path*. This can be done easily, but the things to take into account are two. First of all, the number of samples delivered to the USRP2 per second should be the same as the ones received by the USRP2 in the receiver side. This can be done with the previously mentioned *throttle* module in both sides configured at the same sample rate. The second thing that must be done is configure the USRPs correctly. This is done by using the functions *set_freq, set_gain, etc* that set the parameters needed for the use of a USRP2 in the system. Finally the parameter that sets the USRP2's ethernet interface should be made configurable for convenience. Some of the examples that GNU Radio provides that use USRP2 implement this functions, so they can be taken as an example.

4.2.2 OFDM modulator's implementation

The OFDM modulator is located in the python file `ofdm.py` together with the OFDM demodulator. As we can see in Figure 4.1 it interconnects some blocks that are defined in C++. Looking at the definition of the modulator in the python file we can see that it has no input and one output, that is the complex modulated signal at baseband. The modulator includes a send function that takes as a paremeter the payload that needs to be sent and sends it to the first module of its module chain, *pkt_input*.

The modulator's send function uses a very common method for inserting data in a module chain. In this case this module chain will be the one in the modulator itself. The send function receives as a parameter the payload that needs to be sent. This payload will be converted into a data packet by calling a function named *make_packet* from the Python module *ofdm_packet_utils*. This function will calculate and add the Cyclic Redundancy Check (CRC) and the header to the payload that will be used in the end of the receiver to check the validity of the received frame. The send function will not return any value. Instead, it will finish by calling *insert_tail*, a function belonging to the modulator's first module's queue of messages. This function will put the data that needs to be outputted in the end of these queue of messages, and the module itself will access to this data automatically. This method suggests that there are some buffers in the system, which will play a critical role in certain parts of the development of the prototype.

The first module that the modulator implements is the so called *pkt_input* module. It is the first C++ module that appears in the system and the C++ file that defines it is `gr_ofdm_mapper_bcv.cc`. As its name says, this is a file dedicated to OFDM modulations and it takes a stream of bytes and maps it to a vector of complex symbols according to a specified constellation. These symbols are suitable to be used as the input for the iFFT module included in an OFDM modulator. The output of this module has two different exits. One is the actual data, which will be outputted in a vector of the size of the iFFT, and the other one is a vector of characters that will contain one character for each OFDM symbol outputted. It will contain '1' if the OFDM symbol is the first of the frame and '0' in other cases. This way the preambles module will know where the preamble goes, as it always will be at the beginning of the frame.

The next module in `ofdm.py` is *preambles* and it is defined in the C++ file `gr_ofdm_insert_preambles.cc`. This module is the responsible of adding one preamble to each OFDM frame that we want to send. The information in the preamble is a known sequence of symbols that is constructed from a vector of ones and negative ones that has been randomly generated and is stored in the file `ofdm.py`. The input of this module is made to fit with the output of the *pkt_input* module. This way, the first input port will contain OFDM symbols and through the second one it will receive the character that marks the beginning of the frame. If it is the case, it will buffer the OFDM data symbol, output a preamble and then output the buffered data symbol. This module has the same outputs as its previous module, but in this case only the first port will be used.

The next step is the Fourier transformation and it is done by the *iFFT* module that is defined in the C++ file `gr_fft_vcc.cc`. This module takes a vector of complex values and computes the FFT. It is used both for the computation of the FFT and the iFFT just by editing its parameters. It also allows us to set other parameters such as the window used [18] or if the FFT should be shifted.

After the Fourier transformation, the redundancy belonging to the cyclic prefix must be added to the OFDM signal. This is the job of the next module in the modulator, *cp_adder*. This module is also defined in C++ in the file `gr_ofdm_cyclic_prefixer.cc`. It does a very simple job. It takes the OFDM symbol from its input port and copies the last symbols (the number of symbols to copy is specified in the *cp_size* parameter) at the beginning of the symbol, thus creating a symbol with a size that is the sum of the size of the inputted symbol and the size of the cyclic prefix.

Finally, after the FFT, the OFDM symbols are amplified by a multiplication with a constant value. The resulting signal contains the payload, which has been transformed into a data packet, then the redundancy corresponding to the preambles for the synchronization and equalization, then the redundancy corresponding to the cyclic prefix and finally some amplification. The OFDM signal is now ready to be sent to the channel in the uppermost layer of the application, which, in our case, will be the air by using the USRP2 that will send the modulated baseband signal at the transmit frequency, which is 2.5 GHz.

4.2.3 OFDM demodulator's implementation

The demodulator is made of many C++ and Python files. Each of the files that make the demodulator plays a role in the complete demodulation process. This section explains how all the files that make the demodulator are joined and what is their job. In other sections I will give some more detailed information about certain parts of the demodulation process, such as synchronization.

Like the modulator, the demodulator is firstly defined in the file `ofdm.py`. The demodulator has both input and output. Its input comes, in most cases, from the output of the channel. Its output port will contain the demodulated signal. However, the most commonly used output mechanism is the parallell mechanism to the send function that is explained for the modulator.

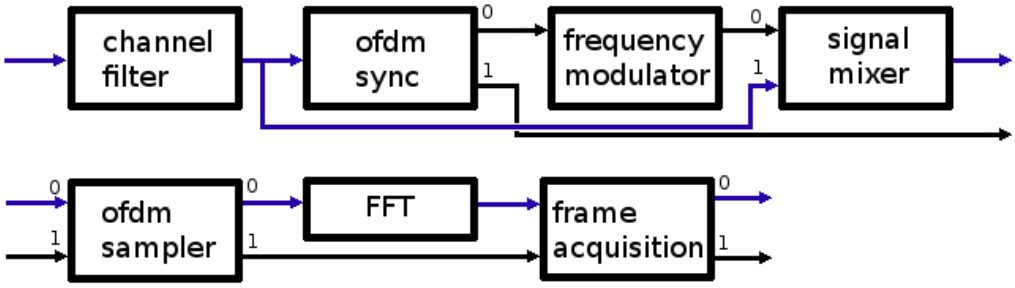

Figure 4.2: Block diagram of the *ofdm_receiver* module

The output data will be sent to a handler via a callback function. There is a function running in a separate thread that monitors the queue of demodulated data packets. Once a new packet enters this queue it will be taken by this function that will check the correctness of the data in the packet by looking at the CRC code. Afterwards it will call the previously mentioned callback function with the payload as a parameter. The callback function will be executed in the uppermost level of the blocks hierarchy, in the `ofdm.py` file.

The inner structure of the demodulator has two big parts, corresponding to the two blocks that are defined in `ofdm.py` and that can be seen in Figure 4.1. One of them, *ofdm_receiver*, takes care of the synchronization and equalization of the CRC signal, while the second module, *ofdm_frame_sink*, is a state machine that demaps the symbols into bits, checks the validity of the synchronized frames and sends them to a superior layer by adding them to the queue of received data packets.

The demodulator's *ofdm_receiver* module is defined in a Python file that includes a number of modules. The fact that the demodulator has one extra layer of Python files is a hint of the complexity of the demodulator. Figure 4.2 shows the blocks that make the *ofdm_receiver* module. The blue arrow in the figure shows the path the signal follows, while the black arrows carry other kinds of data that are not the actual OFDM signal.

The first module of the receiver is a simple Fourier filter on the input signal coming from the antenna that takes the bandwidth corresponding to the number of carriers that contain actual data, which is usually less than the size of the FFT and in our implementation it was 200 used subcarriers for an FFT of 256 subcarriers. Once the signal has been filtered it is fed into the synchronization block. There some calculations will be made to find the right frequency offset and the beginning of the frames. The synchronization block will be explained in the next section. The synchronized signal will then travel in vectors of the size of the FFT to the Fourier transformation module that will output a signal in the frequency domain. Each frequency is a subcarrier and contains information in its phase according to the digital modulation applied to each of the subcarriers; in our case, all of them are modulated with BPSK. The last part of the synchronization will be finding the beginning of the frames and equalizing each subcarrier.

The second module that makes the OFDM demodulator is less complex than its preceding one. The *frame_sink* module is defined in C++ and looks more or less like a state machine. This state machine has three states. One of them is *sync search*, in which the algorithm looks

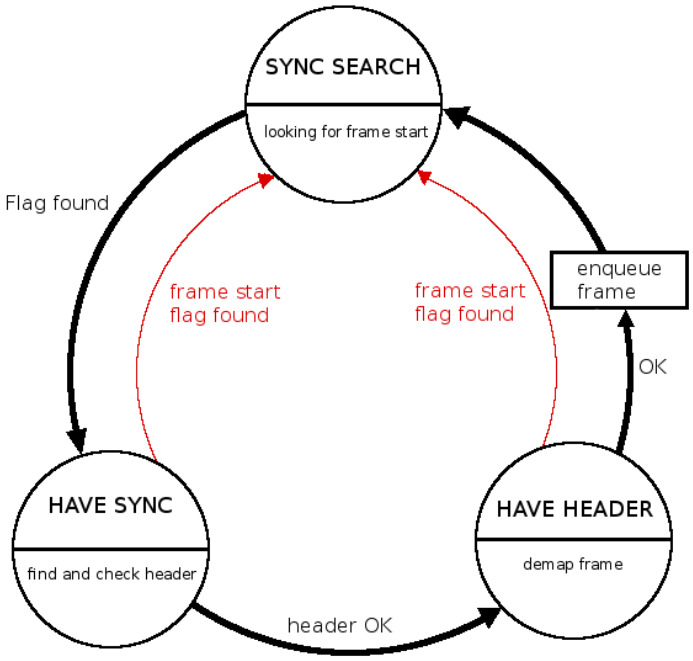

Figure 4.3: Block representation of the state machine in the *frame_sink* module

for the flag in the second input indicating the beginning of the frame. Once it is found it will arrive in the *have sync* state. Then it will let the preamble through. Afterwards the algorithm will demap the symbols and check the bytes corresponding to the header of the data. The header is built in a way that the first and the last half are exactly the same, so this is the way it validates the header. If the header is correct, the state machine will move to the next state, which is called *have header*. There, the algorithm will demap the rest of the frame and insert the resulting data bits in the output queue. As I explained before, that queue is monitored by a thread that will take the data, validate it and send the results to the upper layers of the system. Throughout the duration of the *have sync* and *have header* states, the algorithm will constantly look for beginning of frame flags in its input port. In case it finds one it will detect an error and reset the state machine to the *sync search* state. Figure 4.3 shows the behaviour of the state machine in the *frame_sink* module.

4.3 Synchronization

GNU Radio has three different synchronization mechanisms implemented for OFDM. One of them is the Maximum Likelihood (ML) synchronization [5], the other one is based in the correlation of pseudorandom noise (PN) numbers [7].The third one is an enhanced version of the PN mechanism that uses initial cross-correlation. This enhanced PN synchronization is still showing some flaws and it is not fully usable. One last synchronization method called *fixed synchronization* is also in development, but its use is not recommended and there is no

documentation for it, so I will not comment on it. This leaves us with two main synchronization mechanisms that will be explained in the next subsections. PN and ML.

4.3.1 Pseudorandom Noise implementation

The chosen synchronization mechanism for the prototype has been the PN synchronization. The synchronization takes place in the *ofdm_receiver* module, and it affects most of its modules. The first thing that is done is finding the frequency deviation of the frames from the carrier frequency. Then, the signal will be sampled according to the frequency deviation and then this signal in the time domain will be transformed to the frequency domain with the FFT. The last stage is finding the start of the frame by using the information contained in the preambles and equalizing each subcarrier in amplitude and phase to correct the distortion caused by the channel.

If we observe Figure 4.2 we can see that the input of the *ofdm_sync_pn* module is the signal, but the output doesn't include the signal. Looking at the Python source code and knowing the functionality of its modules we can see that the outputs will be on the one hand a timing signal coming from a peak detection and on the other hand the frequency correction value. The process that the synchronizer follows to find the frequency offset is as follows. First it correlates the input signal with its own conjugate with a delay of half the size of the FFT. The result of this operation will give a maximum when the correlation members are the first and second half of the preamble. The angle of the correlated complex signal will then be extracted. This angle is used for frequency correction by using a sample and hold module controlled by the peak detector. When a peak is detected, the sample and hold will output the angle, which is the frequency offset of the signal. The *ofdm_sampler* module will use that frequency offset to sample the signal that comes straight from the channel. The output of the *ofdm_sampler* is the signal sampled in vectors of the size of the FFT. These vectors will be transformed in the FFT module. Then they will go to the *ofdm_frame_acquisition* module. This module will find the start of the frame based on two known symbols, that will be BPSK PN sequences. PN sequences look like random sequences but are deterministically generated. The input signal will be correlated with these known symbols. This uses the information provided by the indication of the beginning of a preamble that the module receives from its second input. The maximum correlation will give the start of the frame. After this, the same *frame_acquisition* module will use one of the known symbols to estimate the channel response and apply a 1-tap equalization on all subcarriers. This is supposed to help correct amplitude and phase distortion caused by the channel. The synchronized and equalized signal will then exit the *ofdm_receiver* ready to be demapped in the next stage of the demodulation process.

4.3.2 Maximum Likelihood synchronization implementation

The ML synchronization method is another working synchronization method that GNU Radio provides. It works differently to the previously explained PN synchronization method. As

this method has not been used for the implementation, it will not be explained in such detail. However, a module scheme is available in [20], and the theoretical documentation is in [5].

First of all, changing between one synchronization module and another is as simple as changing the value of a variable in the `ofdm_receiver.py` Python file. The ML is one of the synchronization modules that is supported by GNU Radio, so it doesn't require any special configuration. However, some aspects will need to be taken into account. The ML synchronization module uses the redundancy in the cyclic prefix to find the synchronization instead of using the preambles. This means that we need to keep in mind the length of the cyclic prefix, as it is a very important parameter in the robustness of the synchronization. The order of magnitude of the length of the cyclic prefix to achieve good performance of both time and frequency estimators is about 4 or 5 samples for good SNR environments (more than 15dB) or 12 to 15 samples for low SNR channels (4dB) [5]. The estimation of the channel would improve by adding pilots to the OFDM signal for the synchronizer to use, but at the moment it is still not implemented in the GNU Radio ML synchonization.

4.3.3 Synchronization Performance and Measurements

The PN synchronization mechanism has proven to be a good algorithm for our system. It has shown robustness in finding the beginning of the frames correctly. However, there are some indicators that bring us to think that the equalization process could be improved, as we are receiving the symbols with a very low SNR and with some important phase deviation. This can also be caused by the channel, but the equalization is one of the parts of the system that has a weak performance. Figure 4.4 shows a constellation diagram of the received symbols after the equalization stage. There we can see the poor performance of the equalization in both amplitude and phase.

Another parameter that has a direct relationship with the performance of the synchronization is the cyclic prefix. The size of the cyclic prefix is a key factor in the performance of the system, as it can occupy up to 20% or 30% of the overall sent data of the system. Figure 4.5 shows how the size of the prefix affects the overall throughput of the system.

Looking at 4.5 we can see how the size of the cyclic prefix influences the throughput in the system. The measurements have all been made during the same amount of time and with the same packet size. It can be seen that in out environment there is not a minimum size of the cyclic prefix to make the system work. The system will work correctly even if there is only one bit of cyclic prefix, which is the minimum size allowed by the software. Usually, such a graph would show the opposite behaviour, because the effect of the CP helps the successful synchronization, but in this case the CP is is most cases a factor that adds overhead making frames larger instead of a beneficial factor that helps improve the correct synchronization. This fact can be explained by the behaviour of the channel in our laboratory environment. The possible multipath echoes received in the receiver side will have extremely low delays or low power *delay spread* that will almost not cause any interference between SNR symbols. This would be very different in an open air environment with far scatter effects. These results are fitting with the throughput

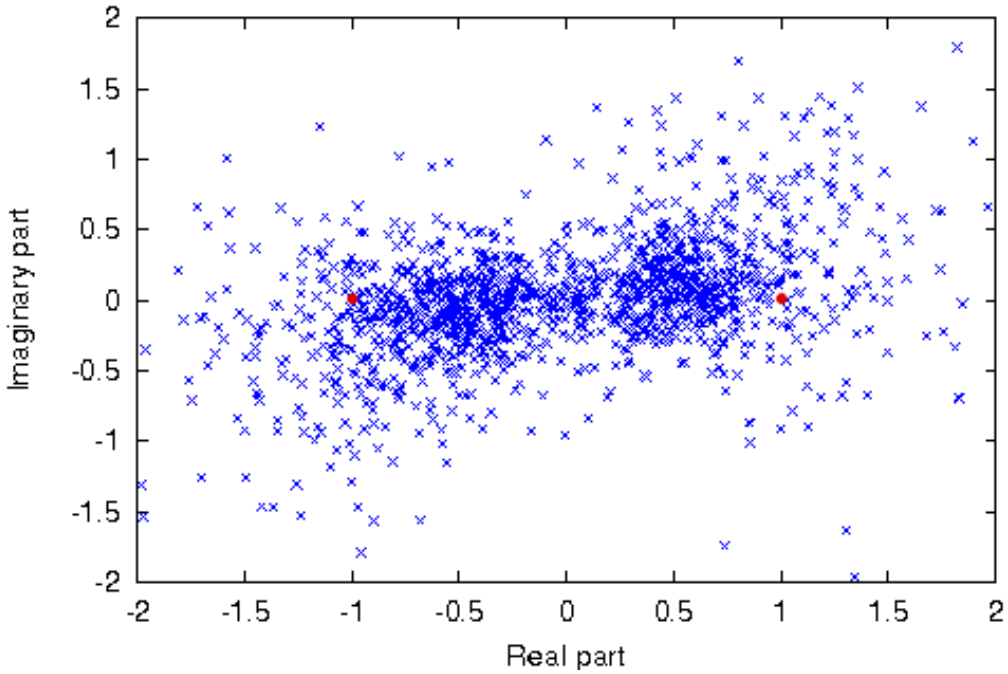

Figure 4.4: Constellation diagram with sent symbols (in red) and received symbols (in blue)

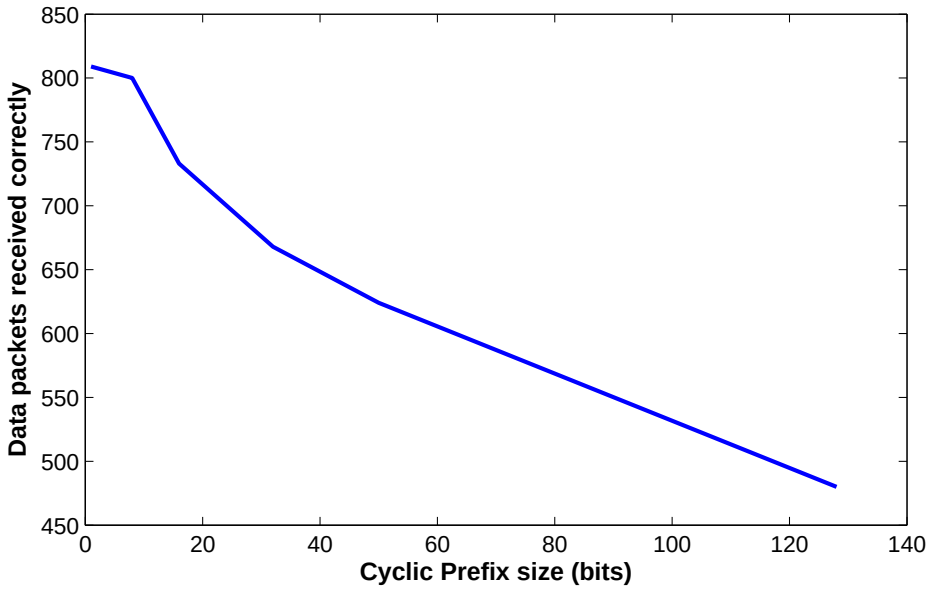

Figure 4.5: Dependence of the size of the cyclic prefix in the throughput of the system

results that we will see in the next section, where we see that the loss of synchronization is not a major cause of transmission errors, proving that the synchronization is robust in the system.

4.4 Forward error correction

In order to improve the throughput in our communication system we need to add a FEC system that will try to solve the transmission errors caused by the channel. GNU Radio provides some modules that are specifically created for this purpose. This section explains how these modules work and how they have been assembled in the communication system, as well as the improvements achieved by using FEC.

4.4.1 GNU Radio's trellis library

The FEC modules are implemented in software package called *trellis* because it includes modules that use trellis codes for FEC. This library is an implementation of a Finite State Machine (FSM) that works for all the offered FEC possibilities. This FSM has been implemented in GNU Radio with the objective of simplifying the implementation of any FEC method in any system. This way once the FSM is added to a system, changing the encoding and decoding for the FEC will only be a matter of configuring the FEC. The actual possibilities that the trellis library offers are trellis encoders that use convolutional codes (CC) for the encoding and Viterbi decoders [12]. There is an attempt of adding a turbo code (TC) based FEC option in the trellis library, but it is still not recommended or well documented, so it has not been used. The configuration that has been used is a trellis encoder with a Viterbi decoder.

The trellis encoder's FSM needs to be defined with the parameters it will use. There are basically two ways of defining these parameters: one of them is to define the number of input symbols, the number of output symbols and the next state and the output state matrix. The other way of defining the FSM is by giving the size of the modulation and the number of shift registers used. This second method is more understandable in a hardware way, but even if we use it the constructor of the class will translate those parameters into the input, output and matrixes parameters. Our implementation uses two possible input symbols, four possible states and four possible output symbols. This means that a hardware implementation would need to use two shift registers for that purpose. With this parameters the FEC encoder is characterized. Now we have to add this module in the sender in order to send encoded OFDM frames.

The Viterbi decoder is also implemented in the trellis library and is also characterized as a FSM. For this reason, the parameters used by the decoder are very similar to the ones used by the encoder, with the exception that the decoder needs to understand the symbols that come to its input port. This means that another needed parameter for the Viterbi decoder is the constellation used in the input. In our case, the BPSK constellation.

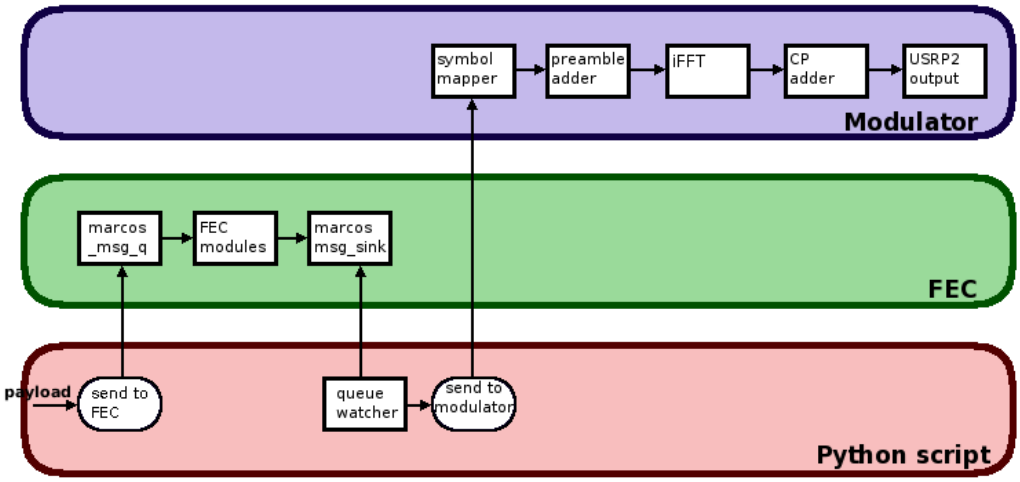

Figure 4.6: Layered structure of the sender's module chains

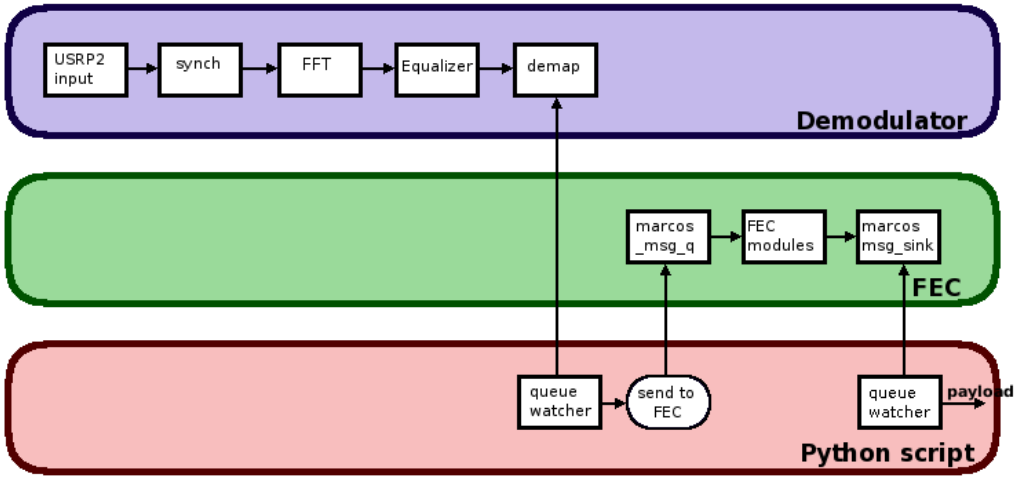

Figure 4.7: Layered structure of the receiver's module chains

4.4.2 Implemented modules for FEC

The modular structure of the implementation of both the sender and the receiver and the implementation of the FEC modules show that there are (both in the case of the sender and in the case of the receiver) two chains of modules. The sender and the receiver have each a chain of modules that starts with the payload bits and ends in the antenna (in the case of the sender) or the opposite in the case of the receiver. Then, there is the chain of modules that is needed for FEC. Figure 4.6 and Figure 4.7 show how the module chains will be assembled; both for the sender and for the receiver.

As we can see in the Figures 4.6 and 4.7, the chains of modules are not assembled into a single chain. Instead, the used strategy has been to run through one chain, return the resulting data to the Python script, and then sending this data to the next chain of modules. This has both advantages and disadvantages. The advantages it offers is a very good modularity, as the FEC works more or less as a plugin that can be added or removed without disrupting the modulator

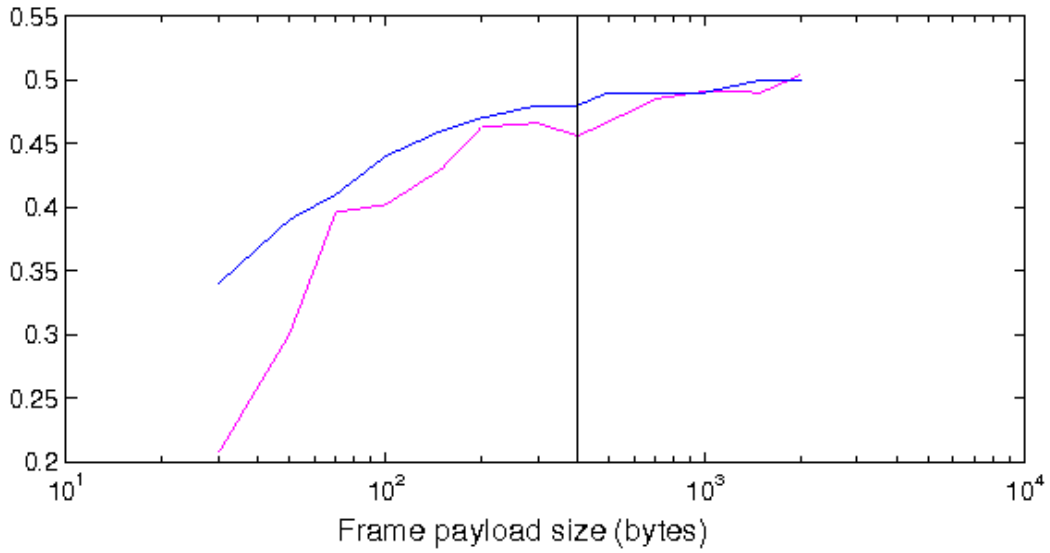

Figure 4.8: Fraction of data packet size dedicated to payload (blue) and frame error rate (pink) in the prototype. The rest is occupied by the header, CRC32 and the preamble

or the demodulator. On the other hand, it might also be source of delays in the system, as data is buffered and not passed from one module chain to the other until a whole frame has been processed, and then it is enqueued before being processed by the other chain of modules.

In order to use the FEC modules in this independent way I have created two modules. One of them is a message queue that will receive a message from a send function and will send it to the next module without applying any transformation to the input data. The name of this module is *marcos_msg_q*. The other module does the opposite function. It stores the messages in a queue that will trigger a callback function in the Python script. The name of this module is *marcos_frame_sink*. These two modules act like a door to and from a module chain. They are not only usable for FEC, they can be used in any system in order to move data from a Python script to a module chain and vice versa. Both modules are implemented in C++ and can be found in the *gnuradio-core* library.

4.4.3 Performance and measurements for the FEC

The performance of the communications system has some meaning after the implementation of the FEC. This section analyzes the performance of the system and shows the measurements done to it that are related to the FEC. We will see the impact of the low SNR observed in the received signal and how the FEC improves the throughput. Figure 4.8 shows a plot in which we can see some relevant parameters that depend on the chosen size of the payload.

The first plot that we can see is the plot showing two measurements. First of all, in blue, we can see the graph with the fraction of the data packet dedicated to payload. In this implementation the data dedicated to synchronization and CRC is fixed. For each frame one OFDM symbol will be dedicated to synchronization and equalization and four bytes will be dedicated to the CRC. This means that the bigger the frame is, the better it will perform in terms of useful data to

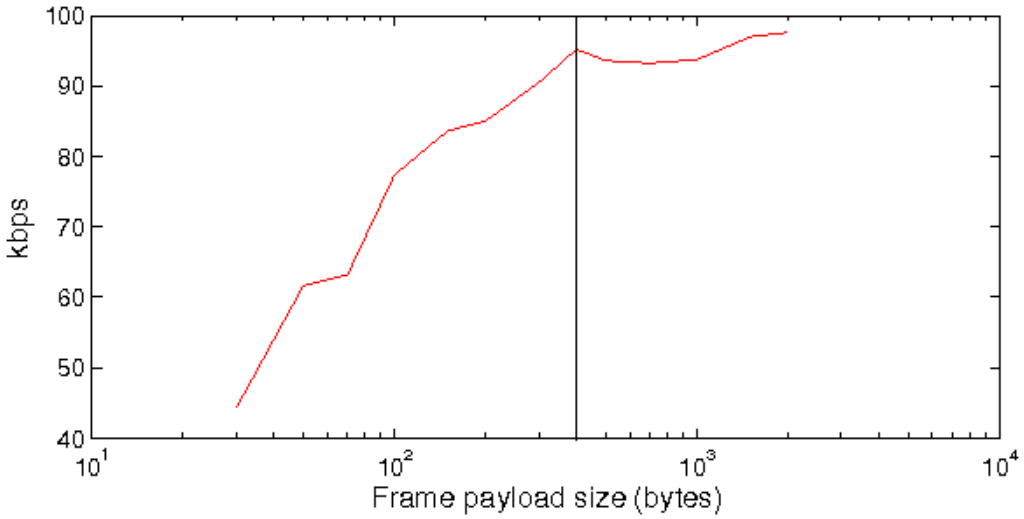

Figure 4.9: Throughput achieved by the prototype depending on the size of the payload of the frames

redundancy ratio. Unfortunately, the size of the frames can not be pushed to the limit because of various reasons. First of all, a big frame can takes longer to be transmitted, and by the time the transmission comes to an end, the synchronization could be lost. If this should happen, the throughput would fall. This would be a problem in channels that change rapidly. An example of these channels would be a channel in which the mobile terminal is moving. In our test scenario, the channel is the air and the environment is the laboratory, an indoor space with very few disruptions. The only cause that can interfere is the wireless LAN in the room, but it would not change the channel's impulse response for good. It would just generate interference when there is activity in the network. This is why the graph shows that for bigger frame sizes we approach to the limit of 0.5. This is because the used FEC has a rate of $\frac{1}{2}$, meaning that for each bit that we want to send, two bits will be generated by the FEC encoder.

The pink graph is the frame error rate achieved, this is the number of frames that have been decoded correctly by the FEC decoder. This graph shows that the error rate becomes higher with the frame size. This results can be easily understood. The cause of this increase in the frame error rate with the size can be caused by two factors. One of them is the possible loss of synchronization, and the other one is the presence of a burst of interference that causes enough erroneous bits to make the FEC decoder fail in the decoding process. Although both of these reasons are logical and understandable, the synchronization failure seems more unlikely, as it depends on the overall channel behaviour more than on the interferences. If the loss of synchronization should be the problem, there should be a clear increase in frame loss from the point the synchronization starts to fail. Also, knowing that the channel is more or less invariant and knowing that the sychronization and channel estimation has been proven robust in most of the tests it makes us think that the reason are the interferences caused by the low SNR of the signal that we receive. Another factor that can be extracted from the graph is that the increase of the frame error rate is lower than the increase of the frame size. It brings us to push the frame size bigger. Figure 4.9 will show how that is translated into throughput.

Table 4.2: Chosen set of parameters for the implemented prototype and performace measures

Parameter	Values
Payload size	400 Bytes
System bandwidth	610 KHz
FFT Size	256
Used carriers	200
Carrier frequency	2.5 GHz
Throughput	95.2 Kbps
Frame error rate	45.6%
% of payload in frame	48%
BER (bit error rate)	0.83%
Spectral efficiency	0.156 $\frac{bps}{Hz}$

The throughput graph in Figure 4.9 shows some interesting results. The first thing that we observe is that it is constantly increasing, so this means bigger frames will give better results. However, we can not increase the size of the frame unendlessly. For bigger frames we have observed that the computational resources used increase to the point of using up all the computation capacity of the used PCs. The performance of the system with 2000 bytes of payload starts to show flaws in the receiver, probably caused by the extra computation needed for the FEC decoding. Should we increase the payload size to 4000 bytes, the system would return errors and it would cease to receive frames. It also seems to have problems with the buffer sizes at that time. However, the packet size should be kept high in order to achieve good throughputs. This is, of course, only applicable to the laboratory's environment or similar indoor environments, as the channel is a critical factor in the performance of the system with the frame size. We have chosen a packet size of 400 bytes, as it shows good results both in frame error rate, and throughput. Table 4.2 shows the set of values used for these measurements as well as the achieved throughput with those parameters and the spectral efficiency, that due to its limitations is lower than other similar wireless systems like WiMAX, that is said to achieve spectral efficiencies of more than 1 $\frac{bps}{Hz}$.

4.5 Automatic repeat request protocol

The ARQ protocol is the Media Access Control (MAC) protocol meant to ensure the reliable transmission by requesting re-sends of the erroneously received frames. This section explains how this protocol has been implemented. This section will also explain the aspects of GNU Radio that make the implementation of real time protocols in GNU Radio systems complicated.

4.5.1 The uplink channel

The first think that we needed to implement in order to have a real time system was an uplink channel. Before starting with the implementation of the MAC layer, the communication system was a unidirectional sender and receiver system, where the receiver never sent anything back to

the sender. Therefore, in order to implement the ARQ protocol, the first ting needed was an uplink channel.

The logical method for the implementation of an uplink channel was either time-division duplexing (TDD) or frequency-division duplexing (FDD). If we choose TDD as the desired way of implementing the uplink channel we will find a deep problem with the structure of GNU Radio. GNU Radio's architecture is aimed at streaming applications. Some of the first examples that were implemented for GNU Radio are things like FM radio decoder. That kind of applications don't need a return channel, and GNU Radio is not well prepared to handle it successfully. GNU Radio's scheduler relies on a steady stream of input data to processing blocks [16]. There are also many modules in the developed prototype that include buffers, and the creation of the OFDM frames is not usually synchronised to anything in particular. There are no simple means of synchronizing the creation of new data frames to certain events such as the reception of a packet. However, some new modules are being implemented by the GNU Radio developers to make GNU Radio friendly to TDD systems. These new modules are called M-blocks, which stands for message blocks. The inclusion of these M-blocks will probably include important changes in the Field-Programmable Gate Array (FPGA) in the USRP. There is also the intention of making the GNU Radio all-C++, thus changing the Python code for C++. This measure would probably make changes in the structure of the systems easier, as there would not be such a strong dependence with how the modules are defined. The unfriendliness of GNU Radio scheduler made us discard TDD as the chosen duplexing method.

The next option was FDD. We have seen in [3] that the USRP allows full duplex communications. The method proposed by the USRP manufacturer is FDD. There are some values that are important to study the possible implementation of FDD in our system. The daughterboard that we have been using is the RFX2400. It has an analog frequency range of 2.3 GHz to 2.9 GHz. The daughterboards have a maximum transmission and receiving bandwidth of 30 MHz. This means that both channels (downlink and uplink) must be fit in these 30 MHz. In our implementation, the bandwidth used is a scarce resource, as more bandwidth means more computation needed for processing the data in the whole frequency band. The fact that we are using a bandwith of 610 KHz means that we can not use all the bandwidth that we want. The biggest issue that FDD would introduce in our system is a tradeoff between the bandwidth used for a channel and the separation between the uplink and downlink channels. The USRP that we are using has the transmission and reception antenna connectors phisically separated. This introduces some isolation between the channels. However it is not always enough to balance the difference in power of the incoming and the outcoming signals. In the ideal case of having the whole 30 MHz available for our transmission and reception channels, we would need to find a suitable balance between the bandwidth dedicated to each of the channels and the bandwidth dedicated to isolation. This bandwidth would be left unused between the two channels in order to improve the inter-channel interference caused by the big difference in power of the sent signal and the received one. The biggest problem in our implementation is that we are using an OFDM based communication system with tight bandwidth resources. OFDM is a broad spectrum modulation, which means that it needs to use a relatively broad frequency band in order to achieve

good throughput. In our system the high computational power caused by FFT computations and FEC decoding reduces the available bandwidth of the receiver. For this reason and due to lack of time to re-structure the system, the FDD solution has also been discarded. Instead, an auxiliary solution will be implemented based on wired Ethernet for the uplink channel and the FDD based uplink channel will be proposed in the outlook as a possible solution for further developments of the prototype.

The chosen return channel has been implemented through Ethernet. The data that needs to be transfered back to the sender will be transfered through Ethernet. To enable that communication Ethernet sockets have been created in the Python file of the top layer of the communication system. The data sent to the sender by the receiver will not look like an OFDM symbol. The needed data will be transmitted through the uplink channel in raw text format. In the next section we will see how this auxiliary solution works.

4.5.2 Protocol implementation

The implementation of the ARQ protocol has used the Ethernet as the transmission method. The implementation in the receiver sends an Ethernet message back to the sender through the Ethernet socket when it receives a packet. The content of the message is the next message to be sent by the sender. On the other side, the sender monitors the Ethernet socket to detect incoming frames. Once a frame arrives it stores the next packet number that it needs to send and sends a packet with that number. This mechanism has a problem. The delay between the sending of one packet and the next causes the synchronization to be lost in many cases. This causes the system to stop working as it should. The best performance has been achieved when the sender has been sending packets all the time and only updating the packet number that was being sent when an acknowledge message was received. One problem of this mechanism is that the buffering of data packets that GNU Radio makes. Most of the time GNU Radio generates a huge number of packets in a row (all of them with the same packet number) and then the updating of the packet number has to wait until all this packets have been sent. This updating of the packet number can take up to few minutes of time to occur.

The dual channel mode implemented in this prototype can only be considered a proof of concept, but it still can not be used as a first final version. With some fine tuning of the parameters of the system, such as delay between sending of packets, the performance of the duplex communication can be improved a little bit, but always reducing the performance of the simplex system. Therefore, no measurements have been made of the throughput of the system with the ARQ protocol because its behaviour has not been good enough. In the case of using the prototype for an educational purpose such as a laboratory course, the uplink channel should work through FDD or TDD. This and the lack of time are the reasons why not a lot of effort has been dedicated to this topic.

4.6 Performance measurements and comparisons

This section puts into context the achieved performance of the communications system. It will tell the causes of the problems found and it will compare some values with theoretical measurements or other similar communication systems.

First of all I would like to measure the theoretical capacity of the channel used and compare it with the capacity achieved. From that overall measurement I will mention the problems the system has and I will deduct some interesting conclusions, that can be translated into future developments of the project.

The capacity of a communications channel can be described by the Shannon-Hartley theorem that depends on the SNR of the channel [22]:

$$C = B\log_2(1 + \frac{S}{N}) \tag{4.1}$$

Using the used bandwidth 610 KHz and the approximate SNR in linear scale, that was, roughly said, around 3 dB that is 2 in linear scale. This rough measure is made by supposing a mean amplitude of the noise of around half of the signal mean amplitude. From Figure 4.4 we could say that the signal has amplitude 0.5 in absolute value and the noise around 0.25. Making use of this roughly approximated data, we have the following result:

$$C = 610K\log_2(1 + 2) = 966.83\,\text{Kbps} \tag{4.2}$$

This result is the maximum achievable channel capacity supposing that the channel is the one we observed in Figure 4.4. In the presence of a better communication channel with about 30 dB of SNR, the theoretical capacity of the channel would increase to around 2.4 Mbps.

The measurements of the throughput have been much lower than the maximum set by the Shannon-Hartley theorem. The measured data comes from calculating the number received frames per second and multiply that number by the number of bits of data in one frame. Therefore, the goodput that comes from the payload of the successfully received frames is:

$$\text{goodput} = 29.75\frac{\text{frames}}{\text{second}}3200\,\text{bits} = 95.2\,\text{Kbps} \tag{4.3}$$

This result is not a bad result taking into consideration that the total amount of data dedicated to the payload is close to 30% of the total sent data. On Figure 4.10 we can see how the overhead is distributed:

Figure 4.10 shows how the sent data (in samples) is distributed for each frame. The configuration that the picture shows is the same one that gave the measured throughput. It is a case scenario that should be usable not only in the laboratory, but also in outdoor environments, due to its large cyclic prefix. However, we have seen in Section 4.3.3 that the cyclic prefix can be reduced almost completely for the indoor environment. By doing this we should be able to improve the percentage of payload data to around 47% of the total amount sent. Another fact that limits the throughput is the amount of the total data sent that is dedicated to the redundancy added by the FEC.

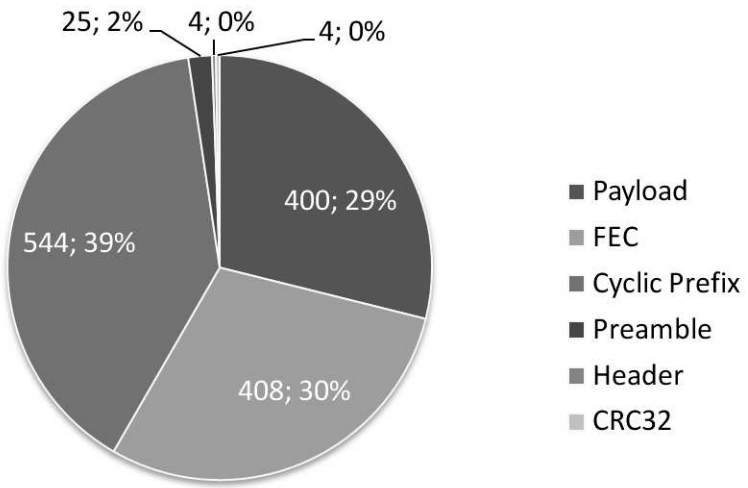

Figure 4.10: Distribution of the data sent for each OFDM frame

Finally, the spectral efficiency has been obtained from the goodput and the bandwidth with this formula:

$$SE = \frac{\text{goodput}}{\text{bandwidth}} = \frac{95.2\,\text{Kbps}}{610\,\text{KHz}} = 0.156\frac{\text{Kbps}}{\text{Hz}} \tag{4.4}$$

This value is not very high. One of the main reasons that the spectral efficiency has not reached levels similar to WiMAX (can reach more than $1\frac{Kbps}{Hz}$) or other wireless systems is because we are using a BPSK modulation for the subcarriers, which only carries one bit for each symbol. Other systems use at least QPSK and often 16QAM or 64QAM, which transmit 2, 4 or 6 bits in each symbol, respectively. Unfortunately, in our system we still can not implement such modulations for the subcarriers because of the low SNR in the channel. If we look at the constellation graph in Figure 4.4 we will easily see it is not wise to introduce a more complex modulation. If the SNR in the system improves, we will see how the spectral efficiency improves considerably.

5 Summary and outlook

In this chapter I will explain briefly all the tasks that were realized for the completion of the project. I will explain what this project has allowed and what the knowledge gained through its realisation can be used for. Additionally, I will talk about the problems and limitations found during the realisation of the project and I will propose some ways for improving the implementation and its possible use in future lectures or practical courses.

This project has been conceived in order to implement a system that will allow us to experiment with state of the art technology of wireless communications. The technologies and protocols that have been dealt with for this project are the ones that our wireless devices will be using in the next years, and they are also technologies that are being studied and taught in electrical engineering courses. These facts motivate the creation of this project. The objective is to have a working prototype system for wireless communications using OFDM in its physical layer and reliable data transmission provided by an error correction mechanism and a retransmission request protocol in the data link layer.

The first steps towards the realization of this project where both theoretical and practical. The ideas behind OFDM had to be clear in order to successfully implement it in software and a basic knowledge of the platform used was also a requirement. Thus, the first stage of the project was to familiarize myself with the tools and technologies that I would be using from that point on. The installation of the GNU Radio software was not very complicated, but it was not trivial, as there were many versions available. In order to get familiarized with the environment, I tried to run and understand basic examples.

Luckily, GNU Radio provides some examples that can be successfully executed without changing any line of code. This was a very convenient fact that compensated the scarce documentation that GNU Radio provides. The problem of documentation makes the implementation slower, as in order to understand the behaviour of a module it is sometimes needed to apply reverse engineering from the working examples.

After experimenting with the first applications, and in order to understand more the behaviour of the USRP2 boards that I was going to use I decided to make a test application that would create a signal with a peak that moved along the usable spectrum. This is the example application explained in section 3.4. This application was implemented by using existing modules provided by GNU Radio and creating a Python script that would create the flow graph an set the parameters for each used module. This application proved to be of great importance because of the experience gained with the Python scripts and because of the things we learned by it. We could understand some of the limitations of the USRP2 boards in terms of distortion. We used a spectrum analyzer to see how the signal was transmitted in all the frequencies that we would later use in the final implementation.

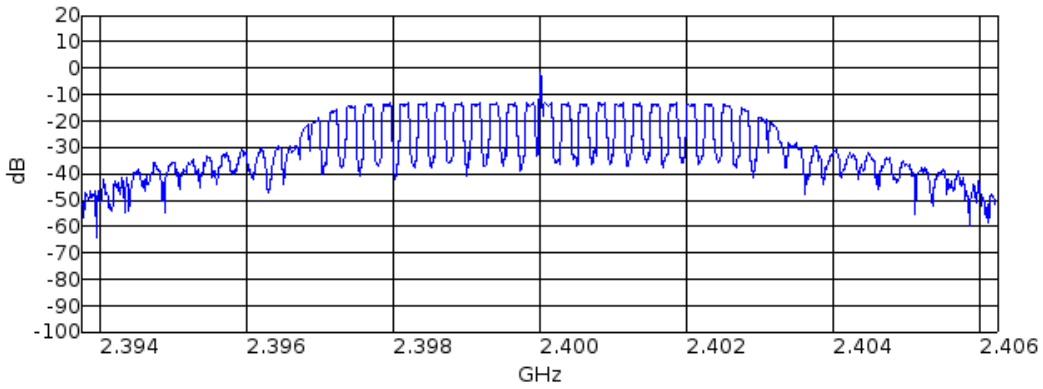

Figure 5.1: Spectrum of the OFDM signal with 1's and 0's in its subcarriers

The next step was the implementation of a transmitter that would send constant values in each of the subcarriers of an OFDM signal. This example was implemented in order to see how the OFDM signal was transmitted by the USRP2 boards and to check the bandwidth limitations that we would probably have. The SNR in the subcarriers was also analyzed by comparing subcarriers with and without signal. The results where positive and helped us see that the centre frequencies and the edges of the used spectrum would probably have a worse behaviour. Figure 5.1 shows a capture of this spectrum where these facts can be seen.

After studying the previous applications a working version of the real OFDM transmitter was implemented. It used BPSK in all of its subcarriers. The receiver was at that time implemented in the same process as the sender, as we were using one of the Python Scrips included with GNU Radio, so it required the separation of the application for sender and receiver and the setup of the synchronization phase. It presented some complications mostly because of the badly documented synchronization modules, that had to be understood mostly through their source code. In this case the modules started to get complicated and a deeper understanding of how GNU Radio works was necessary. In Figure 5.2 we can see how the preambles were found by the correlation in the synchronization process.

With the synchronization successfully running the next step was the implementation of the FEC algorithm. This task proved to be the most complicated. Unfortunately the implementation of the trellis package of GNU Radio didn't support FEC with soft bits with BPSK in the subcarriers. This fact and the fact that the OFDM modules were bound together in a way that it was hard to insert new modules between them made the implementation of the FEC algorithm complicated.

In the end, the ARQ protocol was implemented, but the problems that GNU Radio has with TDD and the lack of bandwidth that would allow us to use FDD forced the protocol to run through an auxiliary wired connection encapsulated in Ethernet frames. Additionally the performance shown by the use of the dual channel was deceiving and it was left as a pending issue due to lack of more time to dedicate to that cause.

The most interesting enhancement that can be made to the implementation is the implementation of the duplex mode via wireless. From my experience I could say that the way that looks

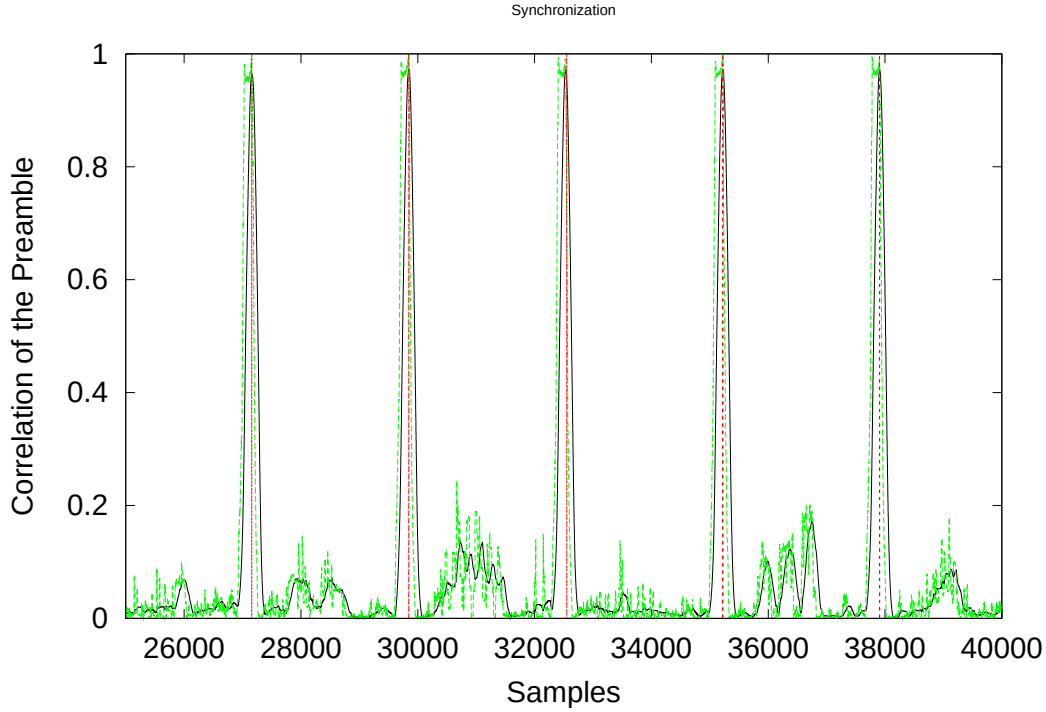

Figure 5.2: Synchronization of the preambles

more realistic is by implementing it by using FDD. GNU Radio supports FDD so the GNU Radio architecture wouldn't need any changes. Then, there should be a balance point between the bandwidth dedicated to the transmitted OFDM signal, the free bandwidth that would avoid interference between the sending and receiving signal and the computational resources used.

Another interesting enhancement for the system would be the possibility to allow multiple users to be transmitting to the same receiver, and also the use of different modulations for each of the subcarriers. This would make the system interesting allowing us to test the channel capacity and the behaviour of the system in different load situations.

Finally, this communications system can be used for lectures or practical courses, allowing users to rapidly understand many of the concepts explained in the lessons that are critical in wireless communications. For that purpose I would propose the inclusion of this prototype in the modules that can be controlled by GNU Radio's graphical interface, the GRC. By doing that the learning time needed to understand the usage of the application would be shortened a great deal, allowing lecturers and students to spend their time optimally in the concepts related to the wireless communications and not in learning how to interpret Python code.

Author's statement

Hereby I certify that I know and I accept that I have no right to exploit the results of my Master Thesis by any means without the written permission of the Institute of Communication Networks and Computer Engineering (IKR).

Furthermore, I certify that I have realized this work on my own, and that all sources that I have used or consulted are duly noted herein.

Marcos Majó

Appendix A: Advanced GNU Radio concepts

This section briefly explains some concepts that are not necessary to know for the basic usage of GNU Radio, but they become necessary as we start developing new modules or when we want to understand in detail the behaviour of each of the modules in terms of buffering, CPU load or time it takes to complete it's job. First I will give some insight into the creation of new C++ signal processing modules. Afterwards I will explain some concepts about GNU Radio's scheduler.

A.1 Creation of new C++ signal processing modules

As GNU Radio is free software it allows its users to modify and create everything about it. However, it tries to encourage its users to create new software modules by providing some documentation for it [4]. For the creation of new modules a few things must be taken into account by the programmer. First of all the modules will have to be defined in C++ and they must be derived classes of the class *gr_block* or one of its subclasses. This procedure converts the class into a GNU Radio block by defining things like its name or inputs and outputs in a standardized way. Secondly, all modules must contain a method called *general_work* that will compute the results that will be outputted from the data read from the input streams. After creating the C++ block following these guidelines we also need to create a file that acts as a kind of glue between Python and C++ code. It is the Simplified Wrapper and Interface Generator (SWIG) *.i* file. This file looks like a header file and it is used by Python to understand the module's ports and parameters, as well as its default values.

The creation of a new module is not a complicated task, but it can become tricky because of the few documentation for it. The same way the creation of a new module requires to follow some rules, if we want to create a new package we must also learn how to do it by looking at other modules and doing reverse engineering. My experience working on this project made me come to the conclusion that customizing GNU Radio is easiest by using a simple module as an example and editing its contents. It has proven to be the easiest and quickest way to have a custom module in the system.

A.2 Flowgraphs and scheduler

For applications that have significant constraints, let it be time constraints or computational ones, we will understand that the performance of each module according to these constraints is a key factor for the success of our application. GNU Radio has been working on the scheduling algorithm, as it has been very commented due to its impact on the performance of applications.

Originally the whole flow graph of GNU Radio was run through one single thread, but since 2008 a new scheduler has been implemented in order to allow one thread per block, thus increasing the control over the scheduler. It is important to keep in mind the scheduling problems that can occur in the programming stage. A good application will keep a balance throughout the modules and its buffers in order to avoid modules becoming bottlenecks.

Appendix B: Lists and Registers

B.1 References

[1] *High Performance Software Defined Radio.* URL: http://openhpsdr.org/.

[2] Ettus Research LLC, ed. *USRP2: The Next Generation of Software Radio Systems.* Mountain View, CA, USA.

[3] Ettus Research LLC, ed. *Transceiver Daughterboards for the USRP Software Radio System.* Mountain View, CA, USA.

[4] Radio Ware from University of Notre Dame. *Writing A Signal Processing Block for GNU Radio.* URL: https://radioware.nd.edu/documentation/advanced-gnuradio/writing-a-signal-processing-block-for-gnu-radio-part-i.

[5] Jan-Jaap van de Beek, Magnus Sandell, and Per Ola Boerjesson. "ML Estimation of Time and Frequency Offset in OFDM Systems". In: *IEEE Transactions on Signal Processing* 45.7 (July 1997). Pp. 1800–1805. DOI: 10.1109/78.599949.

[6] Kyung-Ho Kim. "Key technologies for the next generation wireless communications". In: *Hardware/software codesign and system synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th international conference.* 2006. Pp. 266–269. DOI: 10.1145/1176254.1176319.

[7] T. M. Schmidl and D. C. Cox. "Robust frequency and timing synchronization for OFDM". In: *Communications, IEEE Transactions on* 45.12 (1997). Pp. 1613–1621. DOI: 10.1109/26.650240. URL: http://dx.doi.org/10.1109/26.650240.

[8] Trudy E. Bell. "The Quiet Genius: Andrew J. Viterbi". In: *The Bent of Tau Beta Pi* (2006). Pp. 17–21. URL: http://viterbi.usc.edu.

[9] IEEE. *802.16-2009. IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Broadband Wireless Access Systems.* Technical standard specification. IEEE, 2009. DOI: 10.1109/IEEESTD.2009.5062485. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=5062428.

[10] Firas Abbas Hamza. *The USRP under 1.5X Magnifying Lens!* 2008. URL: www.scribd.com/doc/9688095/USRP-Documentationd.

[11] A. S. Tanenbaum. *Computer networks.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. Pp. 202–218. ISBN: 0-13-162959-X.

[12] Chip Fleming. *A Tutorial on Convolutional Coding with Viterbi Decoding.* tutorial. Derwood, MD, USA. July 2002.

[13] Eric W. Weisstein, ed. *Doppler Effect.* URL: http://scienceworld.wolfram.com/physics/DopplerEffect.html.

[14] Eric Blossom. "GNU Radio: Tools for Exploring the RF Spectrum". In: *Linux Journal* (122 2004). URL: http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html.

[15] Eric Blossom. *GNU Radio*. URL: `http://gnuradio.org`.

[16] Jesper M. Kristensen. "GNU Radio; An Introduction". Presentation handout. Aalborg University. Denmark. Mobile Developer Days 2007. 2007.

[17] Eric Blossom. *Interview to Eric Blossom on GNU Radio*. Ed. by Slashdot. Sept. 27, 2002. URL: `http://tech.slashdot.org/article.pl?sid=02/09/27/1420201`.

[18] LDS group, ed. *Understanding FFT Windows*. URL: `http://www.ee.iitm.ac.in/~nitin/_media/ee462/fftwindows.pdf`.

[19] Hassan Yaghoobi. "Scalable OFDMA Physical Layer in IEEE 802.16 WirelessMAN". In: *Intel Technology Journal* 8.3 (Aug. 2004). ISSN: 1535-864X.

[20] Matt Ettus, Thomas W. Rondeau, and Robert McGwier. "OFDM Implementation in GNU Radio". Presentation handout. 2007 Virginia Tech Symposium on Wireless Personal Communications. June 6, 2007.

[21] Kiichi Niitsu. "Reconfigurable RF System for Software-Defined Radio". Presentation handout. Keio Univ. Kuroda Laboratory, Japan. 2006.

[22] University of Manchester, ed. *The Shannon-Hartley Theorem*. Manchester, UK 2006.

[23] Pierre Jallon and Mérouane Debbah. *Software Defined Radio for all*. URL: `www.sdr4all.eu`.

B.2 List of Figures

B.3 List of Tables

B.4 List of Abbreviations

3GPP ..... 3rd Generation Partnership Project
ADC..... Analog to Digital Converter
ARQ..... Automatic Repeat reQuest
BPSK ..... Binary Phase-Shift Keying
CC..... Convolutional Code
CDM..... Code Division Multiplexing
CP..... Cyclic Prefix
CPU..... Central Processing Unit
CRC..... Cyclic Redundancy Check
DAB..... Digital Audio Broadcasting
DAC..... Digital to Analog Converter
DDC..... Digital Down Converter
DUC..... Digital Up Converter
DVB-H..... Digital Video Broadcasting - Handheld
DVB-T..... Digital Video Broadcasting - Terrestrial
FDD..... Frequency-Division Duplexing
FDM..... Frequency-Division Multiplexing
FEC..... Forward Error Correction
FFT..... Fast Fourier Transform
FM..... Frequency Modulation
FPGA..... Field-Programmable Gate Array
FSM..... Finite State Machine
GRC..... GNU Radio Companion
GSM..... Global System for Mobile communications
HDTV..... High-Definition TeleVision
HPSDR..... High Performance Software Defined Radio
ICI..... Inter-Carrier Interference
IEEE..... Institute of Electrical and Electronics Engineers
IF..... Intermediate Frequency
iFFT..... Inverse Fast Fourier Transform
ISI..... InterSymbol Interference
ISM..... Industrial, Scientific and Medical
LAN..... Local Area Network
LLC..... Limited Liability Company
LTE..... 3GPP Long Term Evolution
MAC..... Media Access Control
ML..... Maximum Likelihood
OFDM..... Orthogonal Frequency-Division Multiplexing
OSI..... Open Systems Interconnection
PC..... Personal Computer
PN..... Pseudorandom Noise
PSK..... Phase-Shift Keying
QAM..... Quadrature Amplitude Modulation

QPSK ..... Quadrature Phase-Shift Keying
RAM ..... Random Access Memory
RF ..... Radio Frequency
SC-FDMA ... Single Carrier Frequency Division Multiple Access
SD ..... Secure Digital
SDR ..... Software Defined Radio
SNR ..... Signal to Noise Ratio
SWIG ..... Simplified Wrapper and Interface Generator
TC ..... Turbo Code
TDD ..... Time-Division Duplexing
USB ..... Universal Serial Bus
USRP ..... Universal Software Radio Peripheral
USRP2 ..... Universal Software Radio Peripheral 2
WiMAX ..... Worldwide interoperability for Microwave Access
XML ..... eXtensible Markup Language