Implementation of a wireless OFDM system using USRP 2 and USRP N210 kits

By

Amr Youssef Fathy Youssef

Karim Mokhtar Hamdna-Allah Hassan

Mohamed Gamal Mostafa

Mohamed Taha Saad

Under the Supervision of

Dr. Mohamed Khairy

A Graduation Project Report Submitted to
the Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science
in
Electronics and Communications Engineering

Faculty of Engineering, Cairo University

Giza, Egypt

July 2012

*“**Do not go where the path may lead ,***
go instead where there is no path and leave a trail . ”

Ralph Waldo Emerson

Table of Contents

List of Tables

List of Figures

List of Symbols and Abbreviations

SDR	Software Defined Radio
OFDM	Orthogonal Division Multiplexing
WIMAX	Worldwide Interoperability For Microwave Access
LTE	Long Tem Evolution
USRP	Universal Software Radio Peripheral
DBPSK	Differential Binary Phase Shift Keying
IEEE	Institute of Electrical and Electronics Engineers
QOS	Quality Of Service
PRR	Packet Received Ratio
PC	Personal Computer
RF	Radio Frequency
SCR	Software Controlled Radio
ISR	Ideal Software Radio
USR	Ultimate Software Radio
CR	Cognitive Radio
GPS	Global Positioning System
OS	Operating System
GRC	Gnu Radio Companion
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
DAC	Digital To Analog Converter
ADC	Analog To Digital Converter
DSP	Digital Signal Processing
GE	Giga Ethernet
DDC	Digital Down Converter
DUC	Digital Up Converter
CIC	Cascaded Integrator-Comb
TX	Transmitter
RX	Receiver
AGC	Automatic Gain Controller
LFTX	Low Frequency Transmitter
LFRX	Low Frequency Receiver
VHF	Very High Frequency

UHF	Ultra High Frequency
USB	Universal Service Bus
LO	Local Oscillator
FM	Frequency Modulation
SNR	Signal To Noise Ratio
FIR	Finite Impulse Response
BW	Band Width
FDM	Frequency Division Multiplexing
CDM	Code Division Multiplexing
PAPR	Peak To Average Power Ratio
SC-FDMA	Single Carrier-Frequency Division Multiple Access
DAB	Digital Audio Broadcasting
DVB-T	Digital Video Broadcasting-Terrestrial
DVB-H	Digital Video Broadcasting-Handheld
CP	Cyclic Prefix
QAM	Quadrature Amplitude Modulation
PSK	Phase Shift Keying
IFFT	Inverse Fast Fourier Transform
ISI	Inter Symbol Interference
ICI	Inter Carrier Interference
FFT	Fast Fourier Transform
BPSK	Binary Phase Shift Keying
FEC	Forward Error Correction
LSB	Least Significant Bit
CRC	Cyclic Redundancy Check
PLL	Phase Locked Loop
GRC	Gnu Radio Companion
BER	Bit Error Rate
MSB	Most Significant Bit
FLL	Frequency Locked Loop
CW	Clock Wise
CCW	Counter Clock Wise
QPSK	Quadrature Phase Shift Keying
EOF	End Of File
PN	Pseudo Noise

Acknowledgments

First and foremost, thanks God the most merciful and most beneficent to who we relate any success in achieving any work in our life.

It gives us the greatest pleasure to express our deepest attitude and warmest thanks to ***Dr.Mohamed Khairy***, Lecturer of Communications, Electronics and Electrical Communications Department, Faculty of Engineering, Cairo University for his kind supervision and valuable suggestions. Dedicating some of his valuable time and encouraging guidance were the reasons for enriching this work.

It seems very difficult to select the suitable words expressing our respect and appreciation to ***Eng.Hazem Nasef*** for his valuable assistance, encouragement and helpful instruction throughout the work.

Also, we would like to extend our deep thanks to our families for their permanent help and support through every step not only in this work but also through every step in our life.

Lastly, some words which must be said, the GNU Radio mailing list is a treasure and some mailing list answers should be gold weighted. Please send feedback corrections for any technical mistakes. GNU Radio is great and impressive project and the USRP is an amazing device. We always ask ourselves, how they build this great project ? We will never know the answer.

Authors,

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 4^{th} 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.
.

Chapter 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 into the motivation section 1.1 and the tasks and contents section 1.2. In the motivation we 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 communication systems on USRP boards. 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 report 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).

1.2 Tasks and contents

For this project, communication systems will be implemented. These communications systems will include Analog environment example(s) , digital modulation techniques such DBPSK and a physical layer example based on the OFDM multiplexing method, thus being comparable to actual technologies such as WiMAX or LTE. The first part of this report is the theoretical part, in which the technologies that will be

used during the implementation are explained. We will try to project the role that these technologies will play in the implementation throughout the theoretical explanation, also we intended to introduce the concepts and ideas behind the chosen software toolkit for the implementation: GNU Radio. We will explain the structure of GNU Radio with the objective of giving the reader a general idea of the possibilities of GNU Radio. This part is included in chapter 1.This chapter can be specially useful for people interested in starting a SDR project that are considering GNU Radio as their platform .In chapter 2 , we go deeply inside communication systems , the best and easiest start is dealing with analog environment .So , in this phase we exhibited analog simple educational example to make the reader not feeling afraid of GNU Radio . The implementation of digital systems , which in fact more sophisticated than analog comes in chapter 3. Actually, in chapter 4 we reach our aim of this project which is implementing OFDM complete systems , you can consider that chapter 2 and chapter 3 as motivation for the reader to be able for understanding OFDM . Successful transmission and reception of OFDM packets have been accomplished with reasonable BER. Also, we discussed in this chapter different issues affecting the whole process . In the end, chapter 5 summarizes the work done and provides some suggestions for the future development of this project.

1.3 Software Defined Radio

1.3.1 Definition

SDR is a concept that has been used since the early nineties. Its original purpose was the creation of a device (radio) capable of emulating many radios working at different frequencies. In addition, it can tune to any frequency band, transmit and receive different modulations and different physical parameters across a large frequency spectrum by using a programmable hardware and powerful software . An alternative definition for is a collection of hardware and software technologies that enable reconfigurable system architectures for wireless networks and user terminals. SDR provides an efficient and comparatively inexpensive solution to the problem of building multi-mode, multi-band and multi-functional wireless devices that can be enhanced using software upgrades. As such, SDR can really be considered an enabling technology that is applicable across a wide range of areas within the wireless industry .SDR-enabled devices can be dynamically programmed in software to reconfigure the characteristics of equipment. In other words, the same piece of "hardware" can be modified to perform different functions at different times. SDR performs significant amounts of signal processing in a general purpose computer, or a reconfigurable piece of digital electronics or the combination of both. SDR is where all the signal manipulations and processing works in radio communication are done in software instead of hardware. So, in SDR, signal will be processed in digital domain instead in analog domain as in the conventional radio. The digitization work will be done by a device called the Analog to Digital Converter (ADC). Fig.1.1 shows the concept of Software Defined Radio. This figure shows that the ADC process is taking place after the Front End (FE) circuit. FE is used to down convert the signal to the lower frequency called an Intermediate Frequency (IF); this is needed due to the limitation of the speed of current Commercial of The Shelf (COTS) ADC. The ADC will digitize signal and pass it to the baseband processor for further processes; demodulation, channel coding, source coding and etc. In conventional radio, all this

processes are done in hardware. In this project, we seek to explore the viability of using GNU Radio; an open source SDR implementation and the Universal Software Radio Peripheral (USRP); an SDR hardware platform, to transmit and receive the OFDM signal with BPSK modulation. Quality of Service (QoS) in terms of Packet Received Ratio (PRR) on the data transmitted will then be investigated and analyzed. Software defined radio nowadays is a tool that helps the wireless and mobile communications industry in many aspects. For your knowledge, the term (SDR) was introduced by Joseph Mitola from MITRE Corporation in 1991. His first paper on SDR was published in 1992 at IEEE National Telesystems Conference. Though the concept was first proposed in 1991, software-defined radios have their origins in the defense sector since the late 1970's in both the U.S. and Europe.

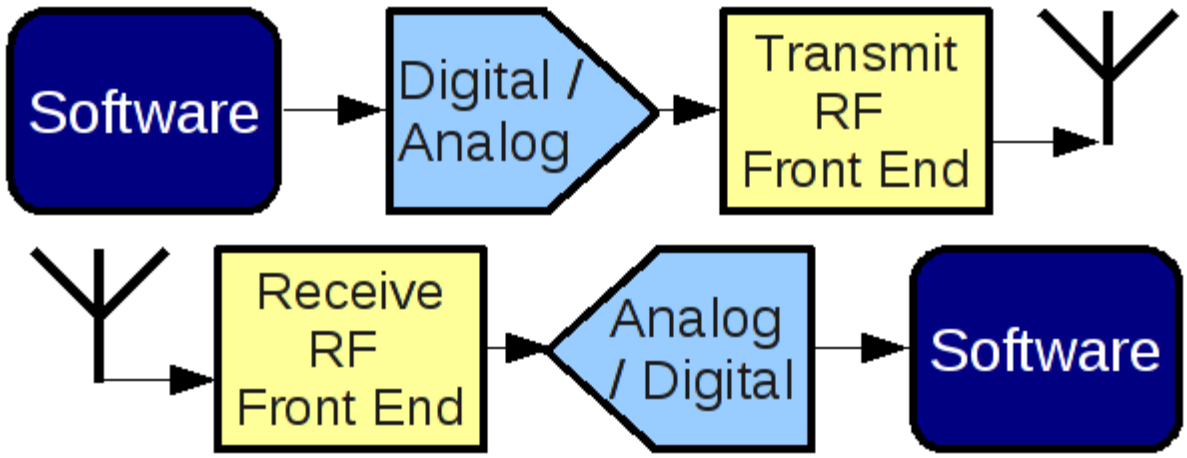


Figure 1.1: Module diagram of a SDR sender and receiver

1.3.2 Basic principle and difference to analog radios

The basic principle of SDR is the reduction to the minimum of the hardware 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. 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 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 we 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. In our application the aim is implementing a communications system based on OFDM, which is also a wide band application. Not all aspects of software radio are positive and there is no free lunch. 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.

1.3.3 Defining software radio using tiers

The SDR Forum has defined the following tiers, describing evolving capabilities in terms of flexibility

- Tier 0 :**The Hardware Radio**
  The radio is implemented using hardware components only and cannot be modified except through physical intervention.
- Tier 1 : **Software Controlled Radio (SCR)**
  Only the control functions of an SCR are implemented in software - thus only limited functions are changeable using software. Typically this extends to inter-connects, power levels etc. but not to frequency bands and/or modulation types etc.
- Tier 2 : **Software Defined Radio (SDR)**
  SDRs provide software control of a variety of modulation techniques, wide-band or narrow-band operation, communications security functions (such as hopping), and waveform requirements of current and evolving standards over

a broad frequency range. The frequency bands covered may still be constrained at the front-end requiring a switch in the antenna system.

- Tier 3 : **Ideal Software Radio (ISR)**
  ISRs provide dramatic improvement over an SDR by eliminating the analog amplification or heterodyne mixing prior to digital-analog conversion. Programmability extends to the entire system with analog conversion only at the antenna, speaker and microphones.
- Tier 4 : **Ultimate Software Radio (USR)**
  USRs are defined for comparison purposes only. It accepts fully programmable traffic and control information and supports a broad range of frequencies, air-interfaces & applications software. It can switch from one air interface format to another in milliseconds, use GPS to track the user location, store money using smartcard technology, or provide video so that the user can watch a local broadcast station or receive a satellite transmission.
  Cognitive radio (CR) is a form of wireless communication in which a transceiver can intelligently detect which communication channels are in use and which are not, and instantly move into vacant channels while avoiding occupied ones. This optimizes the use of available radio-frequency (RF) spectrum while minimizing interference to other.

1.4 GNU Radio

1.4.1 Concepts and architecture

GNU Radio is primarily developed using the GNU/Linux operating system, but, Mac OS and Windows are also supported. In GNU Radio, a radio system is represented as a directed signal flow graph where graph vertices are known as signal processing blocks and edges indicate a connection between the two blocks. Data flows in one direction from a signal source to one or more signal sinks. This construction of software radio is similar to development of hardware radios, but with an additional restriction that the signal flow in a flow graph cannot form a feedback cycle, so implementation of any feedback mechanisms must be contained within one signal processing block. In GNU Radio, the signal processing blocks are defined in C++ for performance, while the connections between the blocks for a given application are declared in Python. Using a high level language like Python allows users to quickly create different applications by constructing a signal flow graph simply by making connections between smaller building blocks. This approach meant that the agility of software development in a high level language can be maximized while at the same time sidestepping its drawback of slow performance by acting only as ‘glue’ code and offloading the heavy lifting to C++ compiled code. Interoperation and data marshalling between Python and C++ is done by employing the Simple Wrapper Interface Generator (SWIG). GNU Radio uses a number of data types to represent the signal at the interfaces of each of the signal processing blocks. The data type used by a particular block can usually be identified through the naming convention that each block should be suffixed with a code to represent its interface. For example, the block gr_rms_cf has a suffix of _cf, which indicates that the block takes input as 8 byte complex values, and produces an output in 4 byte floating point values. Similarly, the block gr_multiply_const_vss would take a vector of 2 byte short integers and produce an output in the same format. Other possible data types include b for 1 byte integers, and i for 4 byte integer values. A new GNU Radio signal processing block is defined

by deriving from the base class gr_block or one of its subclasses gr_sync_decimator, gr_sync_interpolator, gr_sync_block, or gr_hier_block2 in C++. Then, a SWIG interface is defined for this block, which enables it to be constructed and connected from Python. At the core of the signal processing block are two member functions: forecast() and work(). Forecast() returns an estimate of how many units of the input data is required for this module to produce a given number of output units and work() is the function that does the actual computation on the input data and produces an output. This signal processing blocks framework abstracts away the complexity of how one might schedule the work of multiple signal processing blocks on the computer.

1.4.2 GNU Radio installation

In this project Ubuntu 10.10 was used , and we are going to show steps of installation of GNU on it .

(1) Install the following pre-requisite packages from Terminal

```
sudo apt-get -y install libfontconfig1-dev libxrender-dev libpulse-dev \
swig g++ automake autoconf libtool python-dev libfftw3-dev \
libcppunit-dev libboost-all-dev libusb-dev fort77 sdcc sdcc-libraries \
libsdl1.2-dev python-wxgtk2.8 git guile-1.8-dev \
libqt4-dev python-numpy ccache python-opengl libgsl0-dev \
python-cheetah python-lxml doxygen qt4-dev-tools \
libqwt5-qt4-dev libqwtplot3d-qt4-dev pyqt4-dev-tools python-qwt5-qt4


sudo apt-get install git-core cmak
```

(2) #Download and install UHD from git:

```
git clone git://code.ettus.com/ettus/uhd.git
cd uhd/host
mkdir build
cd build
cmake ../
make
make test
sudo make install
cd
sudo gedit .bashrc
write this before  # append to the history file, don't overwrite it
```

LD_LIBRARY_PATH for uhd

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib

LD_LIBRARY_PATH

sudo ldconfig

(3) #Download and install GNURADIO from git:

git clone git://gnuradio.org/gnuradio.git

cd gnuradio/

mkdir build

cd build

cmake ../

make

make test

sudo make install

sudo ldconfig

cd

sudo gedit .bashrc

put it after UHD path

python path for gnuradio

export PYTHONPATH=$PYTHONPATH:/usr/local/lib/python2.6/dist-packages/

python path for gnuradio

sudo ldconfig

sudo apt-get install gnuradio-companion

then reboot Ubuntu

(4) Configure SD card by fpga& firmware

- format SD card in windows then open ubuntu again and insert it by reader or in laptop.
- cd /home/uhd/host/utils/
- open gui burner by sudo ./usrp2_card_burner_gui.py
- download the lateast release of uhd binary image and select image only then downlaod .tar then extract it in /home/uhd/host/utils/

- after open burner select usrp2_fpga.bin & usrp2_fw.bin and select /dev/sdb and burn it.
- unmount SD card and put it in usrp2.

(5) Configuration usrp2 and Open GRC

- configure eth0 (address=192.168.10.1,,subnet=255.255.255.0,,,gateway=192.168.10.2)
- sudo ping 192.168.10.2 (to check usrp2 send echo(response))
- sudo uhd_find_devices (Reading usrp2 and print some information (serial number , version,….)
- sudo uhd_usrp_probe (Reading name of Rf_daughterboard and range of frequencies that operate it)
- sudo gnuradio-companion

1.5 Tools

1.5.1 GRC

GRC is a graphical tool which provides a user interface that lets us create signal flow graphs and activates its source code. This graphical interface, by means of graphical blocks, allows us to set the input parameters which are taken by the source code of each block in order to generate a signal flow, and to visualize the signal at every step of the block chain using graphical sinks. There are mainly four kinds of blocks:

1- Source blocks: Their main functionality is to generate an output signal by means of some input parameters. For this reason, these blocks have no input signal. There are many types of sources, depending on the number of output ports, data type, vector lengths, etc.

2- Sink blocks: In this case, there is no output signal. Sink blocks receive an input signal with a specific data type and length, and, using certain input parameters, the input signal is stored in a vector, file or sent to a binded TCP1or UDP2 socket.

3- Operation blocks: These blocks use a configurable number of input signals with configurable data types, to produce a certain number of output signals with specific data types, using the input parameters to perform a certain operation on the samples at the input. These operations can be modulations or demodulations, coding operations, filters, synchronizations, type or stream conversions, etc. Among the different parameters needed to perform the operation, the sampling rate stands always out so that a correct treatment of the signal can be done.

4- Visualization blocks: These blocks can be classified as a type of sink block which generates a graphical output from the input signals. In this group of blocks, we can mention scopes to provide a time domain representation, FFT sink for a frequency domain screening, constellation plots, etc. The different blocks are connected in a proper way so that the signal data can flow along a chain, taking into account data

types, vector lengths, etc. The core functionality of each block is defined by Python or C++ code.

1.5.2 C++

Signal processing blocks process streams of data from their input port to their output port. The input and output ports of a signal process block are variable. So a block can have multiple outputs and multiple inputs. The signal processing blocks are written in C++.

1.5.3 Python

Don't get afraid of the name, stay tuned, the python will not hurt you . The story of this mysterious name is, at the same time he began implementing python, Guido van Rossum was also reading the published scripts from Monty Python's Flying Circus (a BBC comedy series from the seventies, in the unlikely case you didn't know). It occurred to him that he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python The most important thing in the programming language is the name. A language will not succeed without a good name. About Python,it is a script language is used to connect the signal processing blocks together. In Python the necessary signal sources, sinks and processing blocks are selected and configured with the correct parameters. The flow of data through the flow graph exists out of data in one of the following data-types:

_ Byte - 1 byte of data (8 bits)
_ Short - 2 byte integer
_ Int - 4 byte integer
_ Float - 4 byte floating point
_ Complex - 8 bytes

As mentioned previously, all sources, sinks and blocks are implemented as classes in C++. This results, no matter how complicated the radio is, in good readable Python code. The real heavy load is done in C++. Figure 1.2 shows an example of a Python code . As you can see it is not difficult to create a radio in Python.

```
#!/usr/bin/env python

from gnuradio import gr
from gnuradio import audio
def build_graph ():
sampling_freq = 48000
ampl = 0.1
fg = gr.flow_graph ()
src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
dst = audio.sink (sampling_freq)
fg.connect ((src0, 0), (dst, 0))
fg.connect ((src1, 0), (dst, 1))
return fg
if __name__ == '__main__':
fg = build_graph ()
fg.start ()
raw_input ('Press Enter to quit: ')
```

fg.stop ()

We start by creating a flow graph to hold the blocks and connections between them. The two sine waves are generated by the gr.sig source _f calls. The fsu_x indicates that the source produces floats. One sine wave is at 350 Hz, andthe other is at 440 Hz. Together, they sound like the US dial tone. audio.sink is a sink that writes its input to

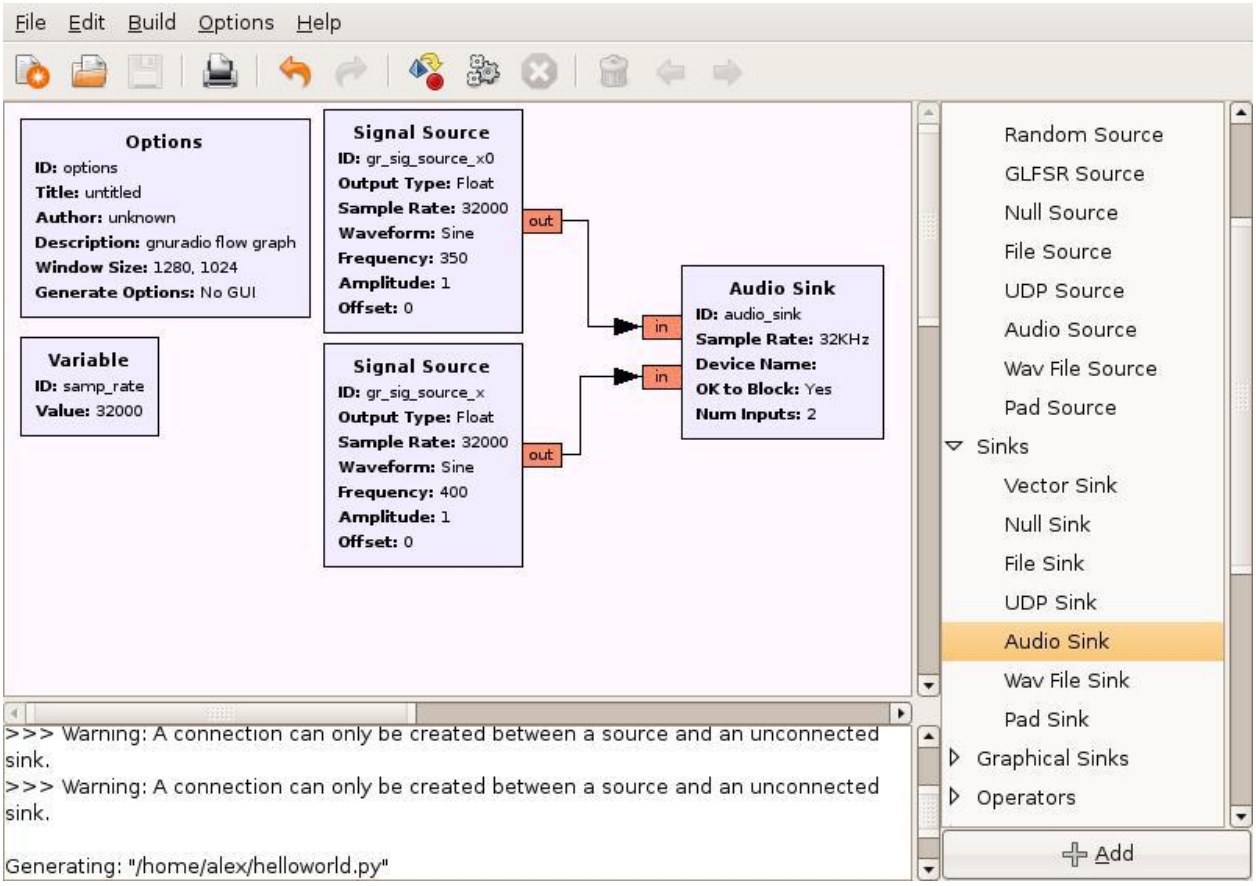


Figure 1.2: Example of a Python code

the sound card. It takes one or more streams of oats in the range -1 to +1 as its input. We connect the three blocks together using the connect method of the flow graph.connect takes two parameters, the source endpoint and the destination endpoint,and creates a connection from the source to the destination. An end point has two components: a signal processing block and a port number. The port number specifies which input or output port of the specified block is to be connected. In the most general form, an endpoint is represented as a python tuple like this: (block, port number). When port number is zero, the block may be used alone. These two expressions are equivalent:fg.connect ((src1, 0), (dst, 1)) and fg.connect (src1, (dst, 1)) . Once the graph is built, we start it. Calling start forks one or more threads to run the computation described by the graph and returns control immediately to the caller. In this case, we simply wait for any keystroke.

1.5.4 SWIG

It is the wrapper for the C++ modules and generates the corresponding Python code and library so that these classes and functions can be called from Python. Most default signal blocks are already created in the GNU Radio, or by third parties. So you only touch the C++ environment to create your own special signal processing blocks. It requires some experienced C++ skills to create these blocks. If you do not have much C++ experience try to use signal blocks that are already part of the GNU Radio library, or try to find code from third parties.

1.6 USRP

1.6.1 Motivation

The Universal Software Radio Peripheral, or USRP (pronounced "usurp") was designed as a low cost board solely for the purpose of running GNU radio applications and allowing general purpose computers to function as high bandwidth software radios. . Fully developed by Matt Ettus, it is a very flexible platform and can be used to implement real time applications . In essence, it serves as a digital baseband and IF section of a radio communication system. You may say , it is the bridge between the software world and the RF world .The basic design philosophy behind the USRP has been to do all of the waveform specific processing, like modulation and demodulation, on the host CPU. All of the high-speed general purpose operations like digital up and down conversion, decimation and interpolation are done on the FPGA. The true value of the USRP is in what it enables engineers and designers to create on a low budget and with a minimum of effort. A large community of developers and users have contributed to a substantial code base and provided many practical applications for the hardware and software. The powerful combination of flexible hardware, open-source software and a community of experienced users make it the ideal platform for your software radio development.

1.6.2 Motherboard and internal construction

Figure 1.3 shows a typical graph for USRP Motherboard . The USRP has 4 high-speed analog to digital converters (ADCs), each at 12 bits per sample, 64MSamples/sec. There are also 4 high-speed digital to analog converters (DACs), each at 14 bits per sample, 128MSamples/sec. These 4 input and 4 output channels are connected to an Altera Cyclone EP1C12 FPGA. The FPGA, in turn, connects to a USB2 interface chip, the Cypress FX2, and on to the computer. The USRP connects to the computer via a high speed USB2 interface only, and will not work with USB1.1. So in principle, we have 4 input and 4 output channels if we use real sampling. However, we can have more flexibility (and bandwidth) if we use complex (IQ) sampling. Then we have to pair them up, so we get 2 complex inputs and 2 complex outputs.. The USB controller contains the firmware that defines its behavior and the USB endpoints. The firmware also takes care of loading the FPGA bit stream. The FPGA handles the high bandwidth computations and reduces the data rate to something we can send over the USB 2.0. The Analog Device chip is a mixed signal

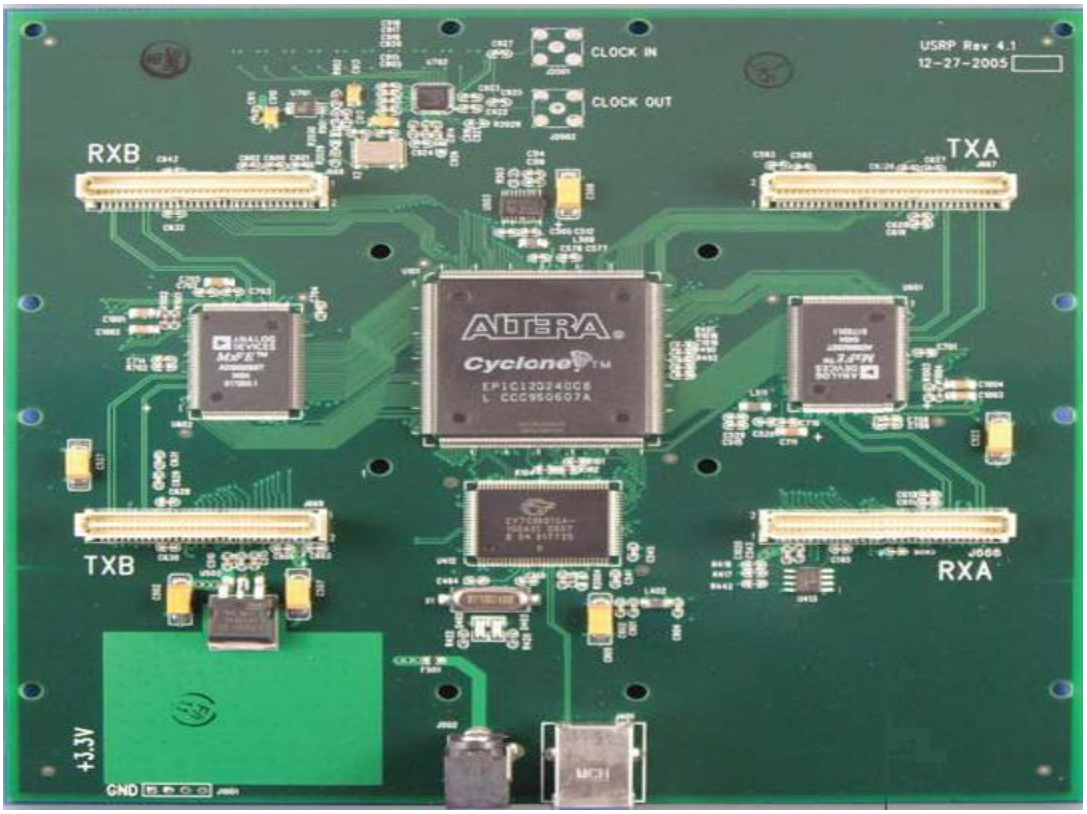


Figure 1.3: USRP Motherboard

processor that takes care of the conversion between analog and digital signals, digital up conversion in the transmit path and interpolation/decimation of the signals. The motherboard can have up to 4 daughterboards, two for receive and two for transmit to achieve wireless communication at different frequencies . They consist of the RF front end where the signal is up converted from the intermediate frequency to the carrier frequency or vice versa for the received signal. And The following figure is a high level view of the USRP.

Figure 1.4: High level view of USRP

1.6.3 From USRP1 to USRP2

USRP1 and USRP2 provide the hardware platform for SDR in order to receive and transmit the signal .Figure 1.5 shows a typical USRP2 board .

Figure 1.5: USRP 2

The USRP motherboard was discussed before. For USRP 2 , it is built on the success of USRP1 but it is not meant to replace USRP1. Motherboard of USRP2 is shown in figure 1.6 .The following new features are added to USRP2:

- Gigabit Ethernet interface
- 25 MHz of instantaneous RF bandwidth
- Xilinx Spartan 3-2000 FPGA
- Dual 100 MHz 14-bit ADCs
- Dual 400 MHz 16-bit DACs
- 1 MByte of high-speed SRAM
- Locking to an external 10 MHz reference
- 1 PPS (pulse per second) input
- Configuration stored on standard SD cards
- Standalone operation
- The ability to lock multiple systems together for MIMO
- Compatibility with all the same daughter boards as the original USRP
- Configuration: flash SD-card

Figure 1.6: USRP 2 and its motherboard

And the following table provides comparison between USRP1 , USRP2 and USRP N210 :

	USRP 1	**USRP 2**	**USRP N210**
Interface	USB 2.0	Gigabit Ethernet	Gigabit Ethernet
FPGA	Altera EP1C12	Xilinx Spartan 3 2000	Xilinx Spartan 3A-DSP 3400
RF Bandwidth to / from host	8 MHz @ 16 bits	25 MHz @ 16 bits	25 MHz @ 16 bits
Cost	$700	$1400	$1400
ADC Samples	12 bit , 64 MS/s	14 bit , 100 MS/s	14 bit , 100 MS/s
DAC Samples	14 bit , 128 MS/s	16 bit , 400 MS/s	16 bit , 400 MS/s
Daughterboard capacity	2 TX , 2 RX	1 TX ,1 RX	1 TX ,1 RX
SRAM	None	1 Megabyte	1 Megabyte
Power	6V , 3A	6V , 3A	6V , 3A

Table 1.1: Comparison between USRP1 , USRP2 and USRP N210

As our project is implemented on USRP2 and USRP N210 , figure shows the internal construction and blocks of USRP2 or USRP N210 as there is no big differences except some improvements in USRP N210 over USRP 2 such as more capable FPGA and reprogrammable over Ethernet , instead of SD card (in USRP 2)

.

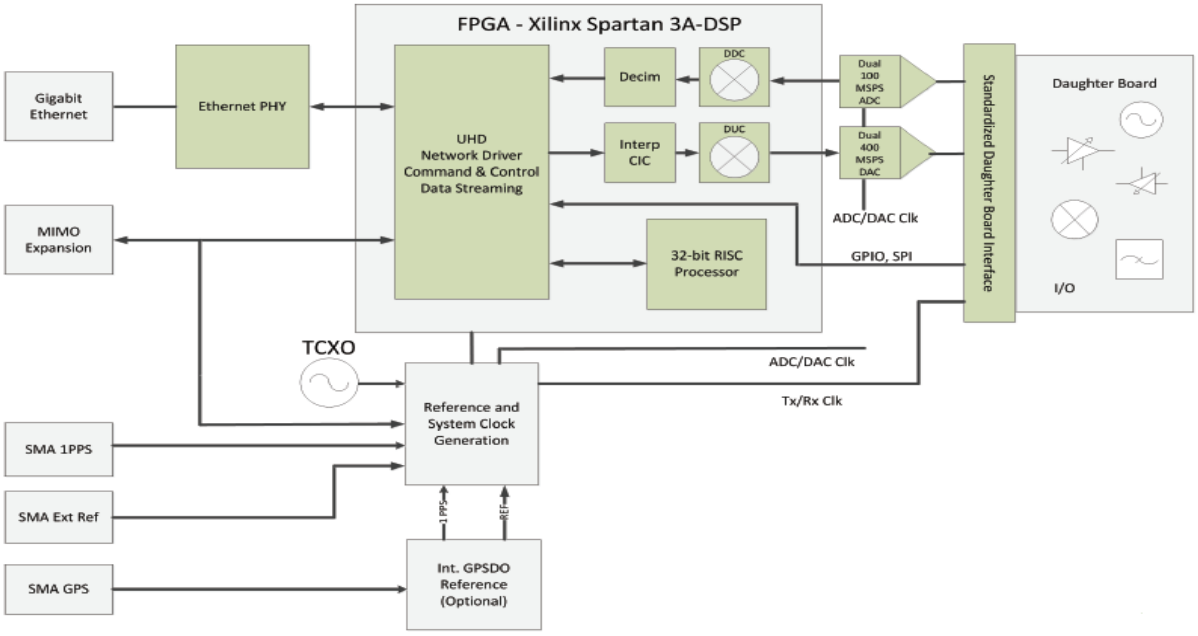


Figure 1.7: Internal construction of USRP 2

1.6.4 Different sections in USRP

We will discuss shortly some important parts inside USRP and USRP2 such as ADC,DAC, FPGA and daughter boards .

1- ADC Section

There are 4 high-speed 12-bit ADC converters. The sampling rate is 64M samples per second. In principle, it could digitize a band as wide as 32MHz. For USRP2 , there are two high-speed 14-bit ADC (of type LTC2284 used at 100 MS/s) . There is two other auxiliary ADCS(of type AD7922 used at 100 MS/s) for each daughter board connector . Giga Ethernet can sustain 1 gigabit/s . So, this is max 800 MB/s given integer decimation and a 100 MHz DSP clock.

2- DAC Section

At the transmitting path, there are also 4 high-speed 14-bit DA converters. The DAC clock frequency is 128 MS/s, so Nyquist frequency is 64MHz. For USRP 2 , The DAC clock frequency is 400 MS/s, so Nyquist frequency is 200 MHz .USRP 2 has main DAC (Dual of type AD9777 used at 400Ms/s)and two auxiliary DACs (of type AD5623)

3- FPGA

Now let's have a look inside the FPGA in order to understand the functionality of each of its building blocks. Probably understanding what goes on the USRP/USRP 2 FPGA is the most important part for the GNU Radio users. All the ADCs and DACs are connected to the FPGA. This piece of FPGA plays a key role in the USRP/USRP 2 system. Basically what it does is to perform high bandwidth math, and to reduce the data rates to something you can squirt over USB2.0 /GE ON USRP/USRP 2 respectively .The standard FPGA configuration includes digital down converters (DDC) implemented with 4 stages cascaded integrator-comb (CIC) filters. Also, it includes digital up converters (DUC) implemented with 4 stages cascaded integrator-comb (CIC) filters . CIC filters are very high-performance filters using only adds and delays. The DDC and DUC each contain 2 halfband filters. The high rate one has 7 taps and the low rate one has 31 taps For spectral shaping and out of band signals rejection

4- Daughter boards

On the mother board there are two slots . One of these slots is for TX and the other is for RX . Each daughter board slot has access to ADC/DAC . The daughter boards are used to hold the RF receiver interface or tuner and the RF transmitter.

Every daughterboard has an I2C EEPROM (24LC024 or 24LC025) onboard which identifies the board to the system. This allows the host software to automatically set up the system properly based on the installed daughterboard. The EEPROM may also store calibration values like DC offsets or IQ imbalances. If this EEPROM is not programmed, a warning message is printed every time USRP software is run. There are several kinds of daughter boards available now such as :

A)- Basic TX/RX Daughterboards

Each has two SMA connectors that can be used to connect external up/down tuners or signal generators. We can treat it as an entrance or an exit for the signal without affecting it. Some form of external RF front end is required. The ADC inputs and DAC outputs are directly transformer-coupled to SMA connectors (50Ω impedance) with no mixers, filters, or amplifiers. The BasicTX and BasicRX give direct access to all of the signals on the daughterboard interface. Each of the Basic TX/RX boards has logic analyzer connecters for the 16 general purpose IOs. These pins can be used to help debugging your FPGA design by providing access to internal signals.

B)- Low Frequency TX/RX Daughterboards

The LFTX and LFRX are very similar to the BasicTX and BasicRX, respectively, with 2 main differences. Because the LFTX and LFRX use differential amplifiers instead of transformers, their frequency response extends down to DC. The LFTX and LFRX also have 30 MHz low pass filters for anti-aliasing.

C)- TVRX Daughterboards

This is a receive-only daughter board. It is a complete VHF and UHF receiver system based on a TV tuner module. The RF frequency ranges from 50MHz to 860MHz, with an IF bandwidth of 6MHz. All tuning and AGC functions can be controlled from software. Typical noise figure is 8 dB. This board is the only USRP daughterboard which is NOT MIMO capable.

D)- DBSRX Daughterboards

Similar to the TVRX board, this is also a receive-only. It is a complete receiver system for 800 MHz to 2.4 GHz with a 3 -5 dB noise figure. The DBSRX features a software controllable channel filter which can be made as narrow as 1 MHz, or as wide as 60 MHz. The DBSRX is MIMO capable, and can power an active antenna via the SMA.

E)- RFX Daughterboards

The RFX family of daughterboards is a complete RF transceiver system. They have Independent local oscillators (RF synthesizers) for both TX and RX which enables a split-frequency operation. Also, it has a built-in T/R switching and signal TX and RX can be on same RF port (connector) or in case of RX only, we can use auxiliary RX port. Most boards have built-in analog RSSI measurement. All boards are fully synchronous design and MIMO capable. Table 1.2 shows USRP daughter boards currently available from Ettus research .

Name	Functionality	Frequency range (MHz) From : To	Cost	Minimum power (mw)
Basic Rx	Receiver	2 to 300	$ 75.00	------
Basic Tx	Transmitter	2 to 200	$ 75.00	------
LFRX	Receiver	0 to 30	$ 75.00	------
LFTX	Transmitter	0 to 30	$ 75.00	------

TVRX	Receiver	50 to 70	$ 100.00	------
DBSRX	Receiver	800 to 2400	$ 150.00	------
RFX400	Transceiver	400 to 500	$ 250.00	------
RFX900	Transceiver	800 to 1000	$ 275.00	200
RFX1200	Transceiver	1150 to 1450	$ 275.00	200
RFX1800	Transceiver	1500 to 2100	$ 275.00	100
RFX2400	Transceiver	2300 to 2900	$ 275.00	50

Table 1.2: USRP daughterboards available from Ettus research

1.6.5 General Notes

These notes are collected from our experience with USRP boards and may be helpful for you .

- Sometimes while using USRPs , "O" "U" "u" "a" characters appear on the screen when you run gnuradio program. It appears conceptually when data flows from USRP to PC is stopped or something near that. And now , we will show the meaning of these characters :
    - "u" = USRP
    - "a" = audio (sound card)
    - "O" = overrun (PC not keeping up with received data from USRP or audio card)
    - "U" = underrun (PC not providing data quickly enough)
    - “aUaU” = audio underrun (not enough samples ready to send to sound card sink)
    - “uUuU” = USRP underrun (not enough sample ready to send to USRP sink)
    - “uOuO” = USRP overrun (USRP samples dropped because they weren't read in time.

  A faster machine will generally cure this problem. This assumes that you are not asking the USB/GE (for USRP/USRP 2, respectively) to do something that it can't.

- RISC processor used in USRP 2 is ZPU .
- Minimum interpolation and decimation rates are 4 and maximum is 512.
- The sampling rate is very important factor in implementations through the USRP 2 .
- The DC component is blocked (by USRP 2) to remove LO leakage. The best way to work around this is to use the advanced tuning parameters to offset the

LO away from your signal; this will move the LO (and thus the DC offset correction) outside the band of interest.

- Sometimes , the gain setting in the USRP 2 sources/sinks is important . It controls the gain of the RF frontend itself; both settings are necessary for proper operation in your application.
- In all our practical communication systems done in this project , we used daughter boards : Basic TX and Basic RX , RFX 1800 and RFX 2400 .

Chapter 2: Analog Environment

2.1 Motivation

We are going in this phase of the project to show practical analog implemented on USRP boards under GRC. Successful transmission and reception of single tone is successfully accomplished .Connecting received tone to audio sink enabled us to distinguish between frequencies according to intensity of pitch. We will discuss transmission path as well as reception path in details in the coming sub-sections.

2.2 Transmission path

For transmission path, the scenario set up is basically constituted of three main different blocks, signal source, WBFM transmitter , and USRP sink as shown in figure 2.1. Now, we will discuss the operation of each block.

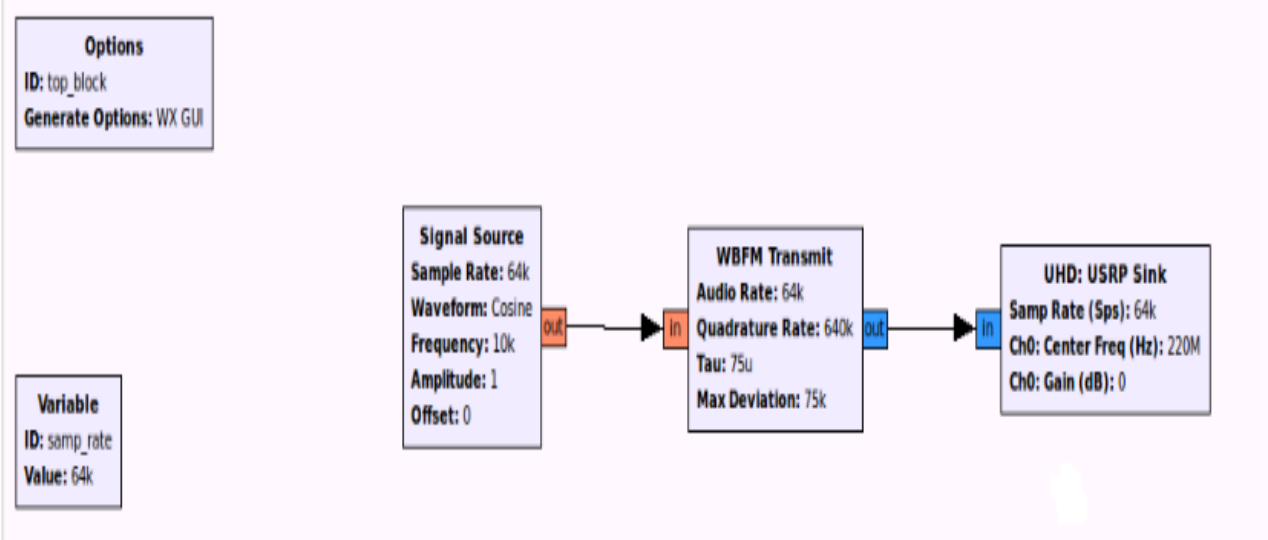


Figure 2.1:Single tone transmission path

2.2.1 Signal source

Simply, it is just a generator generates different types of signals such as , sine and cosine waves , square wave and triangle wave . Here, in our case we generated cosine signal with frequency 10 KHz.

2.2.2 Wide band FM transmitter

It is defined in python file *wfm_tx.py*. It has one input, audio samples, and one output, modulated signal. It takes a single float input stream of audio samples in the range [-1,

1] and produces a single FM modulated complex baseband output based on some parameters such as audio rate and quadrature rate.

2.2.3 USRP sink

The sink used here is taking the baseband complex sampled signal at the modulator block output in order to transmit it through the GE to the USRP 2 motherboard. Basically, the parameters set the transmitting frequency to which the baseband carrier is going to be up-converted.

2.3 Reception Path

To evaluate the USRPs performance when receiving, we set six main blocks as it can be seen in figure 2.2.

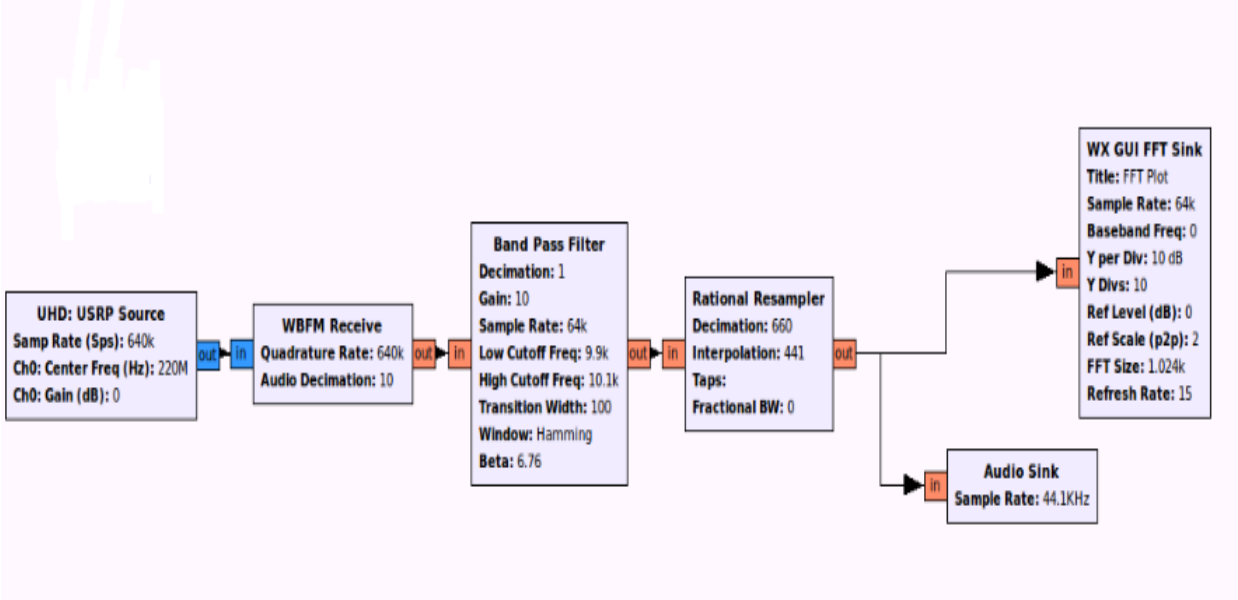


Figure 2.2: Single tone reception path

2.3.1 USRP source

This block, as the beginning of the chain, provides us the received signal coming through the GE link from the USRP 2 motherboard. This signal is a complex digitalized signal with a sample rate of 640 kHz, down-converted to baseband.

2.3.2 Wide band FM receiver

It's defined in a python file *wfm_rcv.py*. It has one input, down converted base band signal and one output; the demodulated audio. It has many blocks implemented in C++, demodulating block, de-emphasizer and audio filter.

1- Demodulating

The first block in the wide band FM receiver chain is fm_demod, an instance of gr.quadrature_demod_cf. To understand the work done within it, we should know about how FM signals are generated. With FM, the instantaneous frequency of the transmitted waveform is varied as a function of the input signal. The instantaneous frequency at any time is given by : f(t) = k*m(t) + fc . Where , m(t) is the input signal, k is a constant that controls the frequency sensitivity and fc is the the carrier frequency .To recover m(t), two steps are needed. First we need to remove the carrier fc, which leaves us with a baseband signal that has an instantaneous frequency proportional to the original message m(t). The second step is to compute the instantaneous frequency of the baseband signal. Thus, our challenge is to find a way to remove the carrier and compute the instantaneous frequency. Removing the carrier has been done on the FPGA via the digital down converter (DDC). Thus the signal coming into the 'guts' has already become a baseband signal and the remaining task is to calculate its instantaneous frequency. If we integrate frequency, we get phase or angle. Conversely, differentiating phase with respect to time gives frequency. These are the key insights we use to build the demodulator. We used the gr.quadrature_demod_cf block for computing the instantaneous frequency of the baseband signal. We approximate differentiating the phase by determining the angle between adjacent samples.

2- De-emphasizer

The second block in the chain is the deemphasizer, deemph is an instance of the class fm_deemph. fm_deemph is also a hierarchical block defined in *fm_emph.py*.

What is de-emphasis? Let's introduce it briefly. It has been theoretically proven that, in an FM detector, the power of the output noise increases with the frequency quadratically. However, for most practical signals, such as human voice and music, the power of the signal decreases significantly as frequency increases. As a result, the signal-to-noise ratio (SNR) at the high frequency end usually becomes unacceptable. To circumvent this effect, people introduce 'pre-emphasis' and 'de-emphasis' into FM systems. At the transmitter, we use proper pre-emphasis circuitry to manually amplify the high frequency components and do the reverse operation at the receiver to recover the original power distribution of the signal. As a result, we effectively improve the SNR.

In the analog world, a simple first order RLC circuit usually suffices for pre-emphasis and de-emphasis.

3- Audio Filter

Maybe you are wondering where we 'pick out' the station of interest from the digitized frequency band. Actually, this is done explicitly by the channel filter in wfm_rcy.py and implicitly by the digital down converter (DDC) on the USRP. Recall that DDC can be regarded as a low pass FIR filter followed by a down sampler. As a result of these two operations, the target station is picked out then spread out in the digital spectrum after decimation. Because we choose an appropriate decimation rate and channel filter bandwidth, we have isolated our station of interest!
To keep our life simple, we just design a mono receiver, using a low pass filter to select only channel signal.

2.3.3 Band pass filter

We used it to filter the received signal into our band only , and hence correction reception is guaranteed . The parameters inside this block can be controlled manually such as low cut off frequency of the filter , high cut off frequency of the filter and transition BW .

2.3.4 Rational resampler

To reach the sample rate of audio sink in used PC . we used this block to do this task , thus we can enter the audio sink with adequate sample rate .

2.3.5 Audio sink

This block is responsible for listening to frequency of the sent tone . Its sample rate differs from one PC to another , that’s why we used a rational resampler block before it .

2.3.6 FFT sink

This block simply displays the received tone in frequency domain As expected for a single tone , we received two approximately deltas at frequency and its counterpart of sent tone .

Chapter 3: Digital Environment

3.1 Motivation

We are going in this phase of the project to show practical digital modulation scheme(s) implemented on USRP boards under GRC. Successful transmission and reception of bits (or packets) is successfully accomplished with reasonable BER. We will discuss transmission path as well as reception path in details in the coming sub-sections.

3.2 Transmission Path

For transmission path, the scenario set up is basically constituted of three main different blocks, random /vector source, DPSK modulator , multiply const and and USRP sink as shown in figure 3.1. Now, we will discuss the operation of each block.

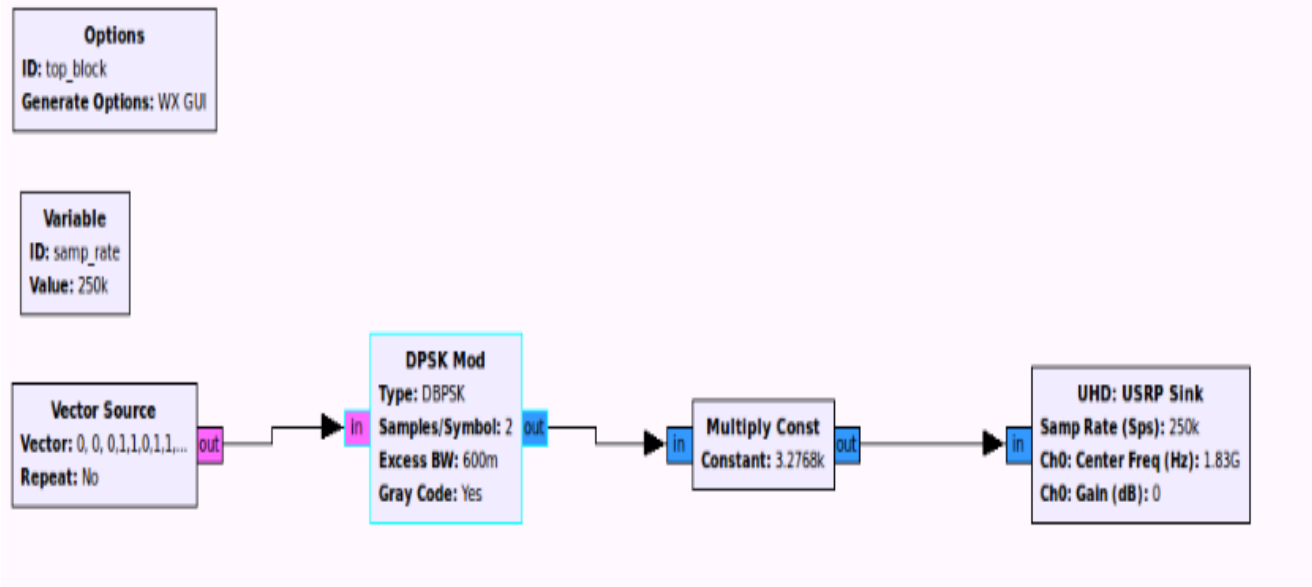


Figure 3.1: DBPSK transmission path

3.2.1 Random/Vector source

Random source: This block, as the beginning of the chain, generates a random digital signal. It provides us a certain number of byte type samples, which values range from 0 to, in our case, 128 – this number is not significant for the meaning of the measurements .Since it is a data source, it has no data input but one output port. As the output are bytes, it is necessary to perform a unpacking from bytes to a 1-bit vector, i.e. groups of 1-bit chunks, as the number of bits per symbols is 1, to treat the signal bits properly in the demodulation process. However, In order to understand the behavior of the modulation block, the random source is replaced by a "vector source" so that we can introduce a known binary data.

3.2.2 DBPSK modulator

Basically, this block takes care of the root raised cosine filtering and performs a phase shift modulation. The input data consists of a byte stream coming from the vector source and the output is a complex modulated signal at baseband. The complete functionality of the modulator block can be easily divided in different five sub-blocks (functions) defined in C++ code and connected with Python code. The parameters that are necessary here are the number of samples per symbol and the excess in bandwidth that refers to the roll-off factor of the root raised cosine filter. These parameters will be explained on greater detail in this section. The signal is flowing in a chain of sub-blocks, as shown in figure 3.2.

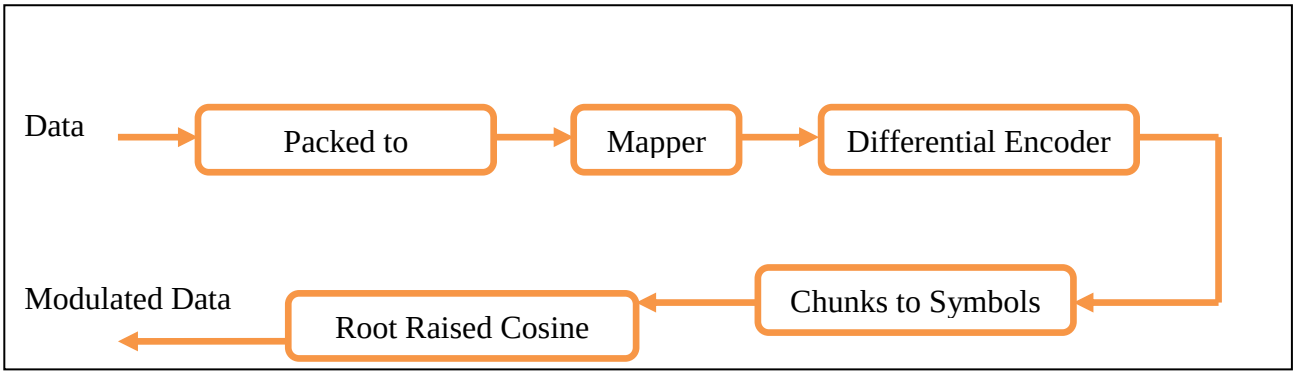


Figure 3.2: Data flow through the modulation block

Note that the vector source generates a binary sequence from a list of byte type numbers with no predefined timing. The timing is set at the USRP sink and the demodulator block, by means of the sample rate and the factor of samples/symbol. That is to say, for a sample rate of 250 kHz at the USRP sink and a factor of 2 samples/symbol for the interpolation filter at the modulation block, the symbol rate of the source is defined, and its value is 125 ksymbols/s ($\frac{250\ \text{Ksamples/s}}{2\ \text{samples/symbol}} = 125\ \text{ksymbols/s}$). The filter considered for any possible demodulation is a root raised cosine with a roll-off factor of 0.6. For that reason, the input parameters are DPSK and an excess bandwidth of 0.6 to match it the signal filter. The number of samples/symbol is set to 2 to provide a signal with symbol rate of 125 ksymbols/s.

1- Packed to unpacked

Since the signal at the output from the vector source are packed bytes, it is necessary to convert it into a 1-bit-chunk stream (It is for BPSK, what about another modulation schemes? . Left for the reader) in order to perform a proper identification and treatment of the symbols. Data type is still being byte type at the output. It exists in file: *packed_to_unpacked_bb.cc*.

This sub-block converts the data based on some parameters such as bits per chunk or Endianness (LSB or MSB). e.g.: If the data of vector source is 101 and is inputted to this sub-block (with 1 bit per chunk and LSB parameters), the output must be as shown below in figure 3.3.

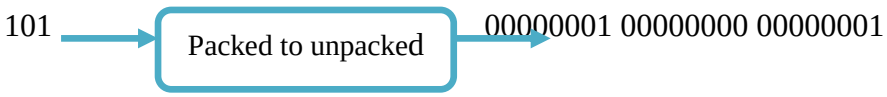


Figure 3.3: Example showing input and output of packed to unpacked sub-block

2- Mapper

After the unpacking, the mapping process on the chunks is executed. In this sub-block a binary sequence is mapped to symbols. As we deal with BPSK, it maps the 0 and 1 to 0 to 1 respectively. But its benefit will appear in the case of QPSK with gray code chosen, the mapping to the 2-bit chunks of 11 and 10 corresponds to 2 and 3, respectively, while 00 and 01 are mapped to 0 and 1. It exists in file: *map_bb.cc*

3- Differential Encoder

A stream of chunks is encoded differentially by

$$Y_{n+1} = Y_n \oplus b_{n+1}$$

In a differential modulation, the different binary chunks add a certain change in phase to the current phase in the carrier signal. That means that the previous phase is taken into account. It exists in file: *diff_encoder_bb.cc*

Example: When the input chunks (1-bit) to this sub-block is 10110100, the output will be as shown below in figure 3.4.

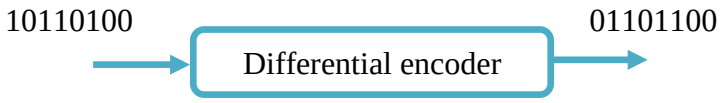


Figure 33.4: Example showing input and output of differential encoder sub-block

4- Chunks to Symbols

It maps a stream of chunks (unpacked bytes) to stream of float or complex constellation points. The combination of packed to unpacked sub-block followed by this sub-block handles the general case of mapping from a stream of bytes into arbitrary float or complex symbols. It exists in file : *chunks_to_symbols_bc.cc*

5- Root Raised Cosine Filter

Once the modulated signal is obtained, a raised cosine filtering is carried out. This filter is required to minimize the ISI, which causes a smearing into adjacent time slots due to time-spreading in a real system. At the output we have the complex filtered samples of a baseband carrier ready to be transmitted. It exists in file: *root_raised_cosine.cc*

3.2.3 Multiply const

This block as its name reveals is responsible for multiplying the signal by a constant. It simply increases the amplitude of the digital data (or any data generally) sent.

3.2.4 USRP Sink

The sink used here is taking the baseband complex sampled signal at the modulator block output in order to transmit it through the GE to the USRP 2 motherboard. Basically, the parameters set the transmitting frequency to which the baseband carrier is going to be up-converted.

3.3 Reception Path

To evaluate the USRPs performance when receiving, we set three main blocks as it can be seen in figure 3.5.

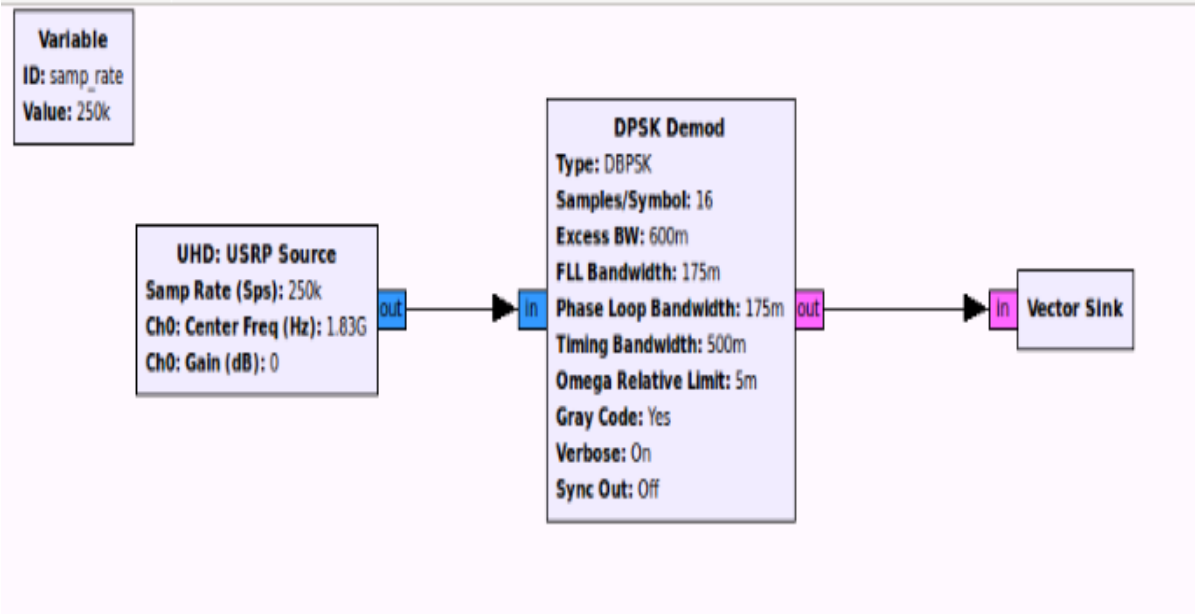


Figure 3.5: DBPSK reception path

3.3.1 USRP source

This block, as the beginning of the chain, provides us the received signal coming through the GE link from the USRP 2 motherboard. This signal is a complex digitalized signal with a sample rate of 250 kHz, down-converted to baseband.

3.3.2 DPSK demodulator

This block takes care of the root raised cosine filtering and performs a differential coherent detection or phase shift demodulation. The input data consists of a complex sampled signal at baseband frequency and the output is a big-endian stream of bits packed 1 bit/Byte .The parameters that are necessary here are the number of samples per symbol, the excess in bandwidth that refers to the roll-off factor of the root raised cosine filter, the Costa's alpha parameter, the mu factor and its gain, etc. These parameters will be explained on greater detail later. As shown in figure 3.6. , the block diagram of the DPSK demodulator consists internally of seven sub-blocks, each will be discussed in detail.

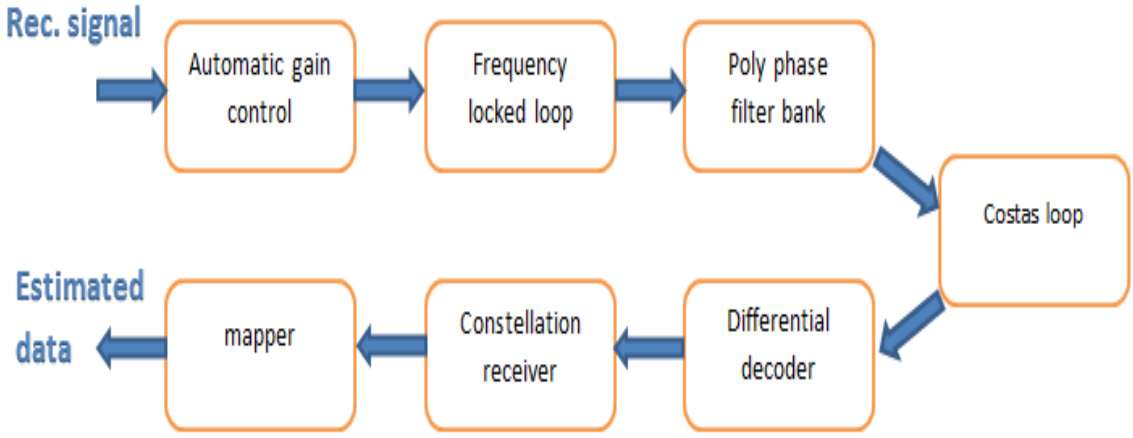


Figure 3.6: Data flow through the demodulation block

1- AGC

It stands for Automatic Gain Control. In this block the signal is first multiplied by a constant to scale the signal from full-range to ±1 . Then an automatic gain control is performed. The procedure consists of a calculation of the maximum power level among the samples. Until this point, a simple rescaling is performed . The AGC works by measuring a smoothed power history, then adjusting an output gain to achieve constant power. Its important parameters are:

- Attack rate: How fast the AGC decreases the gain when a loud signal appears.
- Decay rate: How fast the AGC increases the gain when the loud signal is gone.
- Reference: This is the level the AGC will try to maintain.
- Gain: The overall gain of the AGC.
- Max gain: The maximum gain the AGC can have.

Conclusion, in short, from C++ algorithm we can say that AGC takes an input complex sample and multiplies it by a gain parameter. Then, it calculates the output sample power. It makes a control loop to check signal power reached the desired power represented in "Reference" parameter or not, till reaching it. In addition , it exists in the file : *gr_agc2_cc.cc* .

2-FLL

It stands for Frequency Locked Loop. Its main aim is carrier recovery using band edge filter. But how this is done? It is based on the theory of carrier recovery implemented in C++ algorithm. We use a maximum likelihood frequency estimator to compensate frequency offset in the received signal. We construct a filter that is the derivative of the matched filter in the frequency domain. Note that the derivative is zero everywhere except in the transition band (controlled by roll off factor) of the filter , seeing that the non-zero spectral response of the derivative-matched filter spans the band edges of the matched filter .Then product the input spectrum of the received data that contain some frequency offsets with band edge filter and monitor the two spectral segments in the output of the band edge filter then comparing the absolute value squared of the two segments which providing error signal used to adjust the spectrum of the modulated data. The carrier recovery process is also shown in figure 3.7 for more clarification.

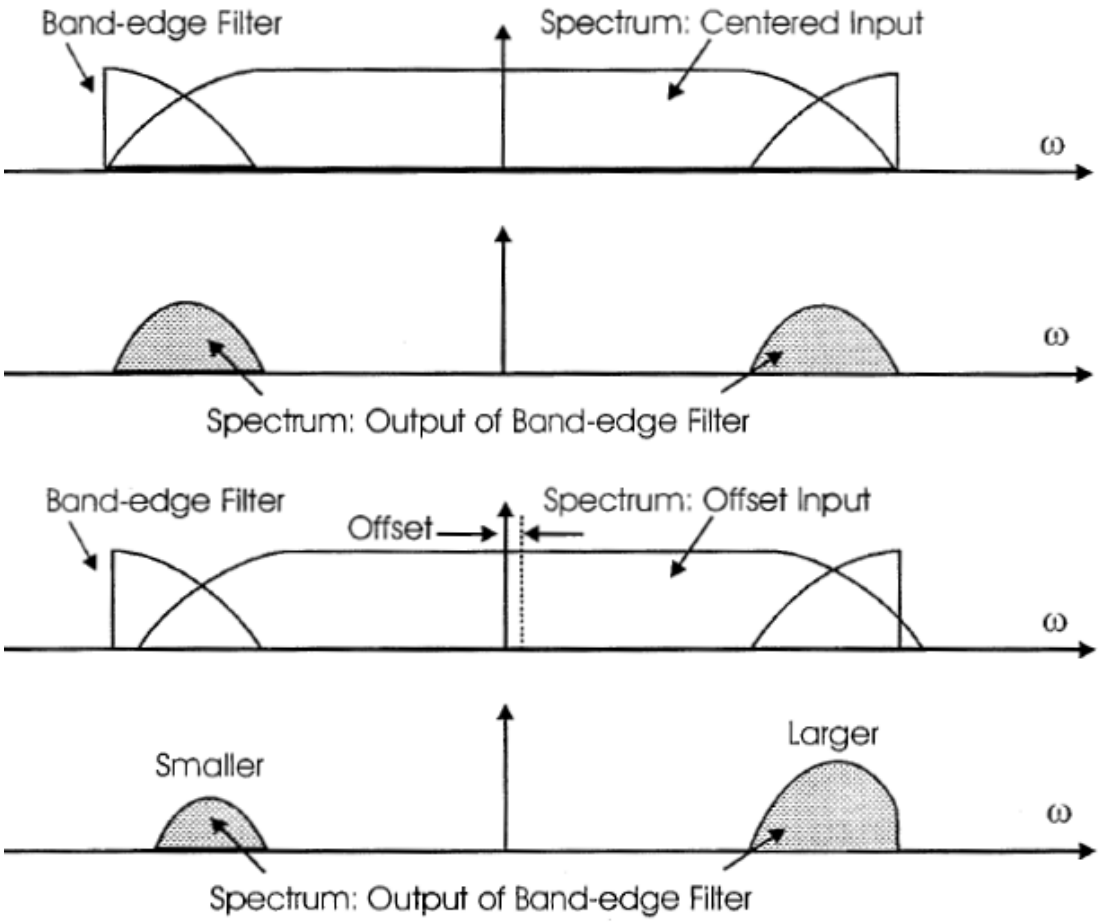


Figure 3.7 :Carrier recovery process

Its important parameters are:

- Samples per symbol: Sets the number of samples per symbol the system should use. This value is used to calculate the filter taps.
- Filter Roll off Factor: This parameter sets the roll off factor that is used in the pulse shaping filter and is used to calculate the filter taps.
- Prototype Filter Size: This sets the number of taps in the band-edge filters. This should be about the same number of taps used in the transmitter's shaping filter and also not very large. A large number of taps will result in a large delay between input and frequency estimation, and so will not be as accurate. Between 30 and 70 taps is usual. In this implementation use 55.
- Loop Bandwidth: Sets the loop filter bandwidth in control process. This should be between $\frac{2\pi}{200}$ and $\frac{2\pi}{100}$ (in rads/sample). It must also be a positive number.

Now, what about tasting the flavor of the communication algorithm implemented on C++ (in the file : *gr_fll_band_edge_cc.cc*) :

- First, We form band edge filter which the derivative of the matched filter in frequency by :
    - Take a quarter sine wave at the point of the matched filter's roll off (if it's a raised-cosine, the derivative of a cosine is sine)
    - Then extend this sine wave by another quarter wave to make a half wave around the band-edges is equivalent in time to the sum of two sinc functions , as shown in figure above .
    - After creating this baseband filter which its bandwidth is determined by the excess bandwidth (e.g., roll off factor) of the modulated signal , we form band edge filters by not only spinning the baseband filter up and down to the right places in frequency but also by normalizing the power in the two filters.
- Performing the dot product of the output modulated samples with the two filters.
- The error signal is the difference between energies of the upper and down output from the result of the dot product.
- Then, adjust the spectrum of modulated data in the right frequency by entering the error signal in the control loop to correct the frequency offset.
- Maximum and minimum frequencies that adjust by error signal in the control loop are 2*pi*(2/samps_per_sym) and -2*pi*(2/samps_per_sym) respectively

Oh, the C++ flavor with some communications algorithms looks different. We expect your smile right now.

3-Poly phase filter bank

Its main aim is timing recovery. It exists in the file : *gr_pfb_clock_sync_ccf.cc* .It is expected that samples of the received signal are not aligned with the data clock used to generate the analog waveform. Our approach for solving the timing recovery problem works by setting up two filter banks ; one filter bank contains the signal's pulse shaping matched filter (such as a root raised cosine filter), where each branch of the filter bank contains a different phase of the filter. Figure 3.8 shows a typical four phase poly-phase taps.
The second filter bank contains the derivatives of the filters in the first filter bank. Thinking of this in the time domain, the output of the first filter bank is shaped as the autocorrelation function. We want to align the output signal to be sampled at exactly the peak of the autocorrelation function.

.

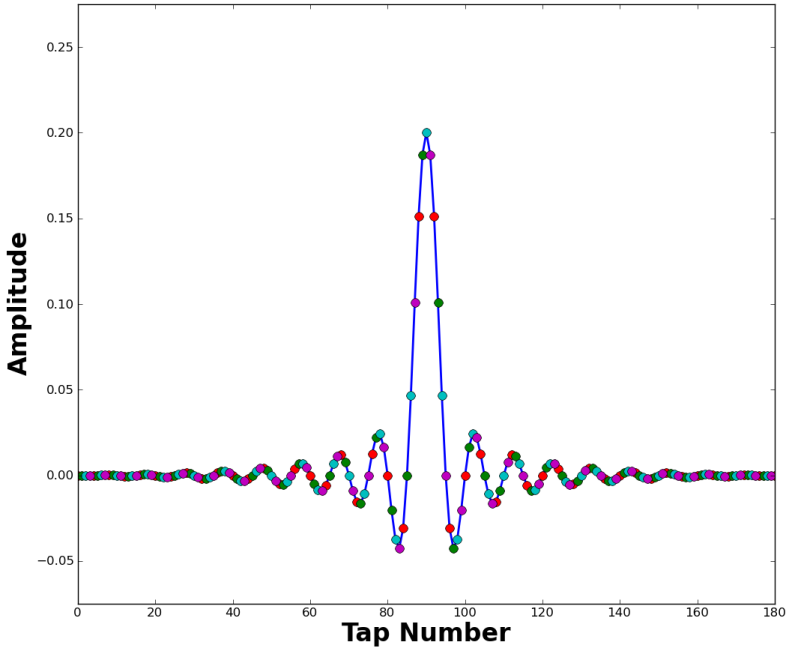


Figure 3.8 Four-phase poly-phase taps

The derivative of the autocorrelation contains a zero at the maximum point of it. Furthermore, the region around the zero point is relatively linear. We make use of this fact to generate the error signal. This error signal is used to tune the selection of the filters in the filter bank depending on the output of the derivative filter bank as shown in figure 3.9.

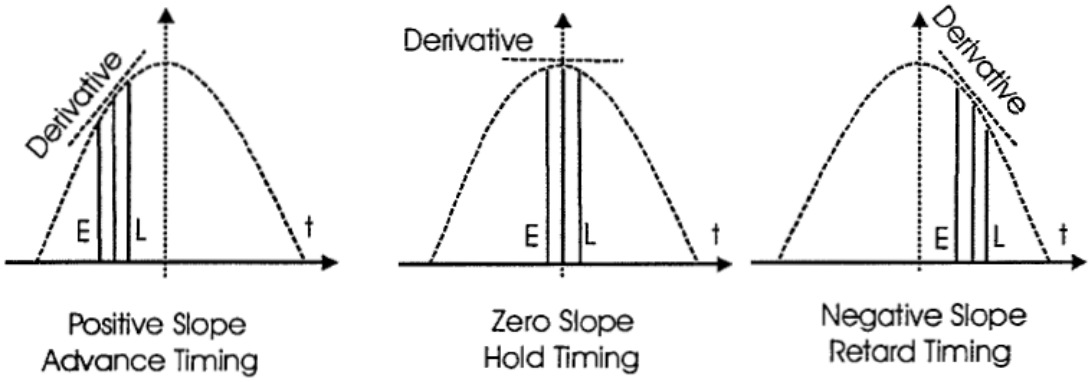


Figure 3.9: Estimating error process

But using the derivative filter bank alone contains insufficient information to determine if the timing should be time advanced, held or retarded. As shown in figure 3.10, a sample set formed prior to the peak will generate positive slope if the correlation values are positive and a negative slope if the correlation values are negative.

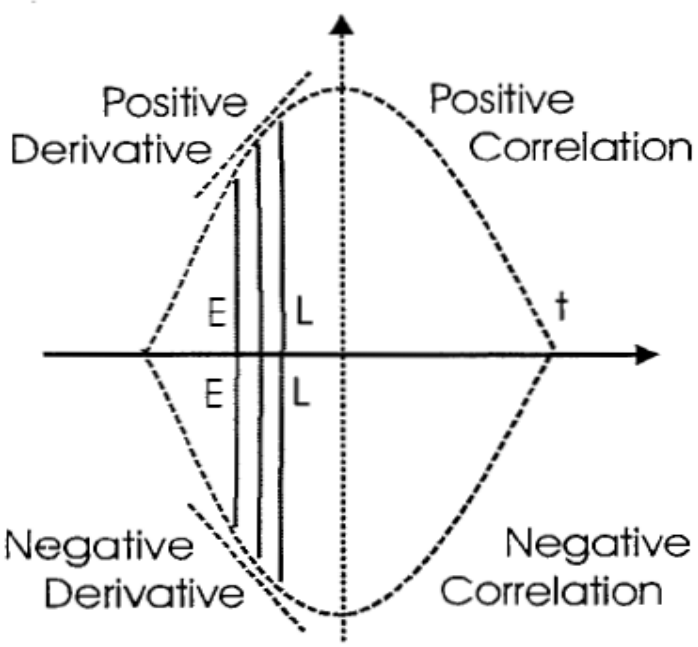


Figure 3.10: Taking the derivative output only is not enough

The solution for this problem is to get the product of the derivative matched filter output with the matched filter output or the sign of it. The whole system is described in figure 3.11.

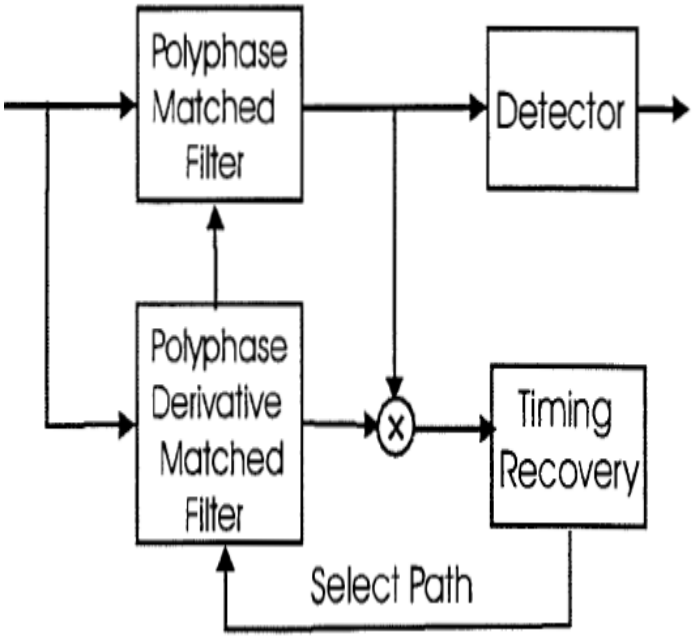


Figure 3.11: Whole system block diagram

Moving to the algorithm, the implementation of the timing recovery system is implemented in python and C++. The number of filters in the filter bank is chosen to be 32 filters with 22 taps for each one. It is represented by two blocks are root_raised_cosine (….) and pfb_clock_sync_ccf (….) , we will discuss them now shortly .

- Root raised cosine filter taps

- Root raised cosine filter taps is generated using the function root_raised_cosine (….)
- The implementation of this function is found in the file : *gr_firdes.cc*

- This function returns a vector of float numbers representing the filter taps.
    - Poly phase filter bank and timing recovery process

- The block pfb_clock_sync_ccf (….) is used to build the two filter banks as mentioned before. It is also used to perform the whole timing recovery process (applying the input to the both filter banks, estimating error and tuning to the right filter bank)
- The implementation of this block is found in the file gr_pfb_clock_sync_ccf.cc
- The input to this block is the complex signal in baseband. The timing recovery process is applied to it and then the output is the complex signal corrected.

4-Costas loop

Its main aim is phase recovery. It is responsible for the correction of the phase error. The implementation of this block is found in the file *gr_costas_loop_cc.cc* .The input to this block is the complex received signal. This block considers that the frequency offset correction and the timing recovery is performed. In BPSK the real part of the output signal is the baseband BPSK signal and the imaginary part is the error signal. This error signal (the imaginary part) is used to estimate a value for the phase error. Then this value is updated through the control loop. Two cases will arise:

- Positive error = phase correction in CW direction = signal $* e^{-j*phase\_error}$
- Negative error= phase correction in CCW direction = signal $* e^{j*phase\_error}$

The error signal should be multiplied by the real part or the sign of the real part to perform sign correction. For more illustration, look at figure 3.12.

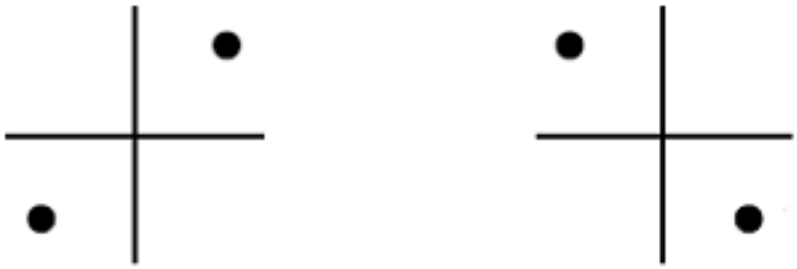


Figure 3.12.a : Phase error in CCW direction Figure 3.12.b: Phase error in CW direction

Figure 3.12 : Phase error

- As shown in figure 3.12.a, we have phase rotation error in CCW direction. So, the error signal (the imaginary part) is positive and the loop generates a phase recovery to rotate the constellation in the CW direction. But in figure 3.12.b, we have phase rotation error in CW direction .So; the error signal (the imaginary part) should be negative in order to generate a phase recovery which makes a rotation in the CCW direction. But the value of the imaginary part is also positive! . So, we multiply by the real part or the sign of the real part to perform sign correction. Finally, figure 3.13 shows a complete block diagram of costas loop.

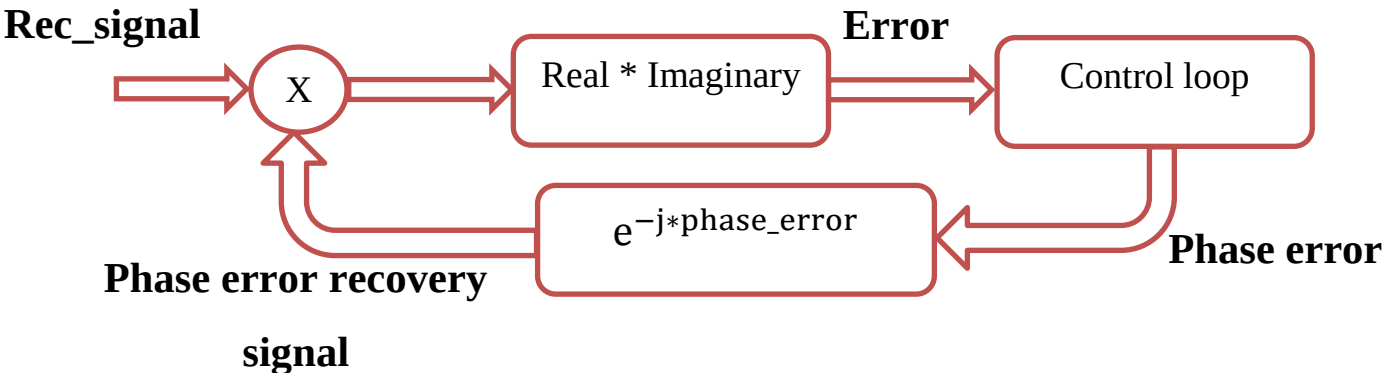


Figure 3.13: Block diagram of costas loop

5- Differential decoder

It is similar to the differential encoder as mentioned before (in transmission path). But here, it makes differential decoding based on phase change of symbols.Differential decoding is done by multiplying each sample of the input by the conjugate of the previous sample so that we can get the phase difference which is actually the data. This can be illustrated through the following code segment:

```
for(int i = 0; i < noutput_items; i++){
   out[i] = in[i] * conj(in[i-1]);
 }
```

The implementation of this block exists in the file: *diff_phasor_cc*. The input to this block is the complex received signal after performing phase, frequency and timing recovery on it. The output is the phase differences.

6- Constellation decoder

This block is responsible for finding the closest constellation point and taking the decision. The decision is taken based on the minimum distance calculated.The implementation of this block exists in the file: *gr_constellation_decoder_cb.cc*. The input to this block is the complex data (the phase differences). After making decision, the output is a byte representing the corresponding constellation point. In our case (BPSK) output will be -1 or 1.

7- Symbol mapper

It takes the constellation points coming from the constellation decoder and performs symbol mapping. The implementation of this block exists in the file: *map_bb.cc*. Its input is a group of bytes representing the constellation points. And finally we get the output as a group of bytes; each byte represents one of the transmitted bits.

3.3.3 Vector sink

Graphically or in a file depending on its data format, sinks are used to visualized the resulting signal at the end of the chain. Input parameters are depending on the kind of input data. Particularly in our scenario, since we want to see the resulting byte-type data after the demodulator block, we used a vector sink where the date recovered will be easily shown in python file , thus we can calculate BER .

Chapter 4: OFDM

4.1 Introduction

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 these independent channels is called subcarrier and transports data that is modulated in the amplitude and phase of the signal. All subcarriers 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.

4.2 Advantages and disadvantages of OFDM compared with SC

Advantages	Disadvantages
1- Efficient way to transmit at very high data rates over multipath fading channels. 2- For slow channels, loading is easier than single carrier. 3- OFDM is robust against narrow band interference, as it will affect only portion of subcarrier and may overcome it by using coding.	1- More sensitive to frequency offset. 2- OFDM has large PAPR which results in reducing the power efficiency or the RF amplifier. 3- Sensitive to doppler shift.

Table 4.1 : Advantages and disadvantages of OFDM comapred with SC

4.3 Objective

This phase from our project 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. WiMAX uses OFDM as its multiplexing method in its physical layer. LTE uses OFDM for its downlink, which is from the base station to the terminal, and a pre-coded 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)) .Again , the main goal is achieving a complete OFDM transceiver (transmitter and receiver) through USRP 2 and USRP N 210 boards , one board is for transmission while the other is for reception . Different scenarios and issues will be handled and discussed. After that, conclusions will be deduced.

4.4 Important issues in OFDM

4.4.1 Orthogonality

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 inverses of the symbol duration. The orthogonality of the frequencies used reduces greatly the crosstalk interference between sub-carriers and increases the spectrum utilization.

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

4.4.2 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, which 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 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 inter-symbol interference (ISI). Figure 4.1 shows the effect that multipath echoes can cause on a transmitted signal. To eliminate ISI completely a guard time is introduced for each OFDM signal as shown in figure 4.2. The guard time is chosen larger than the expected delay spread such that multipath component for one symbol can't interfere with the next symbol. The guard time consists of no signal. However the problem of ICI (Inter-Carrier Interference) would arise. ICI is the cross talk between different subcarriers which means they are no longer orthogonal. To eliminate ICI, OFDM symbols are cyclically extended in the guard time as shown in figure 4.3. This ensures that delayed replicas of the OFDM symbol always have an integer number of cycles within the FFT interval. And that we call CP .

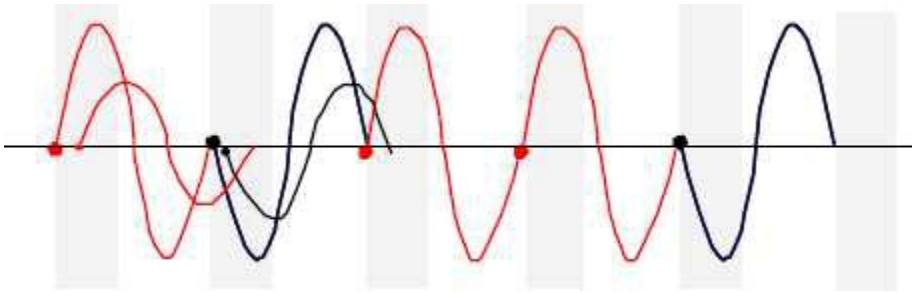


Figure 4.1:Signal in the presence of ISI

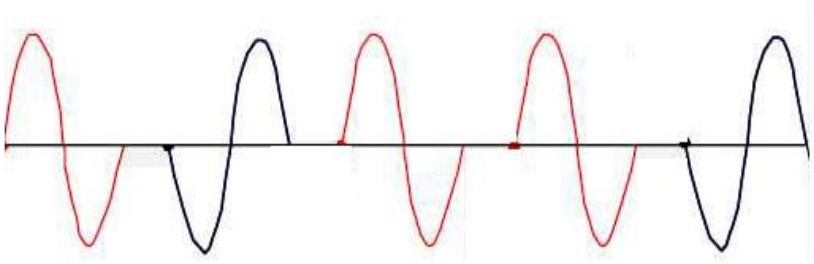


Figure 4.2:Signal in the presence of GT

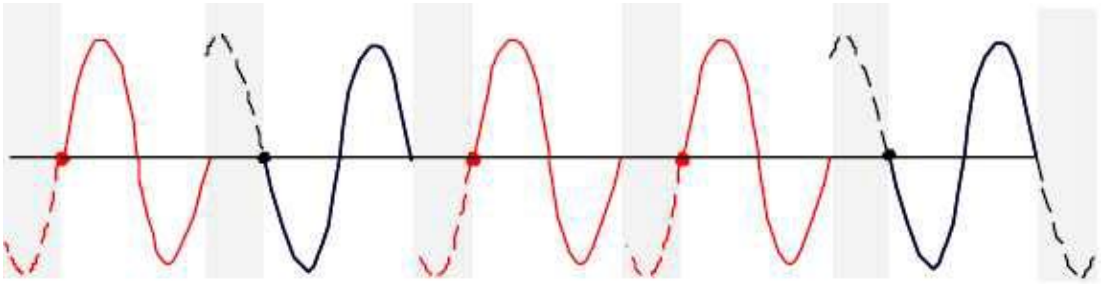


Figure 4.3:The effect of adding a cyclic prefix to the signal

4.5 OFDM modulator

4.5.1 Motivation

The OFDM modulator exits in python file ***ofdm.py.*** Also the OFDM demodulator exists in the same python file. OFDM modulator modulates an OFDM stream based on the options FFT length, occupied tones, and cyclic prefix length, this block creates OFDM symbols using a specified modulation scheme (here is BPSK). It consists of four main blocks: OFDM Mapper, OFDM Insert Preamble, FFT (Reverse) and OFDM Cyclic Prefix, each of them will be discussed in details in next subsection. As shown in figure 4.4, it interconnects some blocks that are defined in C++.

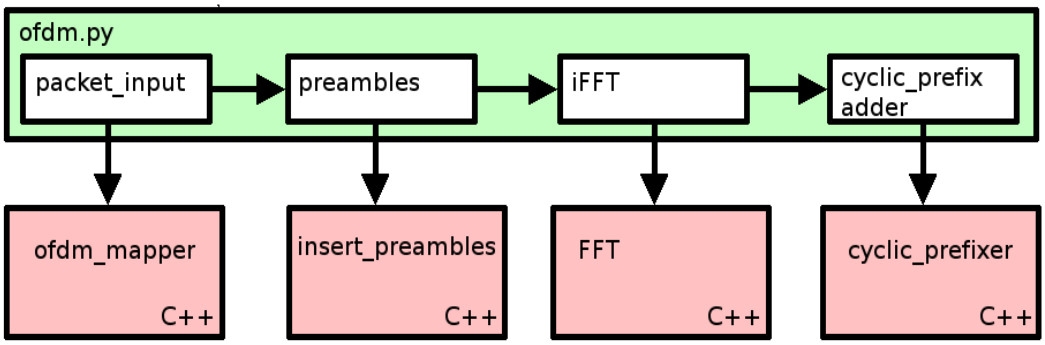


Figure 4.4: OFDM Modulator Hierarchy

Looking at the definition of the modulator in the python file we can see that it has no -direct- input and it has one output which is the complex modulated signal at baseband. The modulator includes a send function that takes as the parameter " the payload" of size 512 byte which will be sent from upper level thread " a queue of messages" , each message contains a payload from data stream .Then ,the "Send_PKT " function sends these data to the first module of the modulator (ofdm_mapper) .This payload will be converted into a data packet by calling a function (make _packet) from the python module "ofdm_packet_utils" . This function will calculate and add CRC as well as header addition, both to the payload.

4.5.2 Packet structure and construction

As we said in previous sub-section, the word “packet”. Now, we will discuss it in details, its structure as well as construction process of it .Basically , the packet structure as shown in figure 4.5 contains three fields are header, payload (512 byte) and CRC32. The header contains two equal length fields, each one of (2 byte) that comprises offset and payload with CRC length. The two equal length fields are used for identifying the packet at the receiver .Moreover, the payload with CRC fields are whitened (scrambled), which involves bit-wise XOR operation with known pseudo random (PN-sequence) stream of data.

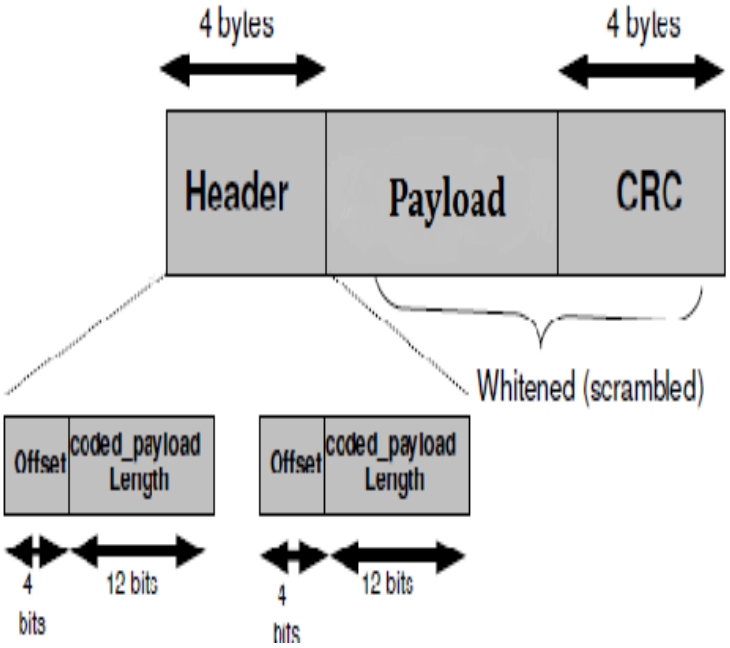


Figure 4.5: OFDM packet structure

Now, moving to construction of the packet, the following steps show how packet is constructed:

- Calculate CRC32 of a payload from crc function and append it to payload.
- Make header function that take a parameter length of payload with CRC and offset (start of window of size 512 which take a known pseudo random number from generation vector.
- Append a character (‘U’) to a payload with CRC which separate between packets.
- Make whitening (Scrambling) for payload with CRC and character (‘U’).
- Append scrambled data with the header and send it to queue of messages.

The end of (Send_PKT) function will put the data that needs to be outputted in the end of these queue of messages, and the first module in the modulator chain will access to this data automatically.

4.5.3 Blocks of OFDM modulator

1- OFDM Mapper

- It is defined in : *digital_ofdm_mapper_bcv.cc*
- Main aim: Mapping the incoming data (messages) into OFDM symbols.

- Inputs and Outputs: It has one -indirect- input which is the pushed messages. And has two outputs, the first is a stream of mapped OFDM symbols (of type gr_complex) while the other output is an array of characters (flags), each indicates the beginning of the first OFDM symbol to ease the process of adding the Pre-amble.

- Steps of operation :

1- Checking and Preparation :
   - Check and compare between occupied carriers length and FFT length.
   - Filling out carriers to occupied carriers length.
   - Padding zeros to the remainder of FFT length.
   - Store the used subcarriers in the subcarrier table with known indexes.

2- M-ary Modulation Scheme and Constellation Table :
   - Identifying the used modulation scheme with the M-ary concept.
   - Build the constellation table by generating random symbols for the pre-defined modulation scheme and insert them in a table to choose from them later.

3- The Mapping Process itself and building the symbols :

   - Checking on EOF, packet is finished or not!
     - ➔ If the packet is still in consuming process and not finished yet then ,
       - Pass on the messages message by message.
       - The flag indicates the first OFDM Symbol is raised at the beginning of consuming the packet.
       - Pass on the message byte by byte.
       - Pass on the byte bit by bit.
       - Then, building the symbols is done by mapping each subcarrier (from the known indexes) to one of the pre-defined symbols from the constellation map (pre-defined too).
       - Different cases were handled to be able to fit the number of M-ary bits.
       - Still loop till the packet goes finishing.
     - ➔ If the packet is finished
       - EOF is raised.
       - Then, go and execute the work function again as long as more packets exist.
   - Check if there is more messages or not.

Deeply into previous tree :

Actually, the most effective thing that should be kept in mind always is the process was run through consuming one (and every) packet to produce as many OFDM symbols of FFT length to hold the full packet. Thus, as long as messages are pushed, the OFDM symbols are produced, each of FFT size (512 bits), you know. The concept

of occupied carriers is used here where the used carriers are only 200 out of 512! And the remainder is padded with zeros from both sides (left and right).

1- *Checking and Preparation*

Checking that (occupied carriers <= FFT length). Otherwise, the process couldn't be completed. Then, filling out carriers to occupied carriers length by starting with the string "FE7F" with equivalent binary 1111111001111111, you know. Then, a loop is considered to pad F (1111) from both sides till reaching a maximum of 7 bits is remaindered. At this case we pad ones too from both sides with the remaindered length and it will be in equivalent for odd number of bits , e.g. : 7 bits will result in padding four bits to the left and three bits to the right . After that, padding zeros for the remainder length of the FFT size from both sides too is done. Finally, passing on each occupied bit (a bit represents ONE) and push it in the map with a known index. Of course, it was done after moving to bit domain from string domain. !

2- *M-ary Modulation Scheme and Constellation Table*

Identifying the M-ary modulation scheme -which will be used- is done here. Actually, the used modulation scheme is BPSK. Also, building the constellation table which we will map from is done here by generating random symbols depending on the pre-defined modulation scheme. For our case, generated symbols may be 1 or -1 (BPSK).

3- *The Mapping Process itself and building the symbols*

Then, the mapping process comes now, it seems late a bit but preparations are important too! Building a single symbol: First, Initialize all bins to zero to set unused carriers. After that, checking that we still have messages is done and we don't exceed number of occupied carriers so far. Then, taking a single message and working on it byte by byte (via d_msgbytes) then for each byte, passing through it too bit by bit. Then , test to make sure we can fit bits of the M-ary, then take the number of bits of M-ary at a time from the byte to add to the symbol .Also, for each taken bits , enter the constellation table and map (it) them to one of the pre-defined symbols

But, what would happen if we couldn't fit the number of bits of M-ary?! If we can't fit the number of bits of M-ary, store them for the next byte by saving residual bits of this message where residual bits must be smaller than number of bits of M-ary. Take the residual bits, fill out with info from the new byte, and put them in the symbol (also, from the constellation table).

When will we end!?! Check on End Of File (EOF) must be done each iteration, if EOF = 1 then packet was finished (free message) and go to work again to consume another packet. Otherwise, the loop is still considered to produce OFDM symbols.

Oh, we are done! GOD I cannot believe it! I have been writing this stuff for hours.

2- OFDM Insert Preamble

- It is defined in : *gr_ofdm_insert_preambles.cc*

- Main aim: Inserting pre-modulated symbols "Preamble" before each payload to ease the synchronization operation specially knowing start of the frame.

- Inputs and Outputs: It has two inputs and two outputs. The first input is a stream of vectors of gr_complex [fft_length], which are the modulated symbols of the payload. And the second input is a stream of characters, the LSB of each indicates whether the corresponding symbol in first input is the first symbol of the payload or not .It is 1 if the corresponding symbol is the first symbol, otherwise it is 0. While the first output is a stream of vectors of gr_complex [fft_length], these include the preamble symbols and the payload symbols. And the second output is a stream of characters; the LSB of each indicates whether the corresponding symbol in the first input is the first symbol of a packet (i.e., the first symbol of the preamble). It is 1 if the corresponding symbol is the first symbol, otherwise it is 0.

- Operation and flowchart: 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 stored in the file ofdm.py. The first input port will contain OFDM symbols and the second one 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. Flow chart is shown in figure 4.6.

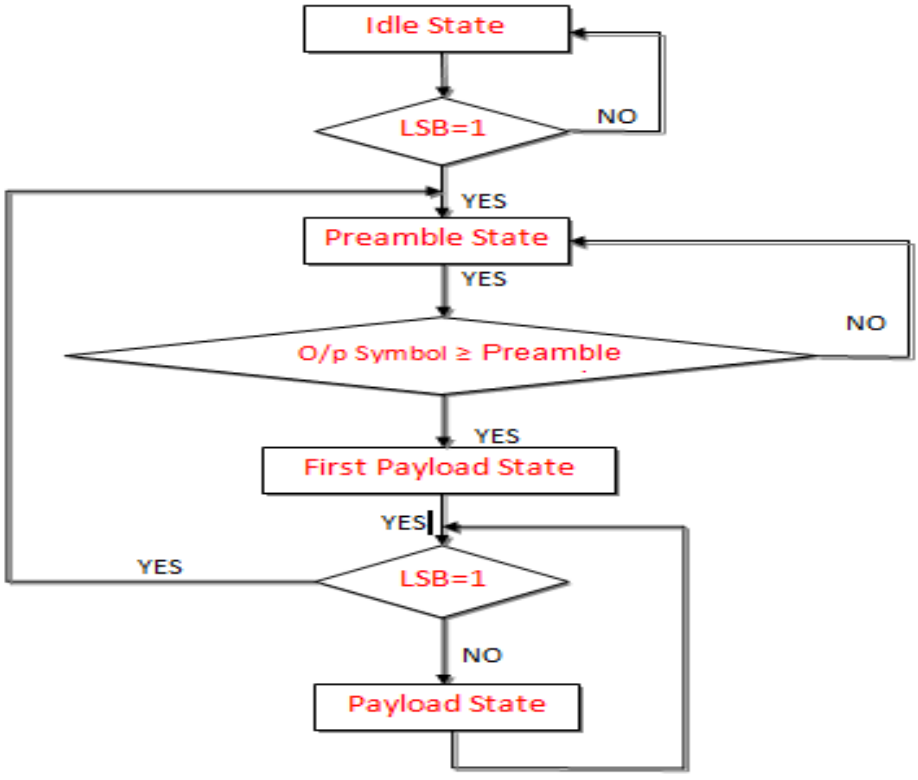


Figure 4.6: Flowchart of OFDM insert preample algorithm

- There are 4 states to insert preamble for each data packet coming from previous block:

➔ Idle state: checks the LSB of each char equal to 1 or not, if not eat on input symbol.

➔ Preamble state: outputs a specific number of OFDM symbols (known symbols) as a preamble.

➔ First payload state: runs after adding the preamble symbols which copies the first OFDM symbol of the payload from the input to the output.

➔ Payload state: copies an OFDM symbol from input to output then checks the LSB corresponding to the next symbol if 1 or not, if 1 repeat the process starting from preamble state.

3- FFT (Reverse) or iFFT

- It is defined in: gr_fft_vcc.cc
- Main aim: Computing iFFT of the produced symbols with pre-ample.
- Inputs and Outputs: It has one input and one output. It takes a vector of complex values and computes the iFFT and it represents the output.
- Operation:

➔ Orthogonality is realized with FFT/iFFT whose output is a sum of orthogonal signals. Recall, the basic functions for an IFFT are N orthogonal subcarriers, each have a different frequency and the lowest frequency is DC, and the iFFT output is the summation of all N sub-carriers. Thus, the IFFT block provides a simple way to modulate data onto N orthogonal subcarriers. The block of N output samples from the iFFT make up a single OFDM symbol.

➔ The block itself 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, if the FFT should be shifted and FFT size.

4- OFDM cyclic prefix

- It is defined in : gr_ofdm_cyclic_prefixer.cc
- Main aim: Adding cyclic prefix to OFDM symbols.
- Inputs and Outputs: It has one input and one output. It the OFDM symbols as its input, resulting in an output symbols with cyclic prefix.
- Operation: After the fourier transformation, the redundancy belonging to the cyclic prefix must be added to the OFDM signal. 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 entered symbol and the size of the cyclic prefix.

4.6 OFDM demodulator

4.6.1 Motivation

As shown in figure 4.7 , the demodulator consists of many C++ functions callable by Python. Like the modulator, it is firstly defined in the file *ofdm.py*. The inner structure of the demodulator consists of two basic modules, corresponding to the two blocks that are defined in *ofdm.py*. First one is " **ofdm_receiver "** which takes care of the synchronization and equalization of the CRC signal while the second module is " **ofdm_frame_sink "** which mainly represents a state machine that de-maps 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 .

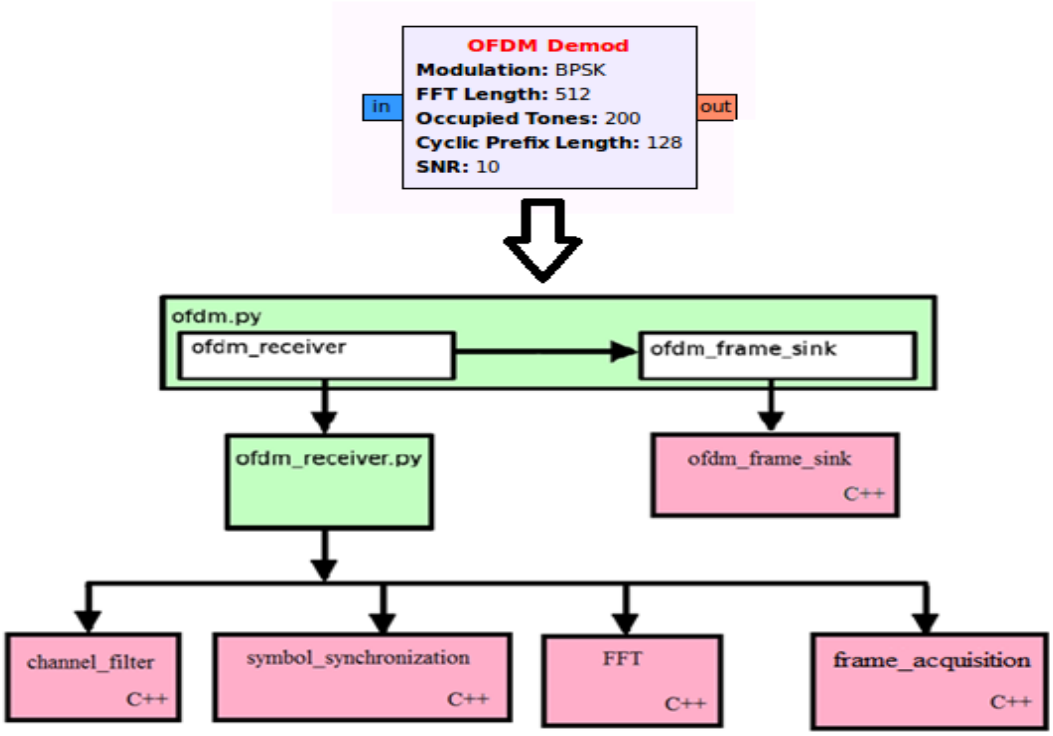


Figure 4.7: OFDM demodulator hierarchy

Each of them will be discussed later in details. Its input comes, in most cases, from the output of the channel. It demodulates a received OFDM stream based on the options FFT length, occupied tones, and CP length. Its output port will contain the demodulated signal. And the output data will be sent to a handler via a "callback function ". At the end of the demodulator chain, the output data in the output queue

(the queue of received packets) from ofdm_frame_sink module will be sent to a handler via a callback function. It 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 if data in the packet is correct or not by looking at the CRC32 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, and the output this data after remove CRC32 and the header sent to output port in ofdm demodulator block. Again and in short, this block performs: Synchronization, FFT, demodulation of incoming OFDM symbols and passing the packets to the higher layer.

4.6.2 Blocks of OFDM demodulator

A)- OFDM receiver

This block is defined in a python file ofdm_receiver.py. It includes four main modules defined in C++ files as shown in figure 4.8 :Channel Filter , OFDM Symbol Synchronization , FFT , Frame Acquisition .The detailed construction of this block is shown in figure 4.8 where the blue arrow shows the path the signal follows, while the black arrows carry other kinds of data that are not the actual OFDM signal.

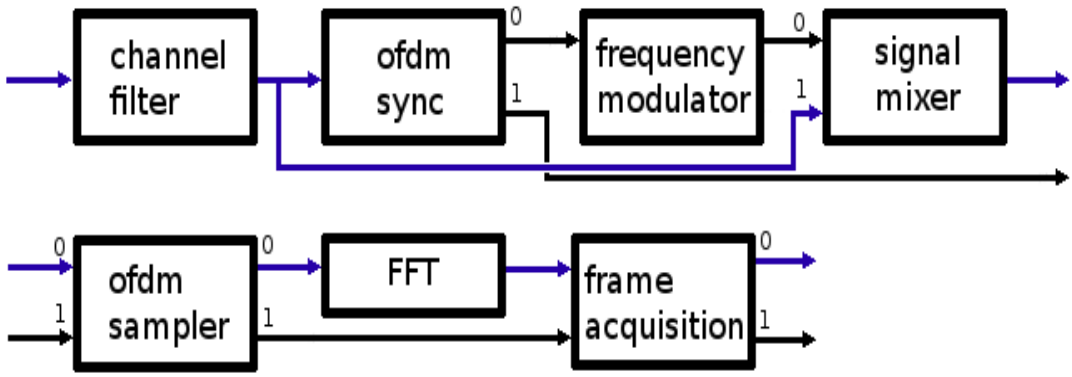


Figure 4.8: Block diagram OFDM receiver block

Its input is the complex modulated signal at baseband. And the output is synchronized packets are sent back to the demodulator. It performs receiver synchronization on OFDM symbols.It performs channel filtering as well as symbol, frequency, and phase synchronization.The synchronization routines are available in three flavors:

1- Preamble correlator (Schmidl and Cox).
2- Modified correlator with autocorrelation.
3- Cyclic prefix correlator (Van de Beeks).

Deeply into the sub-blocks :

1- Channel filter

The first module of the receiver, it is a simple fourier low pass filter for 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 512 subcarriers. BW= [(Occupied tones / FFT size)/2]*sampling frequency. It is defined in gr_fft_filter_ccc.cc. The time domain signal of it - simulated under GNU Radio - which is roughly a Sinc function, is shown in figure 4.9.

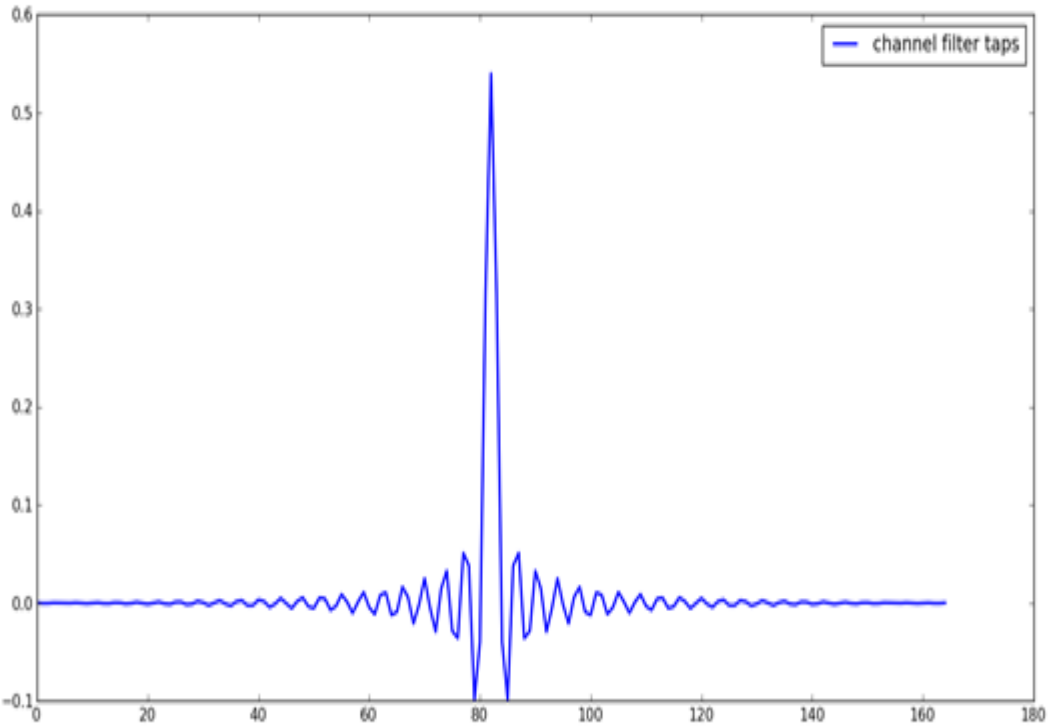


Figure 4.9 : Time domain signal of channel filter

2- OFDM symbol synchronization

It is defined in the python file *ofdm_sync_pn.py*. Once the signal has been filtered it is fed into the synchronization block. This module makes a very important task; it is responsible for making timing synchronization and frequency error correction based on Schmidl and Cox's theory [2]. Synchronization is based on PN sequence correlation. This theory is used for the correlation of received signal r[n] in many

OFDM timing synchronization methods. The preambles which are used for timing synchronization are designed for having good correlation property. In order to achieve the well correlation property, Pseudo noise (PN) sequence is used for preambles of timing synchronization. They construct the first training symbol with 2 identical halves in time by transmitting a pseudo-noise (PN) sequence only on even subcarriers, and transmitting nothing on odd subcarriers, i.e. X(2i+1) = 0 for i = 0,…….,N/2 -1 as shown in figure 4.10.

Figure 4.10:Distribution of preamble in frequency domain

Also , The form of preamble proposed in time by Schmidl and Cox is shown in figure 4.11.

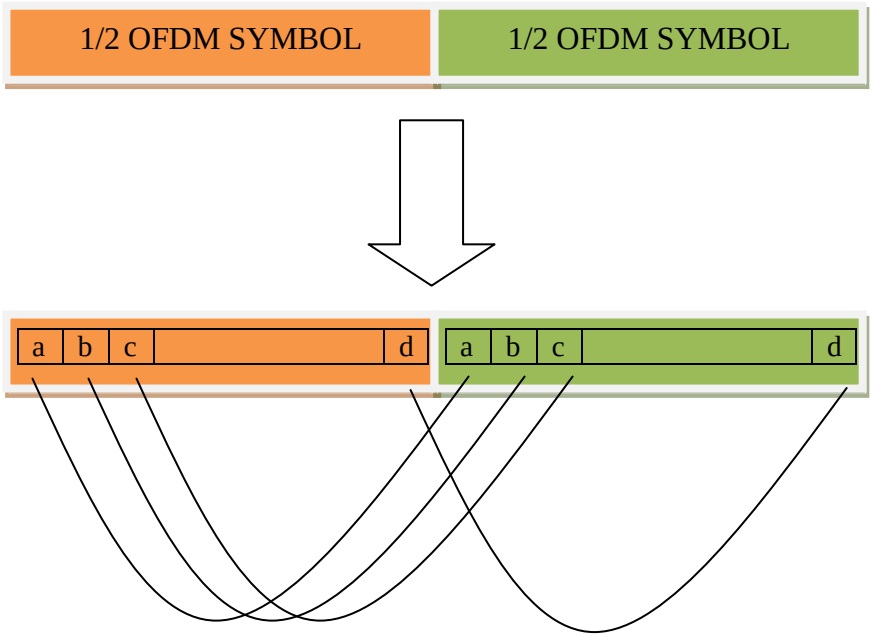


Figure 4.11:Preamble symmetry in time

The Schmidl and Cox's timing estimator finds the start of the symbol at the maximum point of the timing metric given by $M(d) = \frac{|P(d)|^2}{(R(d))^2}$.

Where, $P(d) = \sum_{k=0}^{\frac{N}{2}-1} r^*(d+k).r\left(d+k+\frac{N}{2}\right)$. And, $R(d) = \sum_{k=0}^{\frac{N}{2}-1} |r(d+k+\frac{N}{2})|^2$

Here, d is the time of first signal sample in the window of length N for symbol timing estimation, P (d) is the conjugate multiplicative result and R(d) is energy of received signal. The symbol timing offset is estimated by finding d which maximizes the timing metric M(d). The timing metric of Schmidl and Cox's method has a plateau as shown in figure 4.12 due to the CP which leads to some uncertainty as to the start of OFDM symbol.

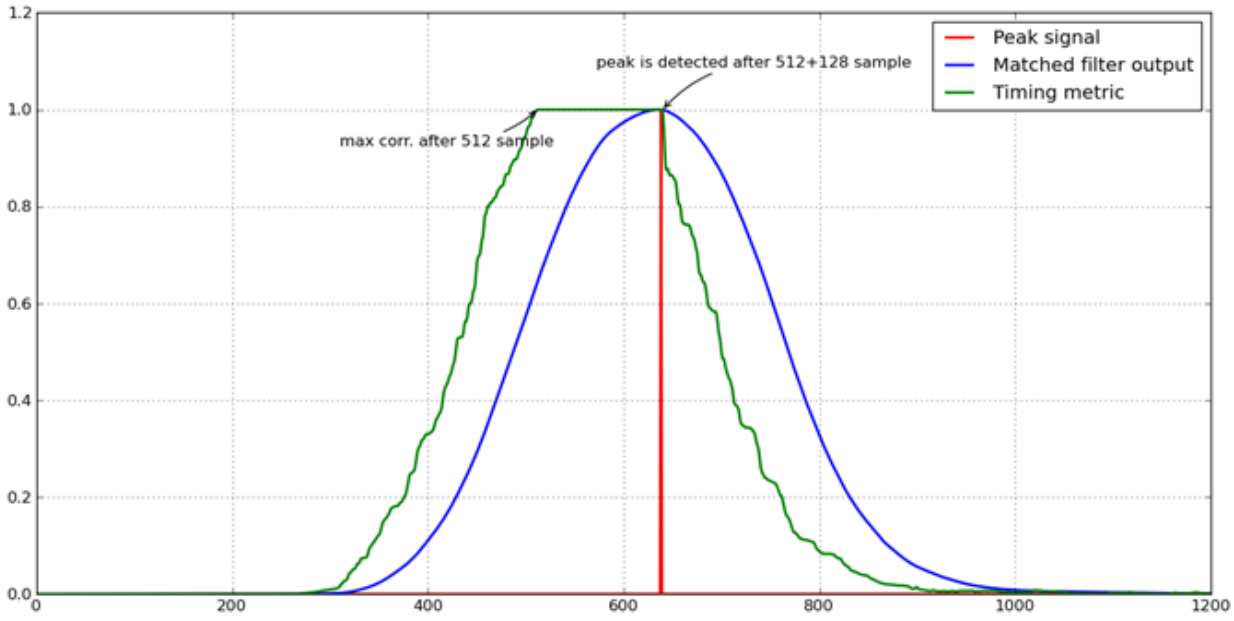


Figure 4.12: Timing metric of Schmidl and Cox method

Back to OFDM symbol synchronization , it has a single input which is the filtered signal (from channel filter) and two outputs as shown in figure 4.13 where the first is the fine frequency correction value and the other is the timing metric signal due to correlation.

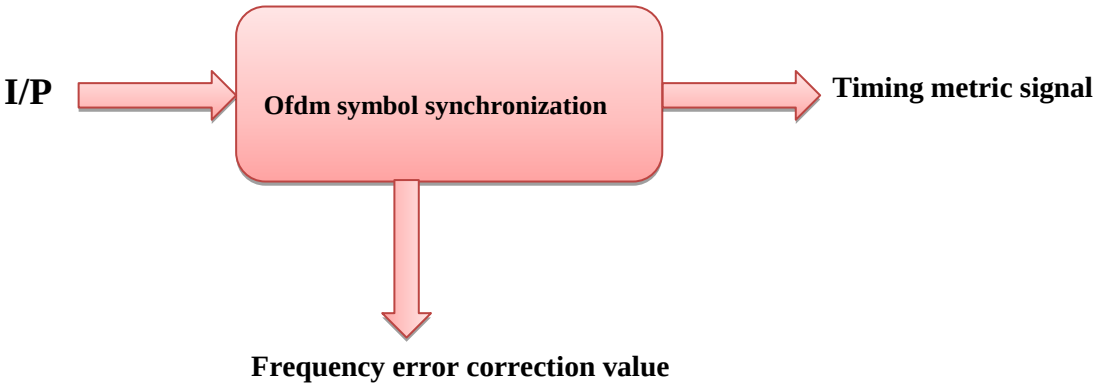


Figure 4.13: Block diagram of OFDM symbol synchronization module

Also, the block diagram of internal structure of the synchronization block is shown in figure 4.14.

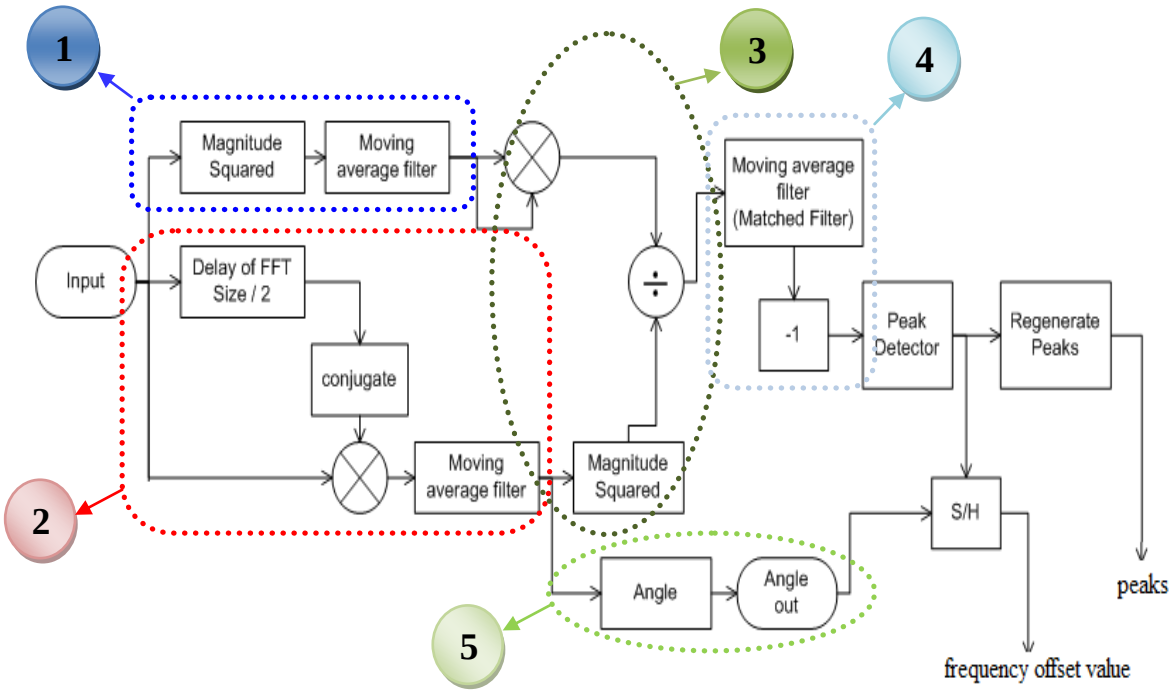


Figure 4.14:Internal structure of OFDM synchronization block

It is clear that there are five steps for synchronization process to get the start of frame and frequency error correction value,

1- The first step is calculation of signal power.
2- Then, apply correlation between the first half of delayed signal and the conjugate of second half of the received signal as shown below in figure 4.15.

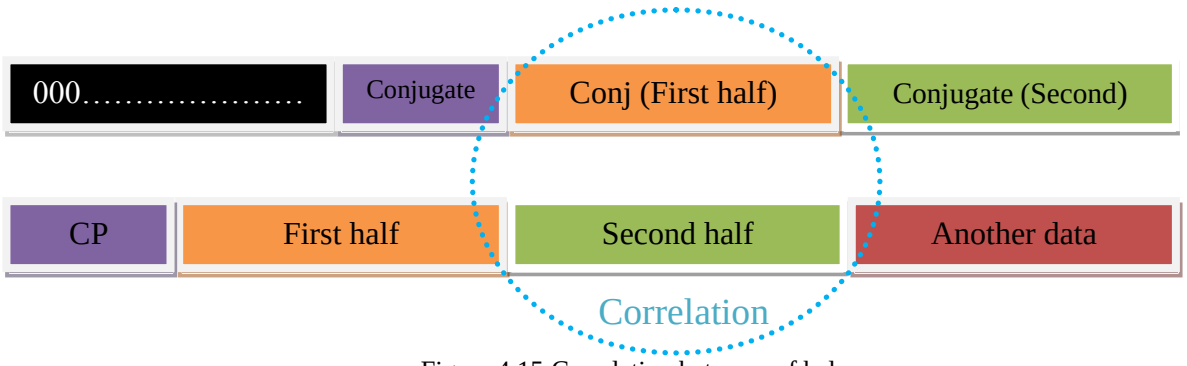


Figure 4.15:Correlation between of halves

3- Thus, we can calculate the timing metric according to the Schmidl and Cox equation as mentioned before.
4- To solve the problem of uncertainty due to the plateau outputted from timing metric, the timing metric is inputted to a matched filter (its taps are ones with CP length) to find the end of the plateau of the metric and get the smooth peak as we will show in the simulation section .
5- Calculating the angle between the first half and the second of preamble which is used for frequency error correction.

3- Frequency Modulator

It is defined in a C++ file *gr_frequency_modulator_fc.cc*. The input to this block is the frequency error correction value outputted from synchronization block which is π*Δf *T as explained before. And, $T=T_c*N$ where, T_c is the time between two samples, and N is the FFT length. So the input is $\pi\Delta f * NT_c$.

Now we want to generate $e^{-2\pi\Delta f*t}$ which we can use to eliminate the frequency offset. But $t=kT_c$ as we deal with a discrete samples, so we want to generate $e^{-2\pi\Delta f*kT_c}$. And this block is used to generate the frequency offset corresponding to each sample.

4- OFDM Sampler

It is defined in a C++ file *digital_ofdm_sampler.cc*. The ofdm_sampler block 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 matches to FFT size without existence of CP.

5- FFT (Forward)

It is also defined in C++ in the file *gr_fft_vcc.cc*. It takes a vector of complex values outputted from the sampler with FFT size length and computes the FFT.

6- Frame Acquisition

It is defined in C++ in the file *digital_ofdm_frame_acquisition.cc*. It takes a vector of complex values of the signal in the frequency domain which is the output of the FFT of a received OFDM symbol and finds the start of a frame based on two known symbols. It also looks at the surrounding bins in the FFT output for the correlation in case there is a large frequency shift in the data. It assumes that the fine frequency shift has already been corrected and that the samples fall in the middle of one FFT bin. It then uses one of those known symbols to estimate the channel response over all subcarriers and does simple 1-tap equalization on all subcarriers. This corrects the phase and amplitude distortion caused by the channel.

B)- OFDM frame sink

It is defined in: *digital_ofdm_frame_sink.cc*. It is the second module in the OFDM demodulator. It is a state machine that demaps the symbols into bits, checks the validity of the synchronized frames (from first module, ofdm_receiver) and sends them to a superior layer by adding them to the queue of received data packets. This module has, in fact, fewer complexities than its preceding one. Recall, it looks more or less like a state machine as shown in figure 4.16. This state machine has three states; the first one is "sync search", in which the algorithm looks for the flag in the second input indicating the beginning of the frame (looks for the flag indicating beginning of packet).

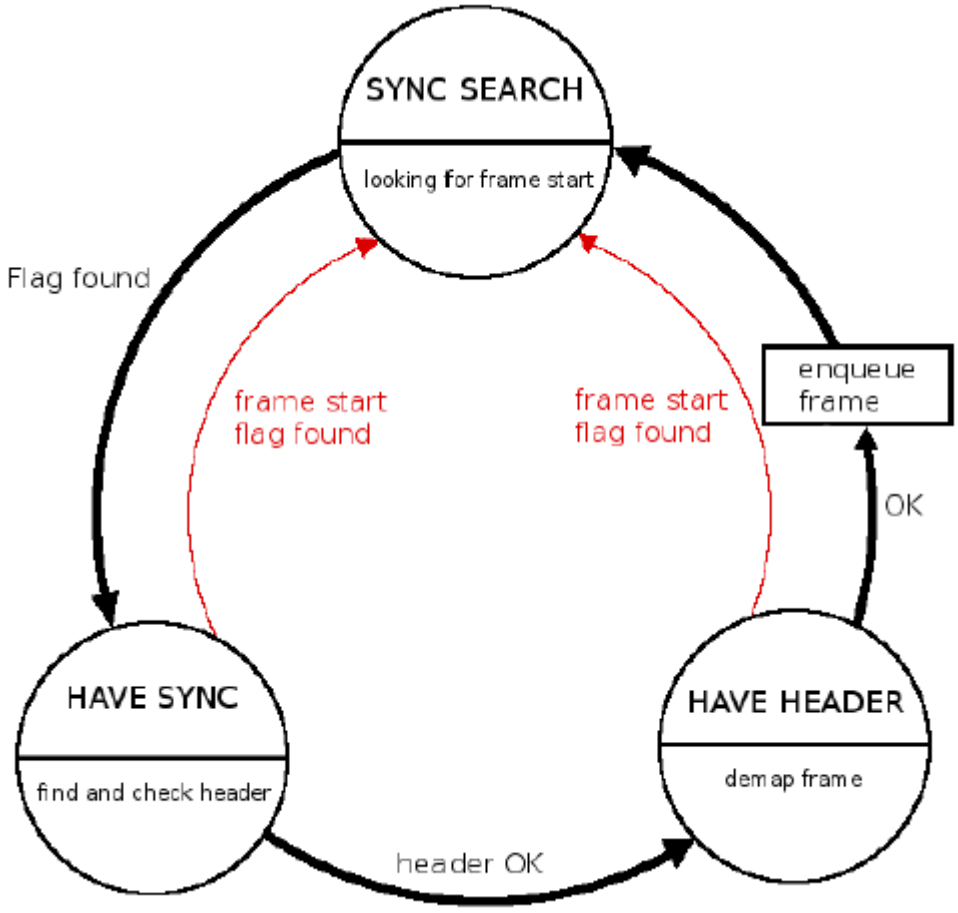


Figure 4.16: State machine representing OFDM frame sink block

Once it is found it will arrive in the "have sync" state, the second state, you know. 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 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.

Moving to the algorithm a bit:
Now, we will discuss some important ((but, not all) functions used such as:

- In enter have sync: State is assigned to have sync. At the beginning of state "have sync ", resetting PLL is done.
- In enter have header: State is assigned to have header. As explained before (in packet format) header consists of two 16-bit shorts in network byte order, payload length is lower 12 bits and whitener offset is upper 4 bits. Here, in "have header" state, the two fields are got.
- Slicer: Takes a stream of complex received data (complex symbol by symbol), calculates the minimum Euclidean distance and returns this index of nearest symbol.
- Demapper: Takes the coming data and produces the output demapped data. The demodulation of OFDM symbols are performed using PLL. The demapper block not only produces demapped data but it returns the number of bytes produced as well. It makes that by passing through each symbol (bit in our case), demaps it and stuff it into a byte (each byte will contain eight symbols). After that, it pushes this byte (a byte by byte) into
  the output vector.
- As explained before (in Mapper) ,the same step in Mapper is done here too
    - Checking and Preparation :

Checking that (occupied carriers <= FFT length). Otherwise, the process couldn't be completed. Then, filling out carriers to occupied carriers length by starting with the string "FE7F" with equivalent binary 1111111001111111, you know. Then, a loop is considered to pad F (1111) from both sides till reaching a maximum of 7 bits is remaindered. At this case we pad ones too from both sides with the remaindered length and it will be in equivalent for odd number of bits , e.g. : 7 bits will result in padding four bits to the left and three bits to the right . After that, padding zeros for the remainder length of the FFT size from both sides too is done. Finally, passing on each occupied bit (a bit represents ONE) and push it in the map with a known index. Of course, it was done after moving to bit domain from string domain. !

- M-ary modulation

Here, identifying number of M-ary bits used is done. Besides that, it checks whether number of produces symbols are true (two in BPSK) and their values are true as well .

But how all work together, from the name, all work through the "WORK" function. Basically, it is the most important function in that block (and every one too). As its name reveals, it makes all things (all work together). It takes the coming symbols from previous module and produces the output in accordance with previous functions (slicer, demapper , PLL ,..) and discussions . Inside thw "Work" function, the following steps are done :

1-Sync search: Looks for flag indicating beginning of the packet. If successfully found, set up for header decode and go to " enter_have_sync " which discussed before .
2-Have sync: Only demodulate after getting the preamble signal. It calls the "demapper" function (and inherently calls the "slicer" function) which discussed

before both returning demapped output symbols and number of bytes produced. Firstly, it gets the header, if it is OK, it completes the packaging of received packet and finally pass it to the output queue .In contrary, if header isn't correct, it represents a packet loss. So, we go out of the loop and see another packet . The process is being repeated till all received data are consumed.

3- If one packet is still not filled with bytes returned. We go to "have Header "state which calls demapper function again to produce bytes again .

A Side note, as said before, the demodulation of OFDM symbols are performed using PLL. To understand more let us assume:

- p_k : is the phase locked loop coefficient; $p_k = |p_k|e^{j\emptyset}$
- ϵ is the accumulated error
- f is the frequency shift
- $\emptyset_g$ is the phase gain
- f_g is the frequency gain
- e_g is the equalizer gain
- $\hat{X}_k$ be the closest constellation point to $\acute{Y}_k$. Also, $\hat{X}_k = C_i$, when euclidean distance metric is minimum i.e., $\min_{1 \leqslant i \leqslant M} \|\acute{Y}_k - C_i\|$, where C_i are from the constellation points of size M.

Where, $\acute{Y}_k$ is the received symbols (after previous module operation).

→ Algorithm :

Step 1 : Initialize, $\phi \leftarrow 0;\ \epsilon \leftarrow 0; f \leftarrow 0; \phi_g \leftarrow 0.25; f_g \leftarrow \phi_g^2/4;\ e_g \leftarrow 0.05$

Step 2 : Initialize, $p = [p_1, p_2, \cdots, p_{\tilde{N}}];\ where,\ p_1 = p_2 = \cdots = p_{\tilde{N}} \leftarrow 1.0 + 0.0j;$

Step 3 : Pass on all received subcarriers and perform the following *:*

$\acute{\hat{Y}}_k \leftarrow \hat{Y}_k \times e^{j\phi} \times p_k$ // Update with previous calculated PLL coefficients and phase error

$\epsilon \leftarrow \epsilon + \acute{\hat{Y}}_k \times \hat{X}_k^{*}; \epsilon \leftarrow |\epsilon| e^{j\theta}$ // Update accumulated error and making phase correction (gradually) by multiplying with conjugate of nearest symbol .

Also , check if $(\|\acute{Y}_k\| > 0.001)$ →if yes do : $p_k \leftarrow p_k + e_g \times (\hat{X}_k / \acute{Y}_k - p_k)$ otherwise , it is considered as not useful and no correction is needed .

Note that : $p_k \leftarrow p_k + e_g \times \left(\hat{X}_k / \acute{\hat{Y}}_k - p_k\right)$ represents updating of PLL coefficients .

$Step\ 4\ :\ \theta \leftarrow \arg(\epsilon)$ / / take phase of accumulated error to update with it

$Step\ 5\ :\ f \leftarrow f - f_g \times \theta$ // update frequency error

$Step\ 6\ :\ \phi \leftarrow \phi + f - \phi_g \times \theta$ // update phase error due to frequency and phase error

Steps 4 ,5 and 6 are done after finishing one complete OFDM symbol (by passing on all its subcarriers) , this means that updating PLL coefficients is done at each iteration (subcarrier). But, the values are assigned, in fact, at each OFDM symbol and hence the corrections is done on OFDM symbols domain (as an average errors and corrections) and not on subcarriers domain.

4.7 A point by point simulation

Actually, this sub-section is not only very educational for beginners but also it completes the whole picture for a professional communication engineer. Simulations are done using python programming language. And the figures are plotted using pylab module. Also, these simulations are done without any USRP boards used, only simulation through GRC to clarify and verify the function of each block. The practical system using USRPs will come later, stay tuned.

4.7.1 Mapper

- Aim: Showing an example for header and scrambled data of OFDM symbols. Simulation is done based on input = [0,0,0,0,0, …………… ,0] with length 512, occupied tones = 200 and FFT length = 512. The input is chosen to be all zeros because the scrambled data are added to the input data (xor for each bit). So, the output will be the scramble data because it is already added to zeros.

- The following data is representing the first OFDM symbol :

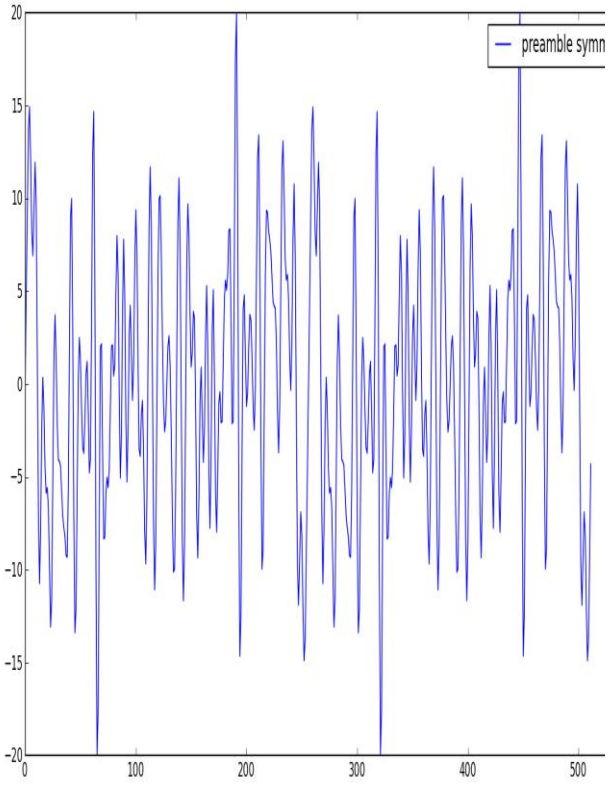
(0.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,

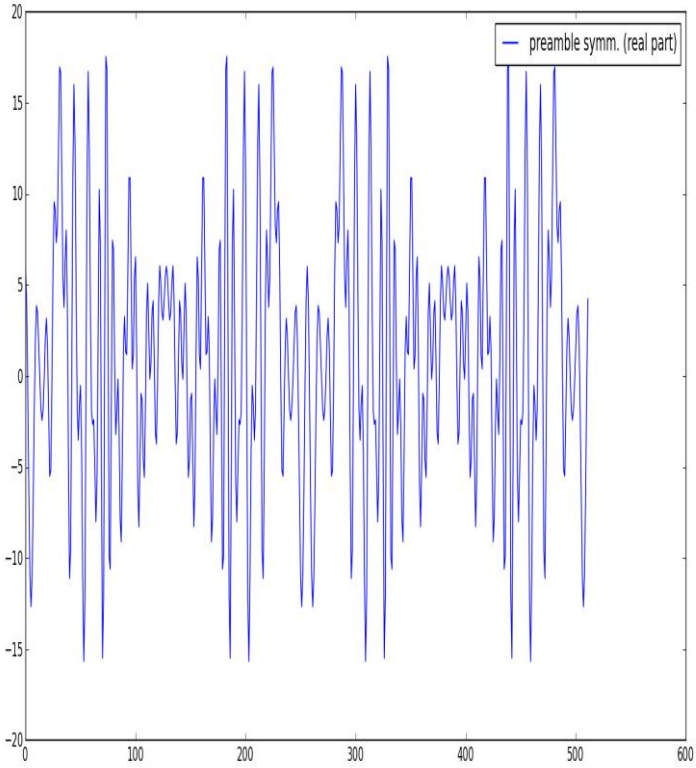
1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, 0.0)

- Header contains a constant data represented in two parts, one is payload length_with_crc and the other is whiten_offset which is '\x02\x04\x02\x04' ant it is generated from make header function. From the above output we can see that the header is (1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0)

4.7.2 Preample

As said before, after adding zeros in odd frequencies of the preamble and its time domain representation resulting in symmetric signal. This Simulation is done for the PN sequence existing in the file *ofdm.py* which represents the preamble after adding zeros in odd frequencies. Figure 4.17 shows this symmetry in time domain in real part as well as imaginary part.





:Real part Imaginary part

Figure 4.17: Preample symmetry

4.7.2 BW and channel filter

- The following figure 4.18 shows the frequency domain representation of t transmitted data with fft_length = 512, occupied_tones = 200,cyclic_prefix = 128 and Sampling rate = 32 k sample/second

- B.W calculations :-

→ Symbol time per subcarrier = 512/32000 second = 0.016 second. What is the reciprocal of previous quantity? (Left for the reader) .

→ Bandwidth of the signal = occupied carriers / Symbol time per subcarrier

→ Then, B.W = 200 / 0.016 =12.5 KHz (as shown in the figure .

→ Or, you may simply say that the BW = Sample rate $*$ $\frac{\text{Occupied tones}}{\text{FFT}-\text{Length}}$ $= 32000 * \frac{200}{512} =$ 12.5 KHz as in previous method.

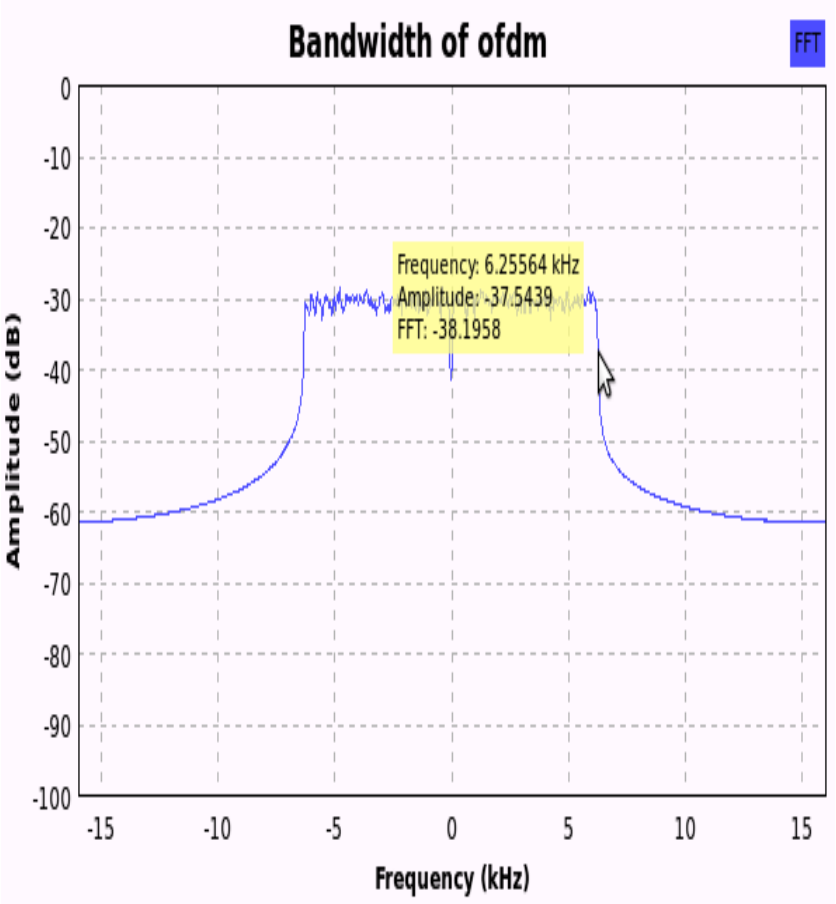


Figure 4.18: Transmitted OFDM signal

Now, can you get number of symbols per packet and number of samples per packet? (Left for the reader) .

Channel filter is designed to cover the whole B.W with an extra value = 0.08*B.W. Figure 4.19 shows the filter taps in the time domain based on: fft_length = 512, occupied_tones = 200

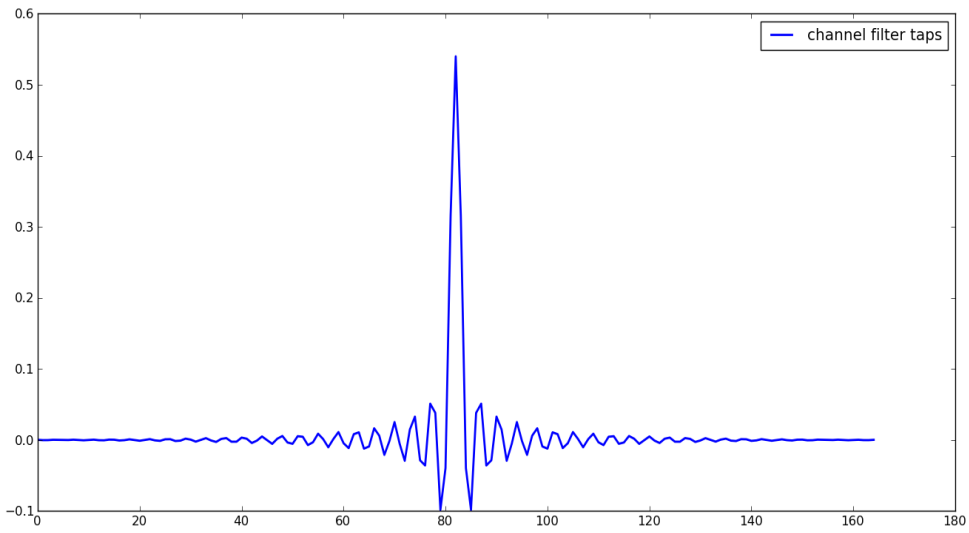


Figure 4.19: Time domain representation for filter taps

4.7.3 PN based synchronization

This part is discussed through figures . The following three figures 4.20 ,4.21 and 4.22 show the output of the timing metric, the matched filter and the peaks respectively .

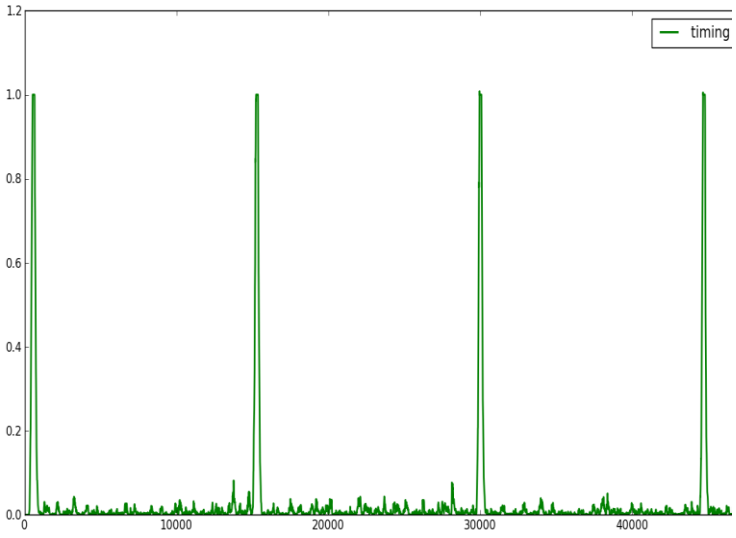


Figure 4.20: Output of the timing metric

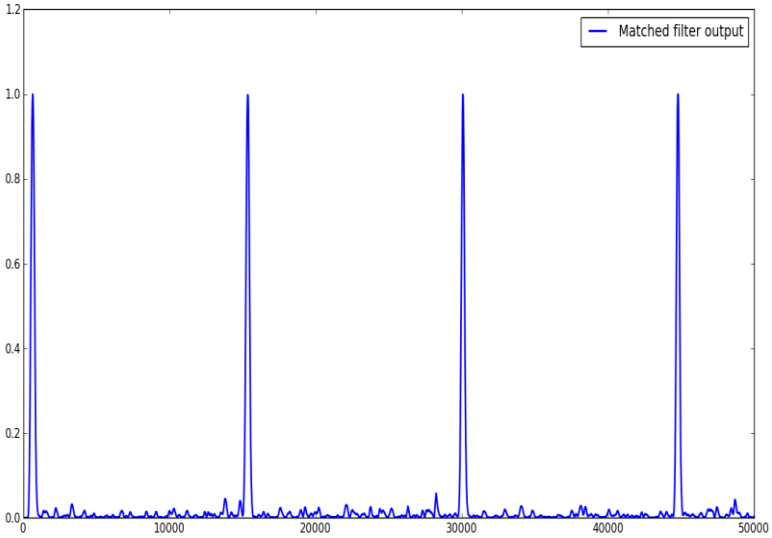


Figure 4.21: Output of the matched filter

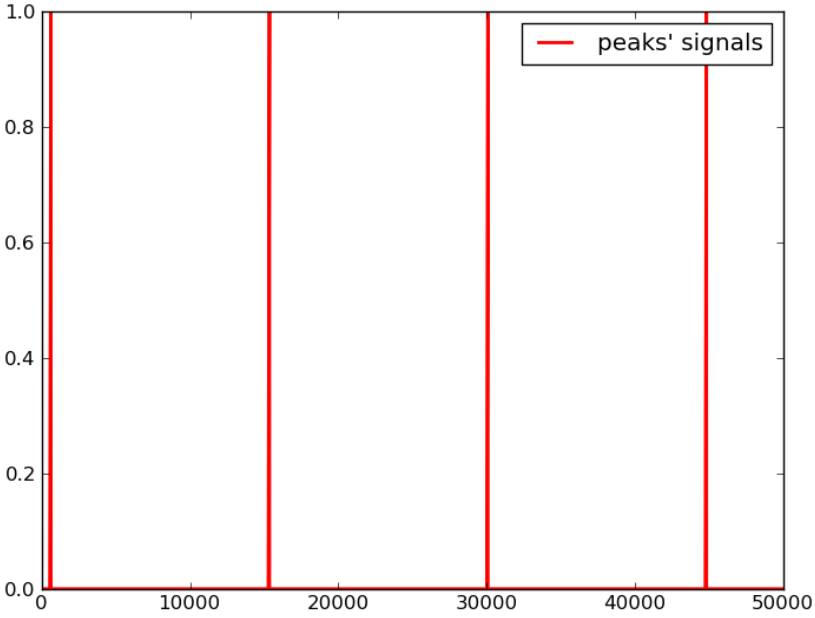


Figure 4.22: The peaks

4.7.4 Peak detection

Figure 4.23 shows simulation for the timing metric, matched filter and the peak signal for only 1200 points. This figure shows also how the peak is detected based on fft_length = 512, cyclic_prefix = 128.

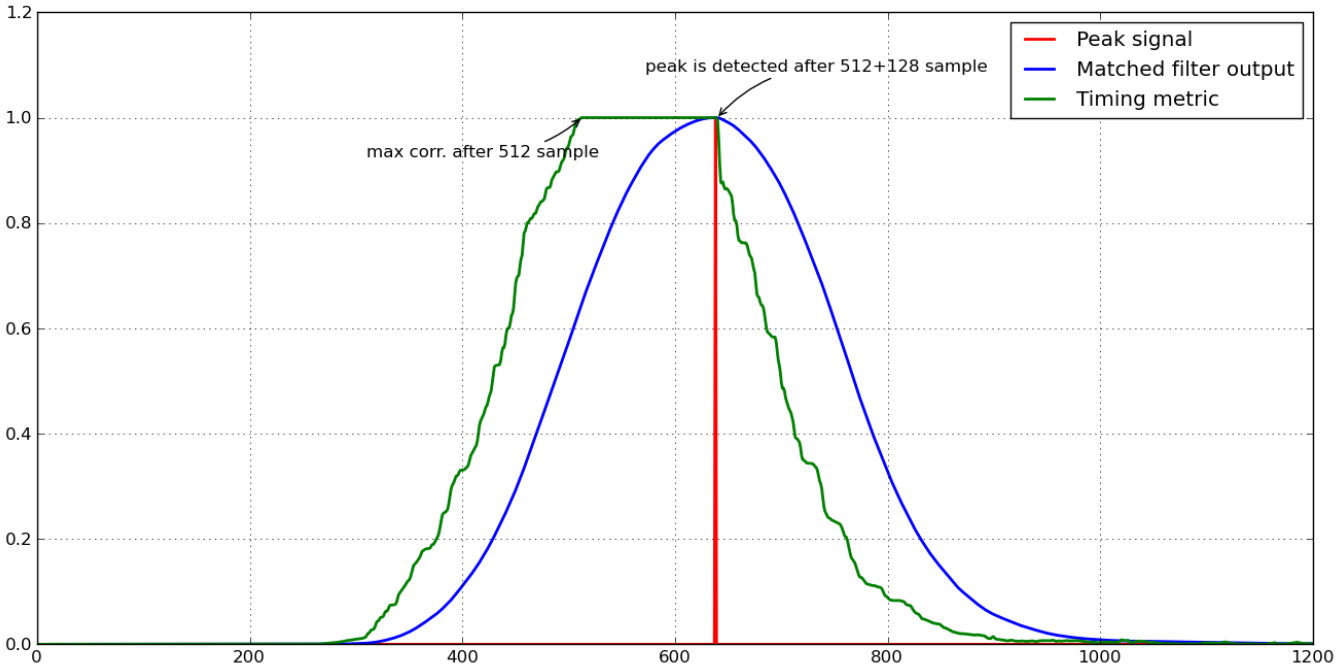


Figure 4.23: Peak detection process

4.8 Simulation codes

Previous discussions, results and figures were obtained from writing some codes through Python programming language. Codes are listed below with same sequence of previous subsection.

4.8.1 Preample

```
from gnuradio import digital
import pylab
import math
from gnuradio import blks2
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import window
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx
class top_block(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Top Block")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000
            _fft_length=512
        _occupied_tones=200



        zeros_on_left = int(math.ceil((_fft_length - _occupied_tones)/2.0))
        ksfreq = digital.known_symbols_4512_3[0:_occupied_tones]
```

```
for i in range(len(ksfreq)):
        if((zeros_on_left + i) & 1):
                ksfreq[i] = 0

# hard-coded known symbols
preambles = ksfreq
##padded_preambles = list()
##for pre in preambles:
##        padded = _fft_length*[0,]
##        padded[zeros_on_left : zeros_on_left + _occupied_tones] = pre
##        padded_preambles.append(padded)
    padded_preambles = _fft_length*[0,]
    padded_preambles[zeros_on_left : zeros_on_left + _occupied_tones] = preambles

##################################################
# Blocks
##################################################
self.gr_vector_to_stream_0 = gr.vector_to_stream(gr.sizeof_gr_complex*1, 512)
self.gr_vector_source_x_0 = gr.vector_source_c(padded_preambles, False, 1)
self.gr_vector_sink_x_0 = gr.vector_sink_f(1)
self.gr_vector_sink_x_1 = gr.vector_sink_f(1)
    self.gr_fft_vxx_0 = gr.fft_vcc(512, False, [], True)
self.gr_stream_to_vector_0 = gr.stream_to_vector(gr.sizeof_gr_complex*1, 512)
self.gr_complex_to_real_0 = gr.complex_to_real(1)
self.gr_complex_to_imag_0 = gr.complex_to_imag(1)

##################################################
# Connections
##################################################
self.connect((self.gr_vector_source_x_0, 0), (self.gr_stream_to_vector_0, 0))
self.connect((self.gr_stream_to_vector_0, 0), (self.gr_fft_vxx_0, 0))
self.connect((self.gr_fft_vxx_0, 0), (self.gr_vector_to_stream_0, 0))
    self.connect((self.gr_vector_to_stream_0, 0), (self.gr_complex_to_real_0, 0))
self.connect((self.gr_vector_to_stream_0, 0), (self.gr_complex_to_imag_0, 0))
```

```
            self.connect((self.gr_complex_to_real_0, 0), (self.gr_vector_sink_x_0, 0))
        self.connect((self.gr_complex_to_imag_0, 0), (self.gr_vector_sink_x_1, 0))
    def get_samp_rate(self):
        return self.samp_rate
    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
#           self.blks2_stream_to_vector_decimator_0.set_sample_rate(self.samp_rate)
if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = top_block()
    tb.Run(True)
     fig = pylab.figure(1)
     sp = fig.add_subplot(1,1,1)
     p2 = sp.plot(tb.gr_vector_sink_x_0.data(), "b", linewidth=1.2, label="preamble symm. (real
part)",)

     fig2 = pylab.figure(2)
     sp2 = fig2.add_subplot(1,1,1)
    p2 = sp2.plot(tb.gr_vector_sink_x_1.data(), "b", linewidth=1.2, label="preamble symm.
(imag. part)",)
     sp.legend()
     pylab.show()
```

4.8.2 BW and channel filter

```
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import window
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from gnuradio.wxgui import fftsink2
from grc_gnuradio import blks2 as grc_blks2
```

```
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import numpy
import wx

class top_block(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Top Block")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000

        ##################################################
        # Blocks
        ##################################################
        self.wxgui_fftsink2_0 = fftsink2.fft_sink_c(
            self.GetWin(),
            baseband_freq=0,
            y_per_div=10,
            y_divs=10,
            ref_level=0,
            ref_scale=2.0,
            sample_rate=samp_rate,
            fft_size=1024,
            fft_rate=15,
            average=False,
            avg_alpha=None,
            title="FFT Plot",
            peak_hold=False,
        )
        self.Add(self.wxgui_fftsink2_0.win)
```

```
        self.random_source_x_0 = gr.vector_source_b(map(int, numpy.random.randint(0,
256, 1000)), True)
        self.gr_throttle_0 = gr.throttle(gr.sizeof_gr_complex*1, samp_rate)
        self.digital_ofdm_mod_0 = grc_blks2.packet_mod_b(digital.ofdm_mod(
                options=grc_blks2.options(
                        modulation="bpsk",
                        fft_length=512,
                        occupied_tones=200,
                        cp_length=128,
                        pad_for_usrp=False,
                        log=None,
                        verbose=None,
                ),
        ),
        payload_length=0,
        )

        ##################################################
        # Connections
        ##################################################
        self.connect((self.random_source_x_0, 0), (self.digital_ofdm_mod_0, 0))
        self.connect((self.digital_ofdm_mod_0, 0), (self.gr_throttle_0, 0))
        self.connect((self.gr_throttle_0, 0), (self.wxgui_fftsink2_0, 0))

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.wxgui_fftsink2_0.set_sample_rate(self.samp_rate)

if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
```

```
tb = top_block()
tb.Run(True)
```

Channel filter

```
import pylab
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class top_block(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Top Block")

        ##################################################
        # Variables
        ##################################################
        fft_length = 512
        occupied_tones = 200
        bw = (float(occupied_tones) / float(fft_length)) / 2.0
        tb = bw*0.08

        ##################################################
        # Blocks
        ##################################################
        self.chan_coeffs = gr.firdes.low_pass (1.0,                # gain
                1.0,                # sampling rate
                bw+tb,              # midpoint of trans. band
                tb,                 # width of trans. band
                gr.firdes.WIN_HAMMING)   # filter type
```

```
    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate

if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = top_block()
    tb.Run(True)
    fig = pylab.figure(1)
      sp = fig.add_subplot(1,1,1)
      p2 = sp.plot(tb.chan_coeffs, "b", linewidth=2, label="channel filter taps")
      sp.legend()
      pylab.show()
```

4.8.3 PN based synchronization

Code is divided into two parts. First one is responsible for writing the needed data to certain files. The second one is responsible for reading the data and plotting it.

→ First code (writing data to files)

```
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from grc_gnuradio import blks2 as grc_blks2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import numpy
import wx
```

```
class top_block(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Top Block")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000

        ##################################################
        # Blocks
        ##################################################
        self.random_source_x_0 = gr.vector_source_b(map(int, numpy.random.randint(0,
256, 1000)), True)
        self.gr_vector_sink_x_2 = gr.vector_sink_b(1)
        self.gr_vector_sink_x_1 = gr.vector_sink_b(1)
        self.gr_vector_sink_x_0 = gr.vector_sink_f(1)
        self.gr_throttle_0 = gr.throttle(gr.sizeof_float*1, samp_rate)
        self.digital_ofdm_sync_pn_0 = digital.ofdm_sync_pn(512, 128, True)
        self.digital_ofdm_mod_0 = grc_blks2.packet_mod_b(digital.ofdm_mod(
                options=grc_blks2.options(
                        modulation="bpsk",
                        fft_length=512,
                        occupied_tones=200,
                        cp_length=128,
                        pad_for_usrp=False,
                        log=None,
                        verbose=None,
                ),
            ),
            payload_length=0,
        )
```

```
        ##################################################
        # Connections
        ##################################################
        self.connect((self.digital_ofdm_mod_0, 0), (self.digital_ofdm_sync_pn_0, 0))
        self.connect((self.gr_throttle_0, 0), (self.gr_vector_sink_x_0, 0))
        self.connect((self.digital_ofdm_sync_pn_0, 0), (self.gr_throttle_0, 0))
        self.connect((self.digital_ofdm_sync_pn_0, 1), (self.gr_vector_sink_x_1, 0))
        self.connect((self.random_source_x_0, 0), (self.gr_vector_sink_x_2, 0))
        self.connect((self.random_source_x_0, 0), (self.digital_ofdm_mod_0, 0))

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate

if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = top_block()
    tb.Run(True)
```

Second code (Read data from files and plotting it)

```
import pylab
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class top_block(grc_wxgui.top_block_gui):
```

```
        def __init__(self):
                grc_wxgui.top_block_gui.__init__(self, title="Top Block")

                ##################################################
                # Variables
                ##################################################
                self.samp_rate = samp_rate = 32000

                ##################################################
                # Blocks
                ##################################################
                self.gr_vector_sink_x_0 = gr.vector_sink_b(1)
                self.gr_throttle_0 = gr.throttle(gr.sizeof_char*1, samp_rate)
                self.gr_file_source_0 = gr.file_source(gr.sizeof_char*1, "current directory/ False)

                ##################################################
                # Connections
                ##################################################
                self.connect((self.gr_file_source_0, 0), (self.gr_throttle_0, 0))
                self.connect((self.gr_throttle_0, 0), (self.gr_vector_sink_x_0, 0))

        def get_samp_rate(self):
                return self.samp_rate

        def set_samp_rate(self, samp_rate):
                self.samp_rate = samp_rate

if __name__ == '__main__':
        parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
        (options, args) = parser.parse_args()
        tb = top_block()
        tb.Run(True)
         fig = pylab.figure(1)
         sp = fig.add_subplot(1,1,1)
```

```
        p2 = sp.plot(tb.gr_vector_sink_x_0.data()[:50000], "r", linewidth=2, label="peaks'
signals")
        sp.legend()
        pylab.show()
```

The previous code is for the peaks file only. The codes of the matched filter and the timing metric are the same except the location of the file and the size of the throttle and the file source output. Then replace :-

```
self.gr_throttle_0 = gr.throttle(gr.sizeof_char*1, samp_rate)
                self.gr_file_source_0 = gr.file_source(gr.sizeof_char*1, "/current
directory/ofdm_sync_pn-peaks_b.dat", False)
```

By:-

```
self.gr_throttle_0 = gr.throttle(gr.sizeof_float*1, samp_rate)
                self.gr_file_source_0 = gr.file_source(gr.sizeof_float*1, "/current
directory/sync_pn_analysis/your_file.dat", False)
```

Peak detection:-

First code (generate files of data):-

```
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from grc_gnuradio import blks2 as grc_blks2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import numpy
import wx

class top_block(grc_wxgui.top_block_gui):
```

```
    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Top Block")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000

        ##################################################
        # Blocks
        ##################################################
        self.random_source_x_0 = gr.vector_source_b(map(int, numpy.random.randint(0,
256, 1000)), True)
        self.gr_vector_sink_x_2 = gr.vector_sink_b(1)
        self.gr_vector_sink_x_1 = gr.vector_sink_b(1)
        self.gr_vector_sink_x_0 = gr.vector_sink_f(1)
        self.gr_throttle_0 = gr.throttle(gr.sizeof_float*1, samp_rate)
        self.digital_ofdm_sync_pn_0 = digital.ofdm_sync_pn(512, 128, True)
        self.digital_ofdm_mod_0 = grc_blks2.packet_mod_b(digital.ofdm_mod(
                    options=grc_blks2.options(
                            modulation="bpsk",
                            fft_length=512,
                            occupied_tones=200,
                            cp_length=128,
                            pad_for_usrp=False,
                            log=None,
                            verbose=None,
                    ),
            ),
            payload_length=0,
        )

        ##################################################
        # Connections
```

```
                ##################################################
                self.connect((self.digital_ofdm_mod_0, 0), (self.digital_ofdm_sync_pn_0, 0))
                self.connect((self.gr_throttle_0, 0), (self.gr_vector_sink_x_0, 0))
                self.connect((self.digital_ofdm_sync_pn_0, 0), (self.gr_throttle_0, 0))
                self.connect((self.digital_ofdm_sync_pn_0, 1), (self.gr_vector_sink_x_1, 0))
                self.connect((self.random_source_x_0, 0), (self.gr_vector_sink_x_2, 0))
                self.connect((self.random_source_x_0, 0), (self.digital_ofdm_mod_0, 0))

        def get_samp_rate(self):
                return self.samp_rate

        def set_samp_rate(self, samp_rate):
                self.samp_rate = samp_rate

if __name__ == '__main__':
        parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
        (options, args) = parser.parse_args()
        tb = top_block()
        tb.Run(True)
```

Second code (Reading data and plotting them)

```
from gnuradio import digital
import pylab
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class top_block(grc_wxgui.top_block_gui):
```

```
    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Top Block")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000

        ##################################################
        # Blocks
        ##################################################
        self.gr_vector_sink_x_0_0_0 = gr.vector_sink_f(1)
        self.gr_vector_sink_x_0_0 = gr.vector_sink_f(1)
        self.gr_vector_sink_x_0 = gr.vector_sink_b(1)
        self.gr_throttle_0 = gr.throttle(gr.sizeof_char*1, samp_rate)
        self.gr_file_source_0_0_0 = gr.file_source(gr.sizeof_float*1,
"/home/mohamed/Desktop/Simulations/sync_pn_analysis/ofdm_sync_pn-theta_f.dat",
 False)
        self.gr_file_source_0_0 = gr.file_source(gr.sizeof_float*1,
"/home/mohamed/Desktop/Simulations/sync_pn_analysis/ofdm_sync_pn-mf_f.dat", False)
        self.gr_file_source_0 = gr.file_source(gr.sizeof_char*1,
"/home/mohamed/Desktop/Simulations/sync_pn_analysis/ofdm_sync_pn-peaks_b.dat", False)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.gr_file_source_0, 0), (self.gr_throttle_0, 0))
        self.connect((self.gr_throttle_0, 0), (self.gr_vector_sink_x_0, 0))
        self.connect((self.gr_file_source_0_0_0, 0), (self.gr_vector_sink_x_0_0_0, 0))
        self.connect((self.gr_file_source_0_0, 0), (self.gr_vector_sink_x_0_0, 0))

    def get_samp_rate(self):
        return self.samp_rate
```

```
        def set_samp_rate(self, samp_rate):
                self.samp_rate = samp_rate


if __name__ == '__main__':
        parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
        (options, args) = parser.parse_args()
        tb = top_block()
        tb.Run(True)
         fig = pylab.figure(1)
         sp = fig.add_subplot(1,1,1)
         p2 = sp.plot(tb.gr_vector_sink_x_0.data()[:1200], "r", linewidth=2, label="Peak signal")
         sp.hold(True)
         p2 = sp.plot(tb.gr_vector_sink_x_0_0.data()[:1200], "b", linewidth=2, label="Matched filter output")
         sp.hold(True)
        p2 = sp.plot(tb.gr_vector_sink_x_0_0_0.data()[:1200], "g", linewidth=2, label="Timing metric")
        sp.annotate("max corr. after 512 sample",xy=(512,1), xycoords='data', xytext=(-150, -30), textcoords='offset points', arrowprops=dict(arrowstyle="->
", connectionstyle="arc3, rad=0.2"))
        sp.annotate("peak is detected after 512+128 sample",xy=(640,1), xycoords='data', xytext=(-50, 30), textcoords='offset points', arrowprops=dict(arrow
style="->", connectionstyle="arc3, rad=0.2"))
        sp.grid()
        sp.legend()
         pylab.show()
```

References

[1] Venkat vinod Patcha. "EXPERIMENTAL STUDY OF COGNITIVE RADIO TEST-BED USING USRP". Thesis by B.TECH in Electronics and Communication Engineering Padmasri Dr. B.V. Raju Institute of Technology (A_liated to JNTU), India, 2008. August, 2011.

[2] 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.

[3] Alexandra Martinez Torio. "Software Defined S-Band Ground Station Transceiver for Satellite Communications". Institute of Telecommunications, E389 Department of Electrical Engineering Vienna University of Technology.

[4] Tuan Ta. "Synchronization in OFDM". Department of Electrical and Computer Engineering, University of Maryland at College Park, September 2010.

[5] Grace R Woo. "Demonstration and Evaluation of Co-channel DPSK Source Separation", Master of Science in Media Arts and Science, Massachusetts Institute of Technology, June 2007.

[6] Patrick Ellis & Scott Jaris Advisors Dr. In Soo Ahn & Dr. Yuefeng Lu. "Implementation of Software-Defined Radio Using USRP Boards", May 10, 2011.

[7] GNU Radio. URL: http://www.gnu.org/software/gnuradio/.

[8] Dawei Shen. "GNU Radio Installation Guide - Step by Step", May 19, 2005.

[9] Fredric j harris . "Multirate Signal Processing for Communication Systems"

[10] Blossom, Eric, Johnathan Corgan, Matt Ettus, and Tom Rondeau. GNU Radio. 2006. Web. 27 Feb. 2011.

[11] Fredric harris ."Band Edge Filtering and Processing for Timing and Carrier Recovery"; Communications and Signal Processing Institute ,San Diego State University , San Diego, California

[12] Wen Li, Senior System Engineer, Advanced Analog Product Group ,and Jason Meiners . "Introduction to phase-locked loop system modeling"

[13] Thomas Schmidt. "CRC Generating and Checking"

[14] Kushal Shah. " Audio Streaming over FM band between USRP1 and USRP2" ,Phase II ,Project Report

[15] Oussama Sekkat . "The FPGA"

[16] Arief Marwanto, Mohd Adib Sarijari, Norsheila Fisal, Sharifah Kamilah Syed Yusof, Rozeha A.Rashid . "Experimental Study of OFDM Implementation"

[17] Jesper M. Kristensen : ppt : GNU Radio , An Introduction

[18] Hlaing Minn, *Member, IEEE*, Vijay K. Bhargava, *Fellow, IEEE*, and Khaled Ben Letaief, *Fellow, IEEE* . " A Robust Timing and Frequency Synchronization for OFDM Systems " .

[19] Firas Abbas . "Simple User Manual for Gnuradio 3.1.1"

[20] Andreas Muller. "DAB , Software Receiver Implementation"

[21] Fredric J.Harris , Senior Member , IEEE and Michael Rice Senior Member , IEEE. " Multirate Digital Filters for Symbol Timing Synchronization in Software Defined Radios "

[22] Tuan Ta ."Synchronization in OFDM ".

[23] J eff Feigin . "Practical Costas loop design"

[24] Dawei Shen. "Writing a Signal Processing Block for GNU Radio"

[25] http://gnuradio.org/doc/doxygen/index.html

[26] http://gnuradio.org/doc/doxygen/index.html

[27] Firas Abbas. "Simple Gnuradio User Manual"
http://www.scribd.com/doc/9688074/Simple-Gnuradio-User-Manual-v10

[28] http://www.tutorialspoint.com/python/index.htm

[29] GRC tutorials
www.csun.edu/~skatz/katzpage/sdr_project/sdr/**grc_tutorial**1.pdf
www.csun.edu/~skatz/katzpage/sdr_project/sdr/**grc_tutorial**3.pdf
www.csun.edu/~skatz/katzpage/sdr_project/sdr/**grc_tutorial**4.pdf
www.csun.edu/~skatz/katzpage/sdr_project/sdr/**grc_tutorial2**.pdf

[30] loading_sd_card_image for usrp2
www.csun.edu/~skatz/katzpage/sdr.../**loading_sd_card_image**.pdf

[31] GNU radio mailing list

http://lists.gnu.org/archive/html/discussgnuradio/

[32] http://www.trondeau.com/blog/2011/8/13/control-loop-gain-values.html

[33] http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[34] http://gnuradio.org/redmine/projects/gnuradio/wiki/TutorialsWritePythonApplications

Future Work

There is a great scope for improving the current project of OFDM. The effective area of research and development work includes :

1. Extending the project to increase the spectral efficiency and decrease BER using methods of channel coding and other communication concepts.
2. Implementation of communication standards such as WI-FI and LTE.
3. Add multiple users. We would like to have a group with three kits, representing two users and a base station. We would also like to have multiple antennas.