COOPERATIVE RELAYING USING USRP AND GNU RADIO

A THESIS IN
Electrical Engineering

Presented to the Faculty of the University
Of Missouri-Kansas City in partial fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

BY

GANGA MANJUSHA YANDAMURI

Bachelors, Jawaharlal Nehru Technological University, 2010

Master of Science, University Of Missouri- Kansas City, 2013

Kansas City, Missouri

2013

COOPERATIVE RELAYING USING USRP AND GNURADIO

Ganga Manjusha Yandamuri, Candidate for the Master of Science Degree.

University Of Missouri-Kansas City, 2013

ABSTRACT

Wireless communication systems have shown a tremendous development in recent years. New technologies are born day to day. With today's technology, users can communicate with each other from any corner of the world. But wireless technologies are often prone to various effects like multipath fading, interference, low signal strength, reduced spectrum efficiency etc. which makes this system less reliable. Because of this reason, researchers are continuously working to develop technologies that can make the performance of a wireless system much better.

Cooperative Communications is one of the fastest growing research technologies that can enable efficient spectrum usage and create a reliable network. In traditional networks, the physical layer is only responsible for communication in between two nodes which are more hindered to the challenges of the network. Cooperative Communication creates an extra communication with the help of a Relay in between the terminals which thereby enhances the signal quality. We implement this strategy using GNU Radio and three Radios (USRP-Universal Software Radio peripheral) which act as a Transmitter, a Receiver and a Relay.

Our main goal is to verify the communication in between the two Radios (a Direct Link) and implement Cooperative communication by introducing a Relay in

between the two radios. The Relay is made to operate on Amplify & Forward and Decode & Forward scenarios. Characteristics like packet error rate (PER), bit error rate (BER) and character error rates are studied with respect to individual scenarios and overall bit error rate (BER) of the system is calculated. Then performance is compared against different scenarios dealing with obstructions, transmit and receive gains, and relaying approaches with the goal of determining which approaches are best in which scenarios.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled "Cooperative Relaying Using USRP and GNU Radio," presented by Ganga Manjusha Yandamuri, candidate for the Master of Science degree, and certify that in their opinion, it is worthy of acceptance.

Supervisory Committee

Cory Beard, Committee Chair

Associate Professor, School of Computing and Engineering.

Ghulam M. Chaudhry,

Professor & CSEE Department Chair

Ken Mitchell,

Associate Professor, School of Computing and Engineering.

TABLE OF CONTENTS

ILLUSTRATIONS

Figure Page

ACKNOWLEDGEMENTS

I would like to thank my Advisor Dr. Cory Beard for his immense support and guidance throughout the research. While giving his students a degree of freedom to work and extending help in each and every challenge of the project are deeply appreciated. It is his excellent teaching skills in Wireless Communications, motivated me take a step towards thesis. His enthusiasm in creating new ideas and genuine interest in the student's research work are commendable. He is not only known for being a knowledgeable person but also known for his kindness, compassion and willingness to help. I have not only got an opportunity to learn technical concepts from him but also to be a better person. I am very lucky to be a part of his research team and much fortunate to be his student.

Special thanks to my project partner Aklilu A. Gebremichail for his constant support and encouragement. This project is incomplete without his guidance on USRP and GNU Radio.

Also, I would like to thank my lab partners Mustafa and Todd for their advices and for being patient with the noises from radios (while testing music transmission).

Last but not least, my sincere gratitude to my parents and friends for supporting me through all phases of my life and believing in me.

To my Parents ……

CHAPTER 1

INTRODUCTION

1.1 Overview

From past few decades, there has been a rapid increase in the demand for wireless devices and applications which has led to significant development in wireless communications. Users expect high data rates without any possible interruptions. Improving the signal quality and meeting the expectations of the user is not an easy thing to do with the environments we live in. The RF signal propagation has faces a lot of channel impairments like small scale and large scale fading, path loss and multi path propagation. The effect of multipath propagation has grabbed a lot of attention and is considered to be a serious problem.

Multipath fading occurs in an environment where the transmitted signal is received in various forms at the receiver. This can come from reflections, diffractions, scattering, or transmission through materials. The destination receives multiple copies of the signal that has different magnitudes and phases. When the phases match they produce a strong signal and when not, they produce a weak signal or no signal. Of course, there is high probability for the signal to arrive with different phases.

Researchers have worked hard on developing various approaches to resolve this issue or to at least provide the signal that is least affected by Multipath fading. One of those approaches is MIMO (Multiple Input and Multiple Output) where the signal is transmitted over independent and individual paths that are distributed in frequency, time and space and combined at the receiver. This approach is called Spatial diversity which

can improve the system performance. But, the limitation of size in the mobile devices makes this approach less practical for now. The signal quality can still be improved without the use of multiple antennas on the user end by a technique called Cooperative Relaying where the signal strength is improved by introducing a Relay in between the Transmitter and the Receiver. The Relay acts like a mediator and helps the TX by forwarding the signal to the RX, especially when the link in between TX and RX is not good enough to carry a signal.

1.2 Objective

The main objective of our thesis project is to implement and test the performance of Cooperative Relaying in a lab environment. Simulation and analytical results are available, but in actual implementations we can see the value and performance of Cooperative Relaying first hand. We have chosen to use USRP and GNU Radio, which are widely used software defined radio hardware and software platforms.

Our approach is as follows. Firstly, the signal performance in between TX and RX is verified in both LOS (Line of sight) and obstruction environments. Secondly, a Relay is introduced in between the source and the destination. There are two ways of implementing Relay operation. They are to use Amplify and Forward or Decode and Forward. Both of these cases are implemented with the help of GNU Radio Companion (GRC) flow graphs. Thirdly, a few necessary changes are made in the Python code at the TX and RX in order to observe the signal quality in terms of statistics like packet error rates, bit error rate, character error rate and overall BER of the system when the signals

are combined using Cooperative Relaying. Also, the results show the individual number of times the Relay path was better than direct path and vice versa.

CHAPTER 2

BACKGROUND

Before discussing the implementation, the general working of the radio integrated with the software is introduced. The documentation of the systems that are used in this thesis is lacking, so first section provides thorough information that is needed.

2.1 Software Defined Radio (SDR)

SDR is a reconfigurable communication system where the functions of physical components like modulators, demodulators, amplifiers etc. in a radio can be implemented by means of software. Different input signals can be processed without the need of changing the hardware. Multiple radios can be integrated together by just changing a few parameters in the software. At one instant a radio can be an AM receiver, a voice transceiver, a digital date transceiver etc. by just making few significant changes in the software block diagram. Upgrades are simple, quick, and of low cost. The flexibility of software-based applications makes SDR an excellent choice to avoid issues with compatibility and hardware reusability. Before we start discussing the functionality of Software Defined Radio, let's take a look at the important topic of Sampling Rate which is considered to be most fundamental.

2.1.1 Sampling Rate

Sampling rate is defined as the number of samples per unit time taken from a continuous signal to make a discrete signal. Folds in the figure below represent the

frequency axis that are integral multiples of one half of the sampling frequency f_s. According to Nyquist's theorem, the signal can be represented by discrete samples if the sampling frequency is at least twice the bandwidth of the signal. Hence the area in between the frequency points is a Nyquist Zone. [1]

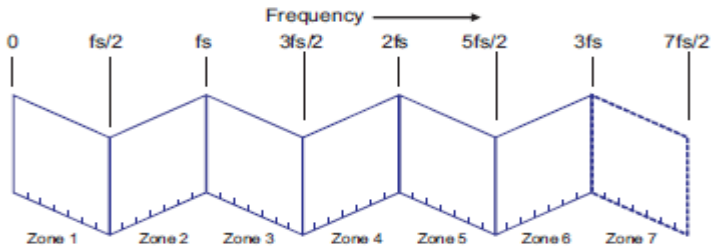

Fig 1: Signal Representation

Signals when mounted upon each other in a spectrum, create an aliasing effect and can no longer be separated. If this effect occurs during sampling, the sampled data is corrupted and can never be retrieved.

2.1.2 Baseband Sampling

The baseband signal has frequency components that start from f_s=0 to a maximum frequency. To prevent aliasing, we have to make sure that all the signal energy falls only in the first Nyquist zone. It can be done by either inserting a low pass filter to eliminate all the signals above $f_s/2$ or by increasing the sampling frequency so all signals can fall below $f_s/2$.

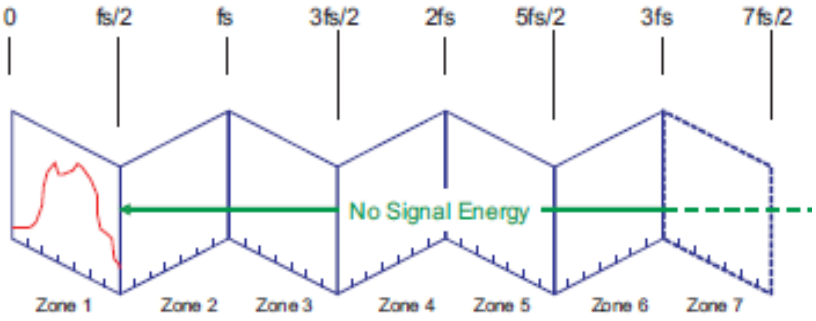

Fig 2: Base Band Sampling

2.2 Detailed Understanding Of A Radio Transmission

Figure 3 shows the processes involved in radio transmission, which are discussed in detail below.

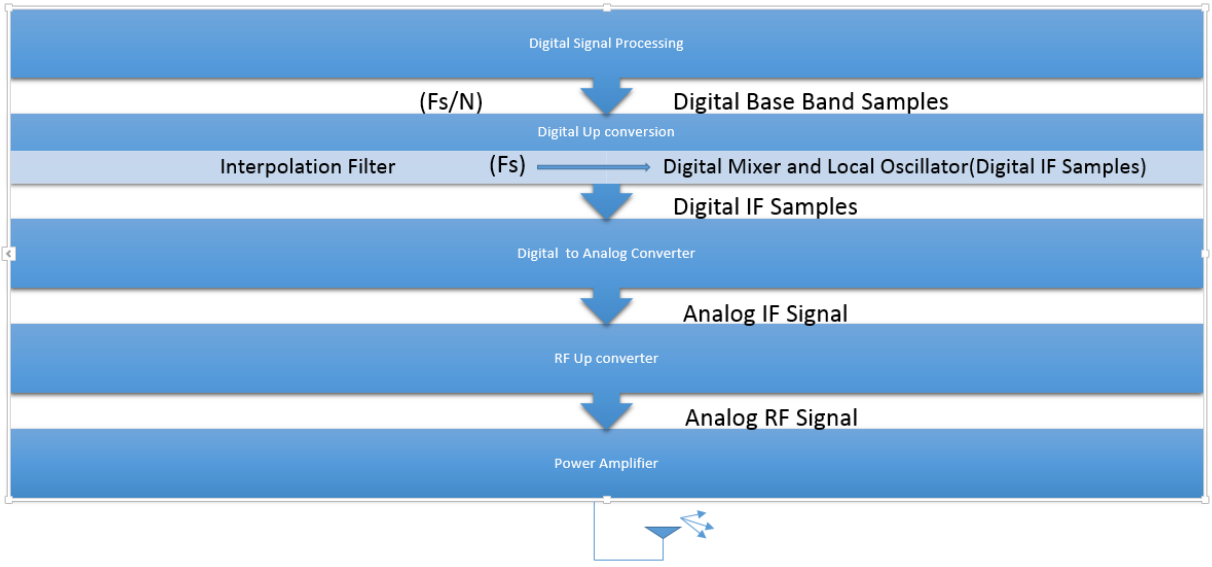

Fig 3: Block Diagram Of A Radio Transmitter

2.2.1 Digital Signal Processing (DSP)

The input to the transmit side of an SDR is a digital baseband signal generated by the sampling stage. The input can be of different types, audio, analog, digital data, videos, pictures etc.

2.2.2 Digital Up Counter

This translates the baseband signal into an Intermediate Frequency (IF).

A DUC has three important sections:

- Interpolation filter
- Digital Mixer
- Digital Local Oscillator

The digital mixer and digital local oscillator translate the baseband samples to the IF frequency. The IF translation frequency is determined by the local oscillator. A mixer generates one output sample for two inputs. The output generated by the mixer must be equal to the D/A sample frequency f_s. Therefore, the sample rates of both the inputs should be f_s. The sample rate of local oscillator is already fs. Since, the input baseband frequency is much lower than f_s, an interpolation filter is used. [2]

The interpolation filter increases the baseband signal frequency by a factor of N known as the Interpolation Factor. [1]

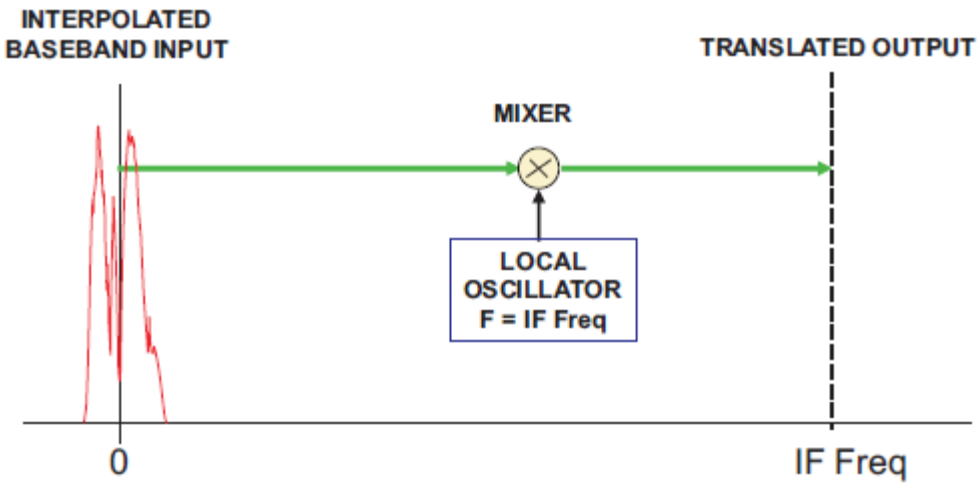

Fig 4: Digital Up Converter Process

2.2.3 D/A Converter

A digital-to-analog converter converts the digital IF samples to an analog IF signal.

2.2.4 RF Up Converter

This converts the analog IF signal to RF frequencies.

2.3 Detailed Understanding Of A Radio Reception

Figure 5 shows the processes involved in radio transmission, which are discussed in detail below. First, the analog RF signals from the antenna are sent to the RF Tuner.

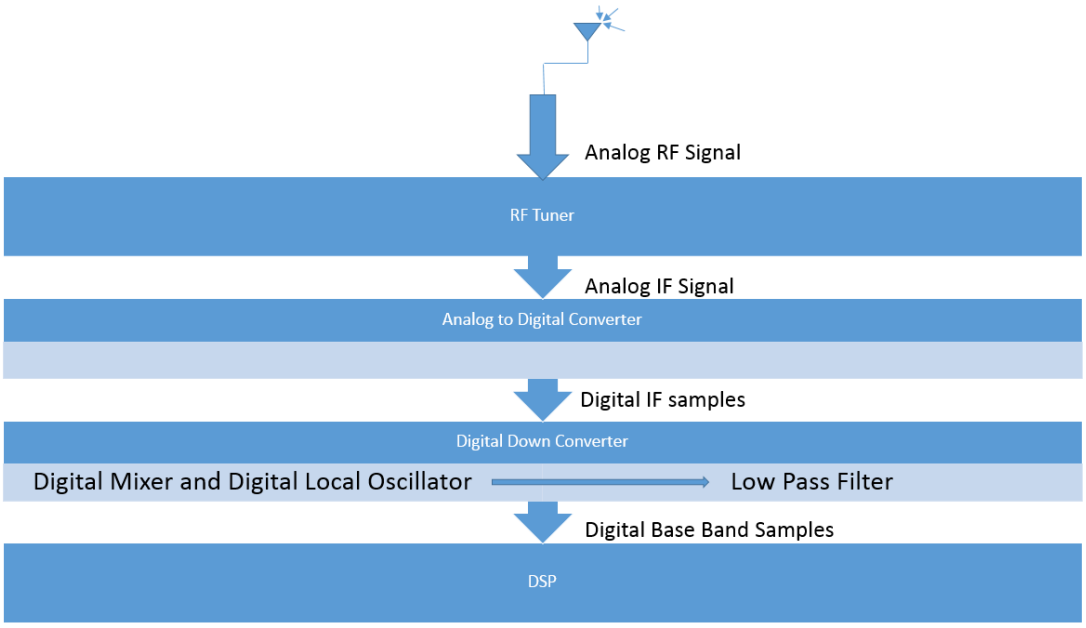

Fig 5: Block Diagram Of A Radio Receiver

2.3.1 RF Tuner

The RF Tuner has an amplifier, a mixer, an oscillator. The RF signal is amplified and then fed as one of the inputs to the mixer .The other input to the mixer is a signal from a local oscillator whose signal frequency is determined by tuning control of the radio. The mixer converts the signal to the Intermediate frequency (IF).

2.3.2 A/D Converter

Once an analog IF signal is obtained, then it is passed through the converter to convert them into digital IF samples.

2.3.3 Digital Down Converter (DDC)

An SDR consists of a DDC which is typically an FPGA IP and is considered to play a key role in processing of the signal.

A DDC has three sections:

- A Digital Mixer
- A Digital Local Oscillator
- An FIR low pass filter

The digital mixer and local oscillator translate the digital IF samples down to baseband. The low pass filter limits the signal bandwidth.

Since the output of the FIR filter is bandlimited, the Nyquist theorem allows us to lower the sampling rate. If we keep only one out of every N samples then sampling rate is reduced by a factor of N called the Decimation factor. Now that the sampling rate is reduced it is always good to make sure the Nyquist sampling theorem is maintained relative to the new sampling rate to avoid aliasing in the digital signal. So, a low pass filter is used as an ant-aliasing filter to reduce the bandwidth of the signal before it is down sampled. [1] [3]

Now it's clear that DDC performs two operations:

- Frequency translation with tuning control local oscillator.
- Low pass filtering with bandwidth controlled by the decimation setting.

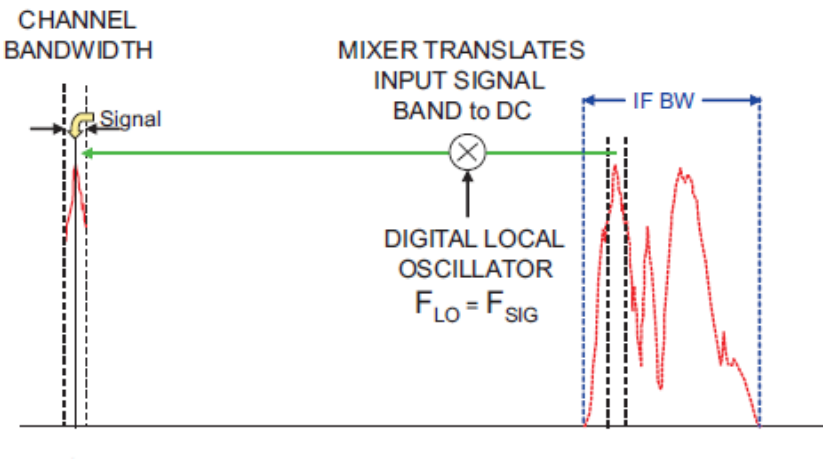

Fig 6: Digital Down Converter Process

2.3.4 DSP(Digital Signal Processor)

The digital baseband samples are then fed to the Digital Signal Processing block where decoding, demodulation and other processing tasks take place.

In our implementation of SDR using USRP radios, the components include:

1. Antennas –to transmit and receive radio signals on the frequency of interest.
2. Radio integrated with daughter boards – These serve as a digital baseband and IF section of the communication system.
3. Personal computer integrated with GNU Radio – to process the radio signals. The signals are not processed on the radio, but rather on the PC and then fed through a USB connection.

2.4 Advantages Of SDR

There are several important benefits to SDR, which is making its popularity increase as the hardware becomes available that can support it.

- Elimination of analog hardware and its cost, resulting in simplification of radio architectures and improved performance.
- Software can be reused across radio "products", reducing development costs dramatically.
- Over-the-air or other remote reprogramming, allowing "bug fixes" to occur while a radio is in service, thus reducing the time and costs associated with operation and maintenance.
- New features and capabilities to be added to existing infrastructure without requiring major new capital expenditures.
- Remote software downloads, through which capacity can be increased, and capability upgrades can be activated.
- Reconfigurable to suit customer requirements.
- The ability to receive and transmit various modulation methods using a common set of hardware.
- The chance for new experimentation.

2.5 Disadvantages Of SDR

There are a few disadvantages of SDR, however.

- Difficulty of writing software for various target systems.
- Fear of the unknown
- Documentation is rare in relation to hardware radios.

2.6 Preparing For Implementation

In the establishment of SDR capabilities in our laboratory, several activities had to be performed.

First, we had to choose the most suitable Software Defined Radio among those existing on the market. Below are the few of those:

- Microsoft Research Software Radio Platform for Academic Use(SORA):

  Microsoft offers this academic kit for research purposes. The estimated price for this kit $4500-$5500. It has multi-core PC, RCB board, RF-front end software.

- Data Soft's Typhoon SDR Development Platform:

  It operates an SDR full duplex transceiver system in the 400 MHz – 4 GHz band with the ability to process signal bandwidth ranging from 50 KHz to 20MHz. It supports GNU Radio and the Click modular router as software architecture. This platform aims mostly at high end users and costs $10,000 per unit.

- CRC's Coral Cognitive Radio Platform:

  Coral offers an experimental Cognitive Radio Platform operating in the 2.4 GHz and 5.8 GHz ISM Bands, costing at $6000. It is primarily intended for improved spectrum use research. It supports different modulation techniques. This platform mostly directs towards cognitive radio which is not a primary focus of Cooperative Relays.

- Ettus Research's USRP N210:

Ettus has established themselves as a best buy provider of platforms for research. It offers several platforms – USRP1, USRP2 and USRP N210 which differ in the levels of instantaneous bandwidth, re-programmability of FPGA, type of interface to connect to the computer (USB or Ethernet) and price. It is compatible with GNU Radio and is the best product for academic research.

The USRP1 developed by ETTUS was chosen as the most suitable SDR for our thesis project.

2.7 USRP (Universal Software Radio Peripheral)

USRP is an open source computer hosted SDR designed by Ettus Research, LLC and National Instruments. The USRP product family is intended to be a comparatively inexpensive hardware platform for software radio and is used widely in research labs, universities etc. USRPs connect to a host computer through a high speed USB where the software controls this radio to transmit and receive information.

A typical USRP can accommodate one motherboard and four daughter boards, two for receiving and two for transmitting. A USRP can simultaneously transmit and receive on two antennas. [4]

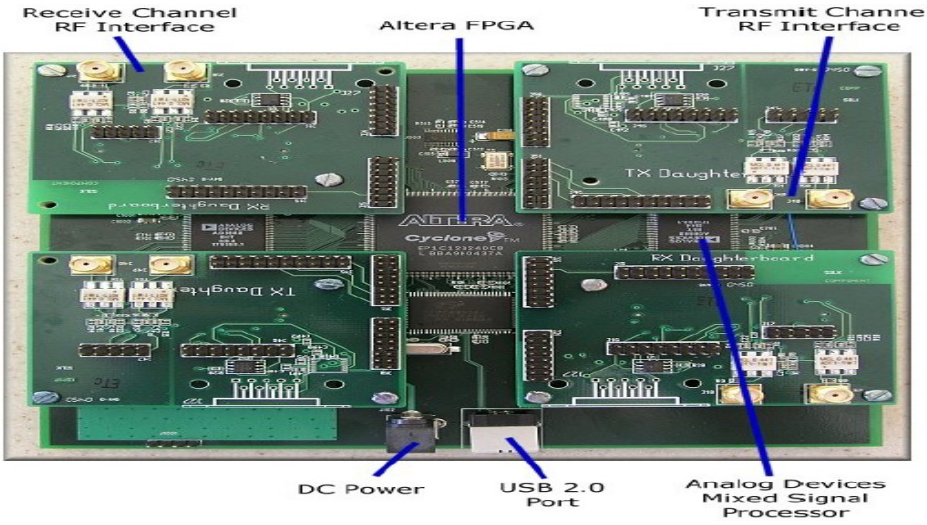

Fig 7: USRP With Four Daughterboards

The motherboard in a USRP has Clock generation, FPGA, DAC, ADC, power regulation and host processor interface subsystems which are the basic components required for processing of baseband signals. A modular front end called daughterboard is used for analog operations such as DAC, ADC etc. According to USRP manual [5], the following are the components:

2.7.1 USRP Motherboard

The heart of the motherboard is Altera Cyclone EP1C12 FPGA. It has 4 Input and 4 Output channels. These are connected to 4 high speed Analog to Digital Converters (ADC) and 4 high speed Digital and Analog Converters (DAC). Each ADC processes 12 bits per sample i.e. in total of 64 Msamples/second and each DAC processes 14 bits per sample i.e. a total of 128 Msamples/second. The FPGA is in turn connected to a USB2 interface chip called Cypress FX2 on to the computer.

2.7.2 ADC Section

This is used when the signal is received .i.e., converting the analog signal from the air to the digital format. As discussed above, there are 4 ADCs that have a sampling rate of 64 Msamples/Second. They can bandpass sample signals up to 200 MHZ. An IF frequency as high as 500 MHz can also be digitized but it causes several decibels of loss in the signal. i.e. sampling a signal with IF larger than the 32 MHz introduces aliasing. Hence, the best band of signal can be mapped to 32 MHz. As the frequency of the sampled signal goes higher, the SNR gets degraded by jitter. Recommended upper limit of the sampling rate is 100MHz.

The full range of the Analog to Digital Converter is 2Vp-p and the input is 50 Ohms differential that produces 10 mW or 10 dBm. The programmable gain amplifier (PGA) is software programmable before the ADC. It amplifies the input signal (if the signal is weak) to utilize the entire range of ADC. The maximum range of the PDC is 20dB. When the gain is set to zero, the input is 2 Vp-p and when set to 20 dB, only a 0.2 Vp-p differential input is required to meet the full scale.

There is no need to provide a DC bias when the signal is AC coupled as long as the internal buffer turned on, which provides a 2V bias. If the signal is DC-coupled, a DC bias of Vcc/2(1.65V) should be provided to both negative and positive outputs with the internal buffer turned off. An ADC provides a reference voltage of 1V.

There are 8 Auxiliary analog input channels connected to low speed 10 bit ADC inputs (labeled AUX_ADC_A1_A, AUX_ADC_B1_A, AUX_ADC_A2_A, AUX_ADC_B2_A, AUX_ADC_A1_B, AUX_ADC_B1_B, AUX_ADC_A2_B, and

AUX_ADC_B2_B) which can be read from software. These ADCs can convert up to 1.25M Samples/Second and have a bandwidth of around 200 KHz. These analog channels are useful for sensing the RSSI signal levels, temperatures, bias levels, etc. The USRP motherboard connectors have 2 independent analog input channels (AUX_ADC_A1_A and AUX_ADC_B1_A for RXA and AUX_ADC_A2_A and AUX_ADC_B2_A for TXA). There is also AUX_ADC_REF which can provide a reference level for gain setting if it is necessary.

2.7.3 DAC Section

The DAC Section is used at the transmitting path. As discussed above; the DAC has a sampling rate of 128 MS/s whose Nyquist frequency is 64 MHz. It can supply 1 Vp-p and the input is a 50 Ohm differential load. The output frequency range which yields a good signal is from DC to 44 MHz. A PGA which is software programmable located after DAC provides 20dB of gain. The DAC signals vary in between 0 to 20 mA and can be converted into voltages by placing a resistor.

Additionally, there are 8 analog output channels connected low-speed 8bit DAC outputs. These are AUX_DAC_A_A, AUX_DAC_B_A, AUX_DAC_C_A, AUX_DAC_A_B, AUX_DAC_B_B and AUX_DAC_C_B. These DACs can be used for supplying various control voltages such as external variable gain amplifier control. In addition, there are two additional DACs (labeled AUX_DAC_D_A and AUX_DAC_D_B) which are constructed using a 12 bit sigma-delta modulator with external simple low pass filter. The USRP motherboard connectors (RXA and TXA)

share one set of the 4 analog output channels (AUX_DAC_A_A to AUX_DAC_D_A for RXA and TXA)

2.7.4 Auxiliary Digital I/O ports

The USRP motherboard has 32 bits for IO_RX and 32 bits for IO_TX constituting of high speed 64 bit digital I/O ports. These are connected to the connecters of daughterboard interfaces (RXA, TXA, RXB, TXB). Each of these connectors has 16 bit digital I/O bits which are controlled by writing and reading operations to the FPGA registers independently and can be configured either as a digital input or a digital output.

Several operations are performed by using these pins like, automatic receive and transmit mode, power supply control, synthesizer lock detection etc. Also, these are used for debugging FPGA implementations when logical analyzer is connected to it.

2.7.5 FPGA

The FPGA plays an important role in the USRP system. The main function of the FPGA is to reduce the data rates to a value that can be passed over to USB2.0 and also to perform high bandwidth math. FPGA circuitry and a USB microcontroller are programmable over a USB2 bus. [6]

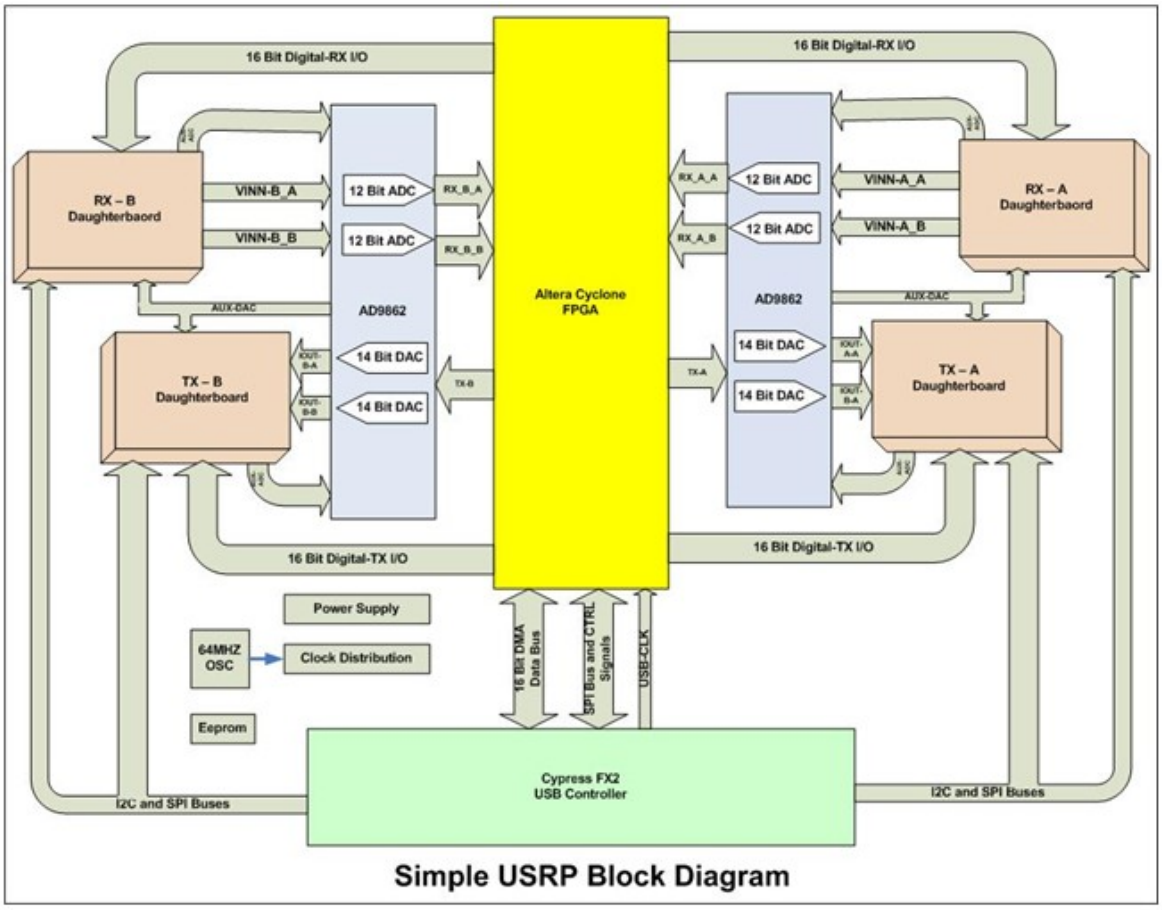

Fig 8: Block Diagram Of USRP Functionality

2.7.5.1 DDC (Digital Down Converter)

DDC is implemented with 4 stages of a cascaded Integrated comb (CIC) filter. These filters use only adders and delays to deliver a high integrated performance. 31 other tap half band filters and CIC filters put together are used for spectral shaping and out of band signal rejection. A standard FPGA has two DDCs.

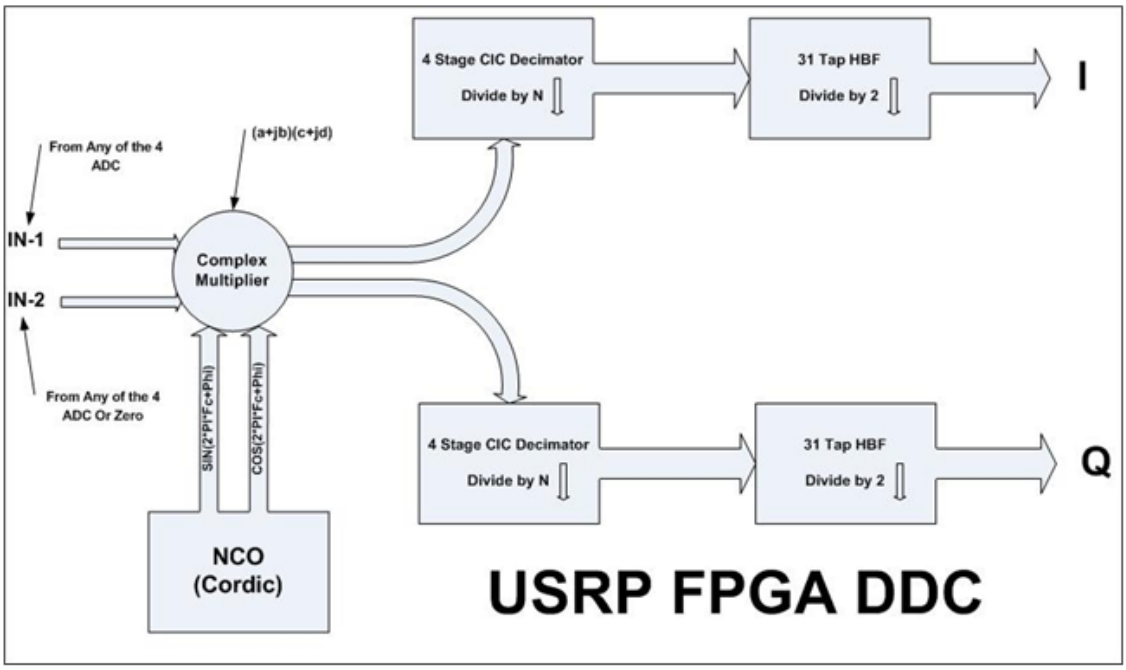

Fig 9: Block Diagram Of A FPFA DDC In USRP

Firstly, the complex input signal (IF) is converted to Base band Signal. It is done by multiplying the signal with a constant frequency which is usually an IF .The resulting signal produced is also complex and centered at 0. Secondly, the signal is decimated by a factor of 'D'.A decimator is nothing but a combination of a low pass filter and a down sampler. Where the low pass filter selects a band of frequency from a big stream of frequencies. And thirdly, the down sampler dispreads the signal from [-Fs,Fs] to [-Fs/D,-Fs/D] narrowing the frequency by a factor of 'D'.

Here we use decimation in order to meet the data rate of the USB2.0 and also can keep up the computer's computing capability. All the samples sent over USB interface are 16 bit signed integers in IQ format ,i.e . 16 bit I and 16 bit Q resulting in 32 bits per sample or 4 bytes per sample. Finally complex I/Q signal enters the computer USB.

If there are multiple channels and are interleaved, then the I/Q format of sending over USB is I0 Q0, I1, Q1, I2, Q3 etc.In this case, all the input channels must be at the same decimation ratio.

2.7.5.2 Digital Up Converter (DUC)

The DUC is found at the transmitter path. The base band I/Q complex signal is interpolated and up converted to the IF band and is sent to the DAC before transmitted over the air.

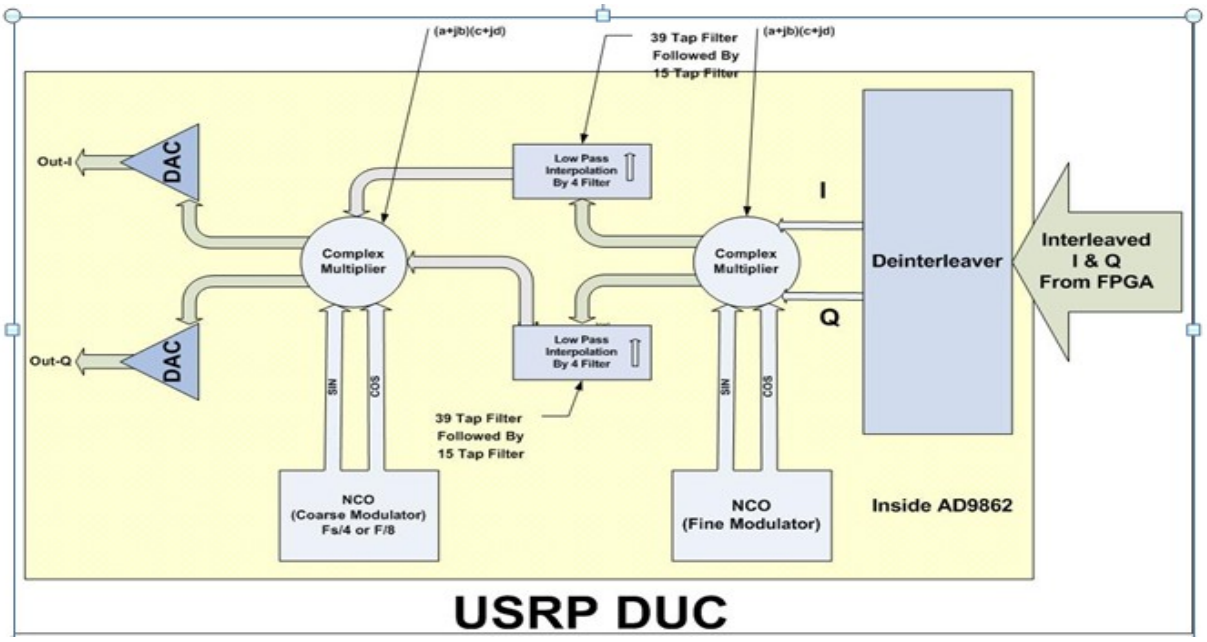

Fig 10: Block diagram Of A Digital Up Converter In A USRP

The DUC is located in AD9862 CIDEC chips but not in the FPGA. The only transmitted signal processing blocks in the FPGA are the CIC interpolators. The interpolator outputs can be routed to any of the 4 CODEC inputs. Multiple TX channels should have same data rates and can have different TX and RX rates. The modes operate independent lof each other but the overall data rate should always be 32Mbytes/second or less.

2.7.6 Daughterboards

A mother board has 4 slots to plugin 2 TX basic daughter boards and 2 RX daughter boards. Each daughter board in turn has 2 slots TXA and RXA. And each slot can access 2-4 high speed A-D/D-A converters. If real sampling is used the daughter boards develops two independent RF sections having 2 antennas. If IQ sampling is used, each board can just support a single RF section. Two SMA connectors on each daughterboard are usually used to connect the input or output signals.

Each daughterboard has an I2C EEPROM which identifies the board of the system. It also stores values like DC Offsets or IQ imbalances. EEPROM is not programmed as compared to other components on board.

Each TX daughterboard has two outputs (IOUTP_A/IOUTN_A and IOUTP_B/IOUTN_B) sampled at 128MS/second and each RX daughterboard has 2 differential analog inputs (VINP_A/VINN_A and VINP_B/VINN_B) sampled at 64 MS/second.

2.7.7 Basic TX/RX Daughterboards

The inputs from ADC and outputs from DAC are directly transformer-coupled to SMA connectors without amplifying or filtering the signal. The basic TX and RX give access to all the signals on the daughterboard interface and utilize a logic analyzer connector for 16 general purpose IOs. The pins are used to perform debugging of FPGA by providing access to the internal signals.

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 with no mixers, filters, or amplifiers. The basic TX and basic RX give direct access to all of the signals on the daughterboard interface (including 16 bits of high-speed digital I/O, SPI and I2C buses, and the low-speed ADCs and DACs). Each of the Basic TX/RX boards has logic analyzer connecters for the 16 general purpose IOs. These pins are used to help debugging your FPGA design by providing access to internal signals.

2.7.8 Low Frequency TX/RX Daughterboards

The main difference in between using LF TX/RX and basic TX/RX is LF TX/RX use differential amplifiers instead of transformers. They also have 30MHz low pass filters for anti-aliasing.

2.7.9 TVRX Daughterboard

This daughterboard is used for receiving of the signals. It is a VHF and UHF system based on TV tuner module. It has a IF bandwidth of 6MHz and an RF frequency range of 50 MHz to 860 MHz.This is the only board which is not MIMO capable and has a typical noise figure of 8 dB.

2.7.10 DBSRX Daughterboards

This board is similar to TVRX daughterboard which is also receive only. But this daughterboard is MIMO capable and can power an active antenna via SMA connector. It also has a software controllable channel filter that can vary in the range of 1MHz to

60MHz. DBSRX can receive frequencies ranging from 700MHz to 2.4 GHz having a noise figure of 3-5 dB.

2.7.11 RFX Daughterboards

The family of RFX daughterboard is a complete transceiver system. It has a built-in Transmit and Receive switching feature so TX and RX can be on same port. It also has an independent local oscillator or RF synthesizers for both transmission and reception which enables a split operation. These are fully synchronous and MIMO capable.

2.7.12 Power

The USRP is powered by a 6V 4A AC/DC power converter. This converter is capable of 90-260VAC, 50/60 Hz operations. The USRP itself needs 5V of supply and 6V of supply is required for the daughter boards. It draws a 1.6 A with 2 daughterboards fixed on it.

2.8 USRP Version 1 And Version 2

We now compare the USRP1 and USRP2. Figures 11 and 12 show pictures of them.

Fig 11:USRP1

Fig 12:USRP2

2.8.1 Their Comparisons

Interface	USRP1	USRP2
ADC	12 bit ,64 MS/second	14 bit,100MS/second
DAC	14 bit,128 MS/second	16 bit,400 MS/second
Power	6V,3A	6V,3A

RF Band width	8 MHz at 16 bits	25 MHz at 16 bits
FPGA	Altera EP1C12	Xilinx Spartan 3 2000
Daughterboard Capacity	2 TX,2RX	1TX,1 RX
SRAM	None	1 M byte
Cost	$700	$1500

After analyzing the characteristics of the above two versions, we decided to choose the USRP1 over the USRP 2 because it meets our requirements at lower cost.

2.8.2 USRP1

Fig 13:USRP1 And Its Components

The Ettus Research's USRP1 is the original hardware of the USRP (Universal Software Radio Peripheral) family of products, which enables engineers to rapidly design and implement powerful, flexible software radio systems. The USRP1 provides

an entry-level platform with built-in MIMO expansion and a modular design allowing the hardware to operate from DC to 6 GHz. The architecture includes an Altera Cyclone FPGA, 64 MS/s dual ADC,128 MS/s dual DAC and USB 2.0 connectivity to provide data-to-host processors. The USRP1 includes connectivity for two daughterboards, enabling two complete transmit/receive chains. This feature makes the USRP1 ideal for applications that require high isolation between transmit and receive chains, or dual-band dual transmit/receive operation [7]. The USRP1 can stream up to 16 MS/s to host applications. On-board DDCs and DUCs provide 15 MHz of tuning resolution and adjustable sample rates.

The USRP Hardware Driver is the official driver for all Ettus Research products, and supports rapid development in a comprehensive environment. The USRP Hardware Driver supports Linux, Mac OSX and Windows. [8] [9]

2.8.2.1 RFX 900 Daughterboard With A Range Of 750-1050 MHz Rx/Tx

Fig 14:RFX900 Daughterboard

The RFX900 is a high-performance transceiver designed specifically for operation in the 900 MHz band. With a typical power output of 200 mW, and noise figure of 8 dB jumper settings, it can bypass an on-board SAW filter to allow operation

in a wider frequency range. Example application areas include cellular, paging, two-way radio and 902-928 MHz ISM band. [10]

2.8.2.2 VERT 900 Vertical Antenna (824-960 MHz, 1710-1990 MHz) Dual Band

This is an Omni directional antenna and has a 3dBi Gain. [11]

Fig 15:VERT900 Antenna

2.8.2.3 SMA-Bulkhead Cable

Fig 16: SMA-Cable

A 0.2M long SMA-M to SMA-F bulk head cable connects daughter board and the antenna [12].

2.8.3 Features Of USRP1

- Use with GNU Radio
- Modular Architecture: DC-6 GHz
- Connectivity for Two, Complete Tx /Rx chains
- Two Dual 64 MS/s, 12-bit ADC's
- Two Dual 128 MS/s, 14-bit DAC's
- DDC/DUC with 15 mHz Resolution
- Up to 16 MS/s USB Streaming
- USB 2.0 Interface to Host
- Auxiliary Digital and Analog I/O
- 25 ppm TCXO Frequency Reference

One way to control the USRP1 is through a computer with GNU Radio installed in addition to USRPs. This is our approach.

2.9 GNU

GNU is a free operating system developed by GNU Project which was intended to be compatible with UNIX. It was first initiated by Richard Stallman in 1983. Several versions of GNU have been released but there is no stable version till date. It's abbreviated as "GNU's not UNIX!" – a recursive acronym. Although GNU is like UNIX, it differs by being free software and containing no UNIX code.

2.9.1 GNU Radio

This is an open source software development tool kit that helps to process the signal through software defined radio. You can write applications to transmit and receive data streams through hardware.

It has various elements (blocks) like filters, modulators, demodulators, encoders, and decoders etc. which are typically found in a Radio. Primarily, it has a method of connecting these blocks and managing the data step by step. Since GNU Radio is software and handles only digital signals, the output of the transmitters and input of the receivers are always complex baseband samples. The shifting of the signal to the desired center frequency is done by the analog hardware.

Any data type can be passed from one block to other – bytes, float or other complex data types. There is also a feature where you can create a block you like and use it for your application, which makes GNU Radio user friendly and hence used to support real world radio systems and wireless communications mainly in academic and research environments.

Gnu Radio's applications are primarily written in the Python programming language while the signal processing path is written in C++ which makes GNU Radio a rapid application development environment.

2.9.2 Installing GNU Radio

There are several ways of installing GNU Radio, using

- PyBOMBS

- Build-gnu radio script
- Pre-compiled Binaries
- Pre-compiled binaries from Ettus Research
- Manual install for source

We chose to use the 'Build-gnu radio script' way because of its easy approach and compatibility with UBUNTU. We just needed to run a script that is available in www.gnuradio.org and the installation was done automatically. [13]

This downloads the installer (build-gnuradio) and makes it executable. It then downloads and installs all dependencies, downloads both UHD and GNU Radio from Git (which means it will automatically install the latest version from the 'master' branch), runs the make process, and installs it on your system). In most cases, simply running the script will do all you need to get a running GNU Radio system built from source. Also, you will have all the source code lying on your hard disk and therefore available for future modifications. It combines the flexibility of installing from source with the ease of using binaries and is recommended for most users of Ubuntu and Fedora.

2.9.3 GRC (GNU Radio Companion)

Although GNU Radio's process is completely written in Python, there is a tool called GNU Radio Companion (GRC) which allows you to construct blocks on the application you want to build. So, it is not necessary for the users to know the Python language to construct an application. Of course one needs to know Python if wanting to understand the back end functionality of the application/flow graph that is made.

The functionality of “Cooperative Relays using GNU Radio and USRP” is implemented in GRC and certain additions are being made in Python in order to obtain the desired output and performance analysis.

According to GNU Radio website [14] [15], the following are a few important parameters used in GRC.

2.9.4 Explanation of GRC Parameters

2.9.4.1 Variables

Variables map symbolic names to values. In GRC, a variable can define a global constant or a variable can be used in conjunction with a GUI to control a running flow graph. A variable holds a number that is available to all elements in the flow graph. Variables serve two purposes: First, parameters can use a variable as a way to share values. For example, if all parameters for sampling rate use the samp_rate variable, changing the samp_rate variable once is easier than modifying every parameter. Second, variables can also have a range (min and max) associated with them.

2.9.4.2 Variable Block

The variable block is the most basic way to use a variable in GRC. The ID parameter of the variable block is the "symbolic name". The symbolic name must be alpha-numeric (underscores allowed) and begin with a letter. To use the variable, simply enter the symbolic name into a parameter of another block.

2.9.4.3 Variable Controls

Certain blocks have callback methods that allow their parameters to be changed while executing flow graph. Variable controls in GRC use variables in combination with

callback methods to modify these parameters. If a parameter has a callback method, the parameter will be underlined in the block-properties dialog. The variable slider, variable text box, and the variable chooser block provide graphical widgets such as sliders, text boxes, radio buttons, and drop downs as variable controls. In addition, the variable sink block takes samples from a gnu radio stream and writes the samples to a variable.

2.9.4.4 String Evaluation

String parameters have a two-phase evaluation. First, GRC evaluates the parameter as it is. If the parameter does not evaluate to a string data type or the evaluation fails, then it is understood that the parameter had implied quotation. In this case, GRC will evaluate the parameter again with quotation marks; which will return a string with the exact code that was typed into the parameter window.

To use a variable inside a string simply type the name of the variable into the parameter: my_var. If the variable is not a string, cast the variable with Python's str function: str(my_var). Standard Python string functionality applies: "My Var = " + str(my_var).

Note: String parameter types also include the file open and file save types.

2.9.4.5 Flow Graph

A flow graph is an interconnection of signal processing blocks. GRC provides a scrollable window to place and connect various signal blocks.

2.9.4.6 Signal Block

Signal blocks perform all of the processing in a flow graph. For example: A signal block can be a filter, an adder, a source, or a sink. GRC represents signal blocks as

rectangular blocks. Each block has a label indicating the name of the block and a list of parameters.

2.9.4.7 Parameters

Parameters influence the function of a signal block. For example, a parameter can be a sampling rate, a gain etc. Most parameters for a signal block are displayed below its label.

2.9.4.8 Sockets

Sockets are the inputs and outputs of a signal block. Each signal block has certain sockets associated with it. For example, an adder has two input sockets and one output socket.

GRC represents a socket as a small rectangle attached to the signal block. The socket has a label indicating its function. Labels are usually named "in" or "out". Some labels are named "vin" or "vout" to indicate a vector type. Sockets are also colored to indicate their data type. Blue for complex, Red for float, Green for int, Yellow for short, and Purple for byte.

2.9.4.9 GR Buffer and GR Buffer Reader

Every block has a GR Buffer and a GR Buffer Reader. GR Buffer is used to hold the processed data and is at the output of the block. GR Buffer Reader acts as an input of the block which reads the data from its upstream GR Buffer. These two Buffers provide a channel for exchanging data and status updates.

Note: A source block doesn't have GR Buffer Reader since this block provides the data and has no prior block. Vice versa is true with the Sink Block.

2.9.4.10 Connections

A connection joins an input and an output socket. GRC represents connections by drawing a line between the two sockets. Connections must be between matching data types, including vectors.

2.9.5 Using GRC

2.9.5.1 Running The Interface

To run a .grc file use simulate and run buttons on the top nav. A flow graph generates a Python file called top_block.py.

2.9.5.2 Adding A Block

Select a block from the signal block tree menu. Double click the block. A new signal block will be placed on the screen.

2.9.5.3 Moving A Block

Left click a block and drag it around the flow graph. If you drag a block near a border that can be scrolled, wiggle the block to advance the scroll bar.

2.9.5.4 Rotating A Block

With a block selected, select rotate right or left from the tool bar.

2.9.5.5 Deleting A Block

With a block being selected, select delete from the tool bar. Short-cut keys: delete

2.9.6 Connecting Blocks

Left click on one socket and then left click on another socket. A connection will only be created between an input socket and an output socket, and the input socket may have no existing connections. An output socket may have unlimited connections. Red lines surrounding the connection indicate that the data types of the sockets do not match.

2.9.7 Modifying Parameters

Double click on a block or select it and choose properties from the edit menu. A dialog containing all the parameters for the signal block will appear. Some parameters are set via a drop down menu, and most must be typed in as characters. These parameters are usually numeric, representing sampling rates, gains, and amplitudes. Numeric parameters may contain mathematical expressions with variables.

2.9.8 Numeric Expressions

A numeric expression may contain any number of variables, numbers, and operators. There must be an operator between every pair of numbers/variables. Possible operators are + - * / ^. Numbers can be integers, decimals, floating-point, and complex. Python's built-in floating point and imaginary formats are used. Floating point numbers end in an ‘e’ followed by a signed integer. Imaginary numbers end in a ‘j’. Matching bracket pairs are allowed: (), [], {}. Variables are denoted by a leading '$' character.

2.9.9 Flow Graph Validation

1. Connections are between input and output sockets of the same data type. Different data types have different colors. Therefore, input and output colors must match. Invalid connections are highlighted red.

2. All sockets must be connected (except for the optional sockets). Disconnected sockets cause their signal block to have a red label.
3. All signal block parameters must be valid. For instance, numerical expressions can be parsed. Invalid parameters have colored red labels and cause their signal block to have a red label.

2.9.10 Running A Flow Graph

If a flow graph is valid, all parameters are valid and all sockets are connected. Choose run from the tool bar or press F5. A window will appear with any sliders or graphs that were added. To stop the flow graph, close the window, press stop in the tool bar, or press F7. Flow graphs can be run without the interface by running top_block.py.

2.9.11 A Simple Example Of GRC

Here is an example GRC implementation of a Band Pass Filter

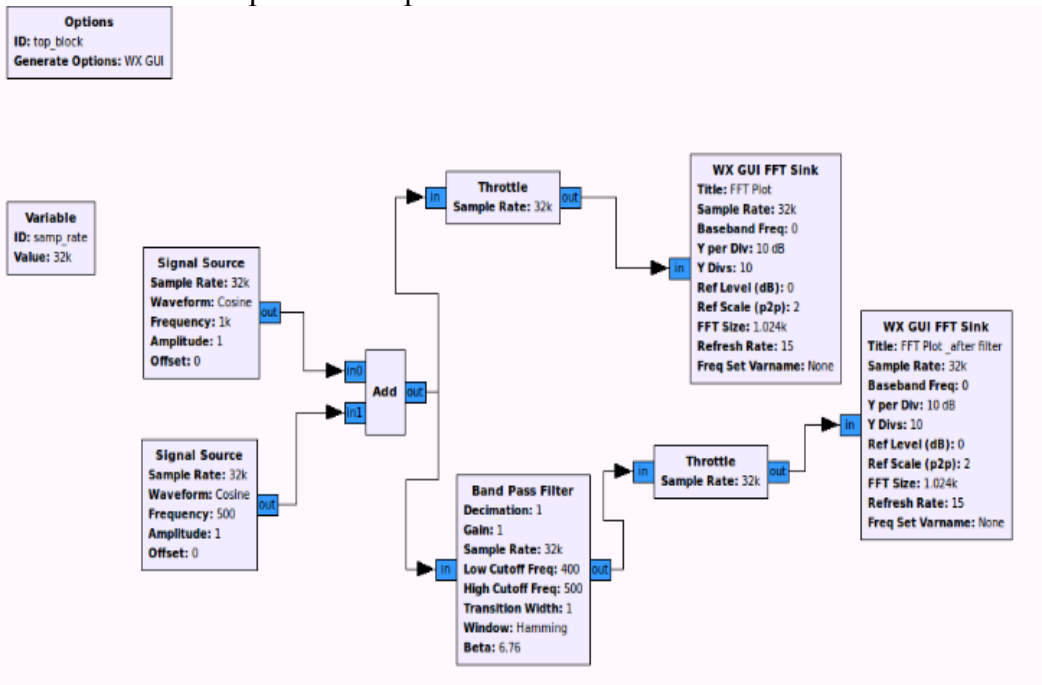

Fig 17: GRC Implementation Of A Band Pass Filter

This has two signal sources. A signal source has the important following properties:

- Sample rate: Sample rate is defined as the number of samples per unit of time take from analog (continuous) signal to make a discrete signal. For our example we are using a sampling rate of 32 kHz which means 32000 samples are being generated for unit of time.

Note: According to Nyquist theorem, the sampling rate should be at least twice the highest frequency of the signal.

- Signal Source: This option lets you choose which analog signal you want to use. Cosine is the analog signal we chose.
- Frequency and Amplitude: Any frequency can be set since its signal is not going to be transmitted over air. (We are not using any transmitter or receiver to send the signal.) For our example, we chose two frequencies 500 and 1000Hz.

Both of these signal sources are generating a Cosine wave at 32 kHz sampling rate and at different frequencies are added together to produce a resultant 1.5 kHz signal. A Throttle block is necessary in order to avoid congestion in the CPU.A The "Wx GUI FFT Sink" plot is used to check our outputs. An adder is used to combine the two signals and to produce a resultant 1.5 kHz signal.

The 1.5 kHz signal is sent through the band pass filter to avoid noises/other signals. A frequency range of 400 to 500 Hz is given so as to pass the 500 Hz signal. Note: The decimation factor is set to 1. The other GUI plot is used to see the difference in between the filtered and non-filtered signals.

The plot below shows the difference between the filtered (above) and non-filtered signal (below). The highest frequency is 16 kHz since the sampling rate of 32 kHz is used.

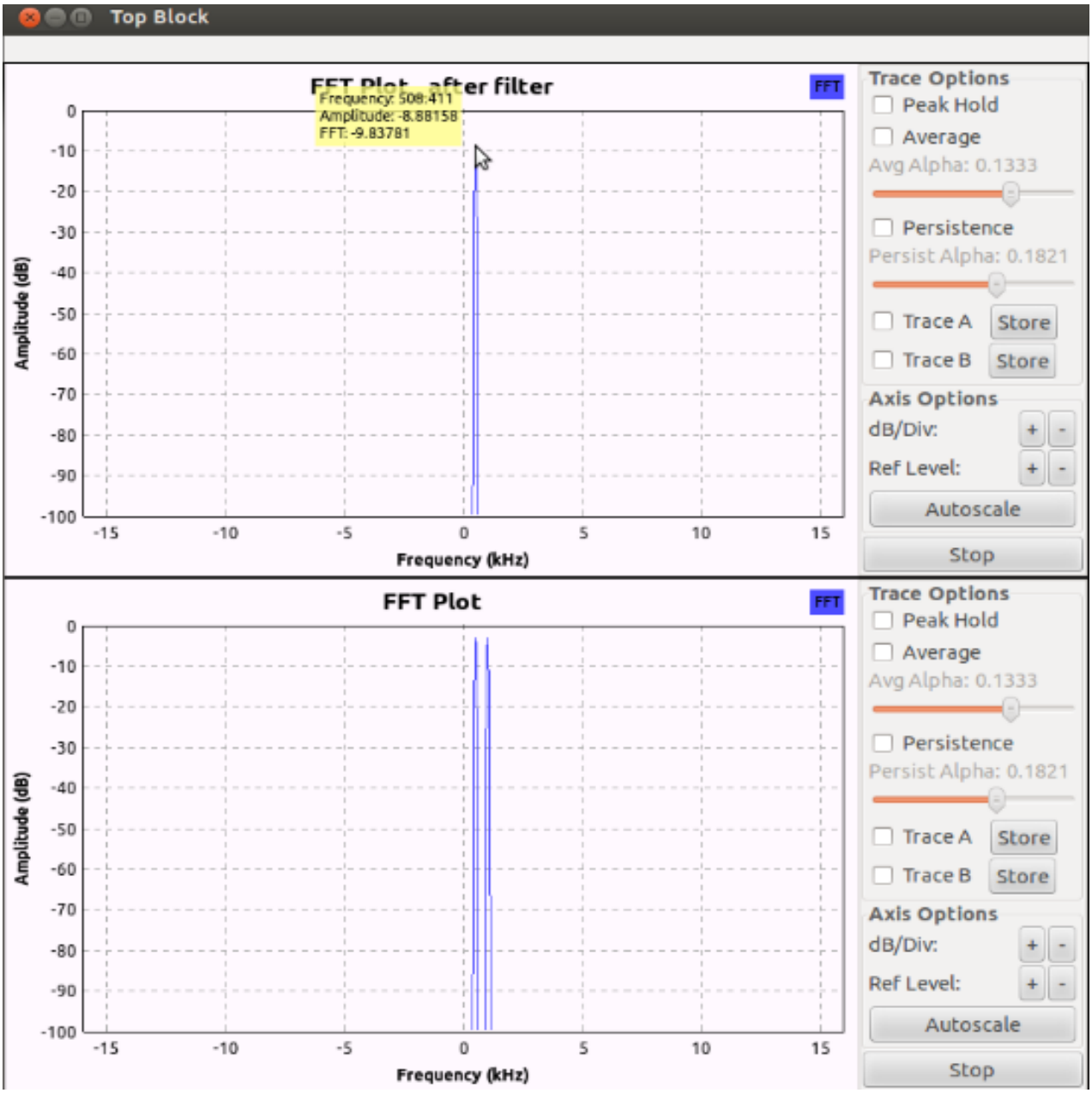

Fig 18: FFT Plot Of A BPF

Now consider changing the decimation factor to 2. A decimation factor of 'N' reduces the sample rate by 'N' which means the signal coming into the filter at 32000

samples /second is changed to 16000 samples per second that changes the highest frequency to 8000 HZ.

Note: Make sure to set the sample rate = sample rate/2 in WX GUI plot for correct results.

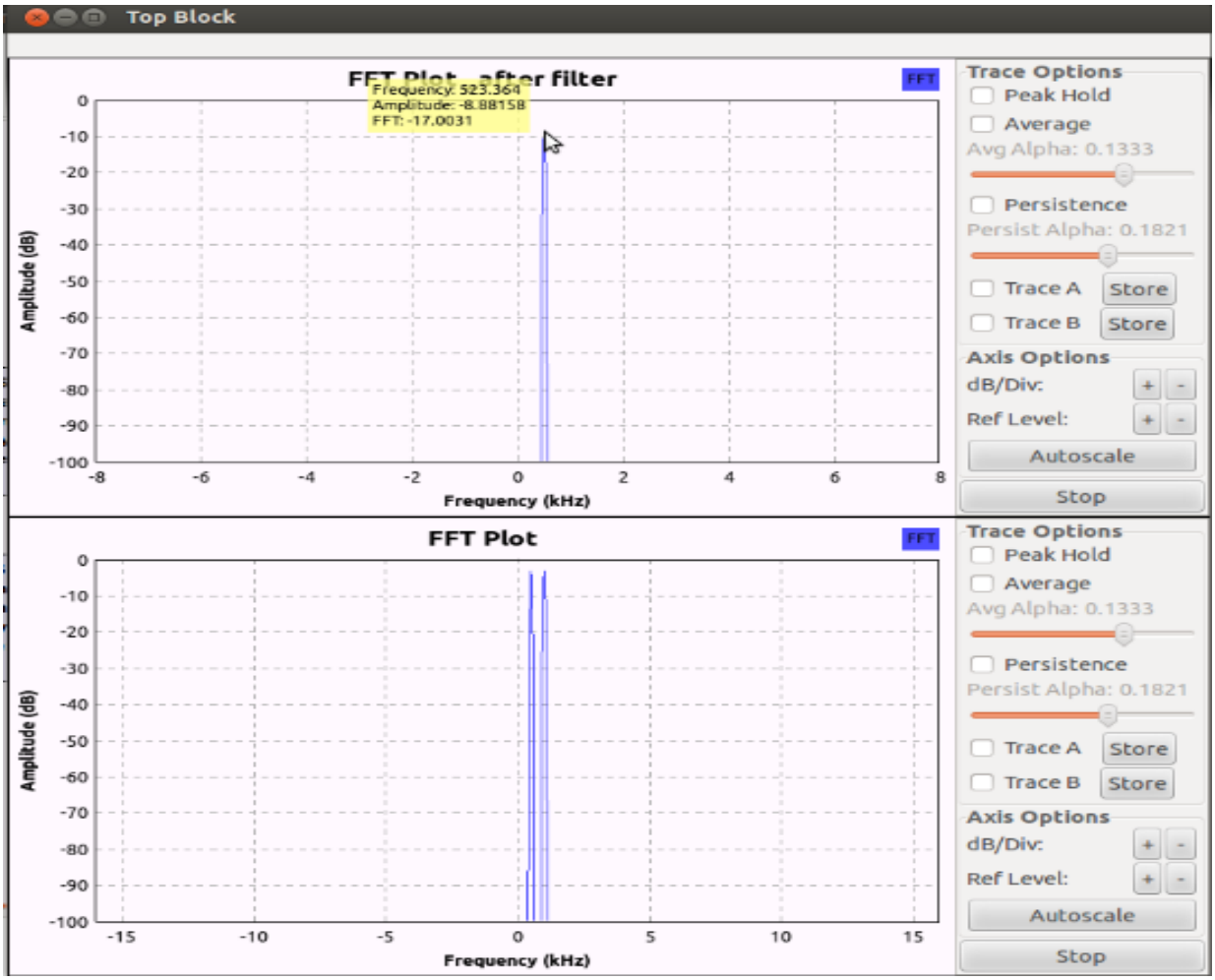

Fig 19: FFT Plot Of A BPF With Decimation Factor Of 2

2.10 Cooperative Relaying

Cooperative Relaying enables efficient utilization of communication resources, by allowing nodes or terminals in a communication network to collaborate with each

other in information transmission thereby increasing the quality of service. If the transmitter is far from the receiver, a relay definitely acts as a best choice to improve the signal quality. Of course, both the Direct Link and the path through the relay have their own independent channel qualities.

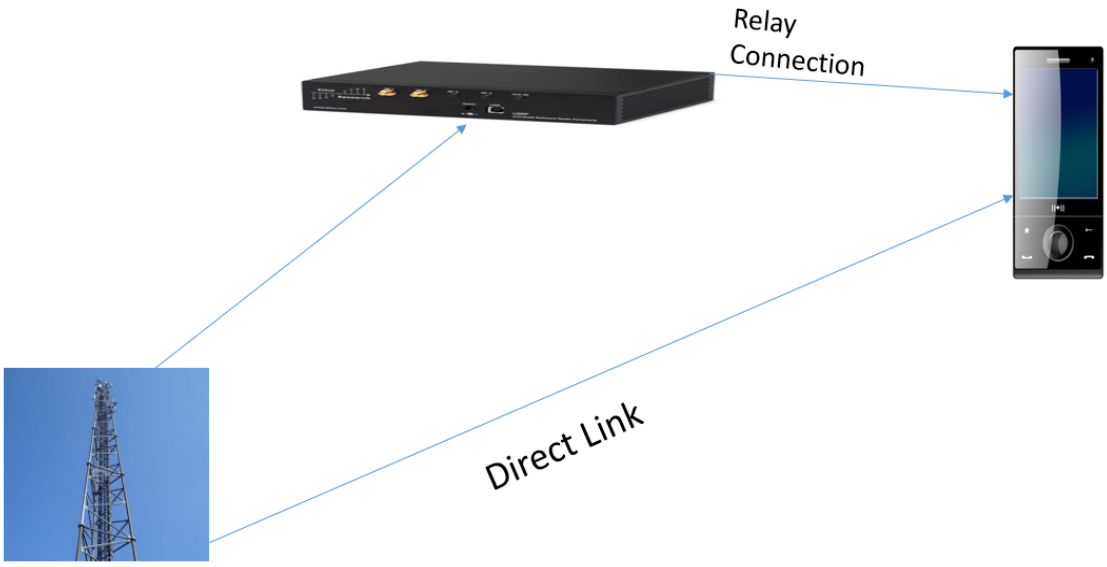

Fig 20: Cooperative Relaying

We implement this method of communication using three Radios (USRP1's), which act as a Transmitter, Receiver and a Relay. Also, the GNU Radio software is used for signal processing.

Relay operation has been implemented in two ways:

1. Amplify and Forward: Amplifies the signal first and then forwards it to the destination.
2. Decode and Forward: Decodes the signal and retransmits it.

CHAPTER 3

DESIGN

This chapter discusses the design that was used for this thesis work. It provides all of the details of the design so that further work can be conducted afterwards.

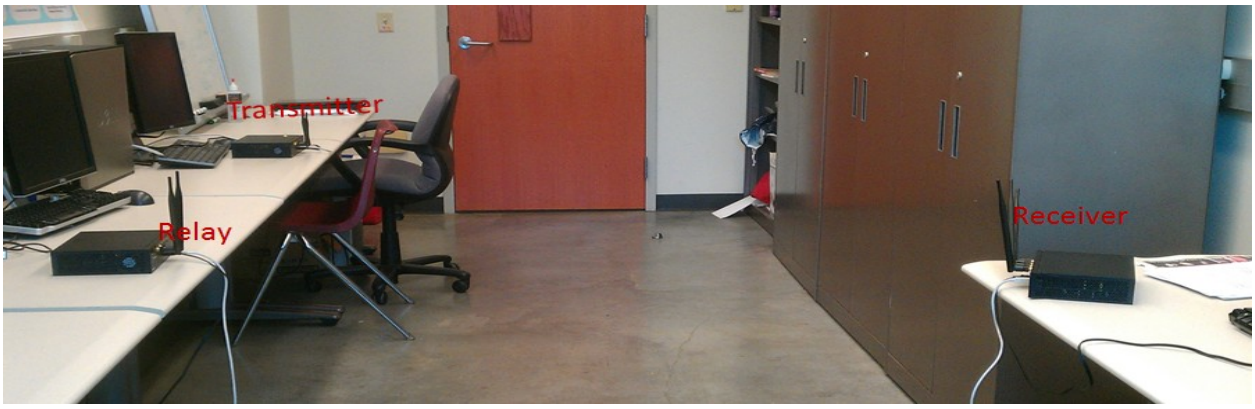

Fig 21: Experimental Setup In Lab

3.1 A Transceiver

We started our research by first creating the ability to observe the characteristics of a Transceiver using USRP 1 and GNU Radio.

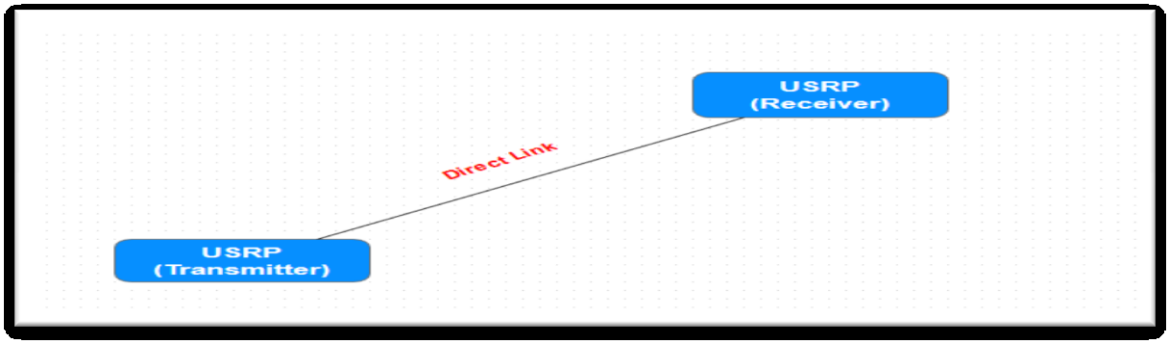

Fig 22: Transceiver Implementation

Our goal is to find the characteristics of a received signal in terms of packet error rate, character error rate and bit error rate.

In order to implement this, we need a transmitter and a receiver. The functionalities of a transmitter and a receiver are developed on the GRC files and used for this experiment.

3.1.1 A Transmitter

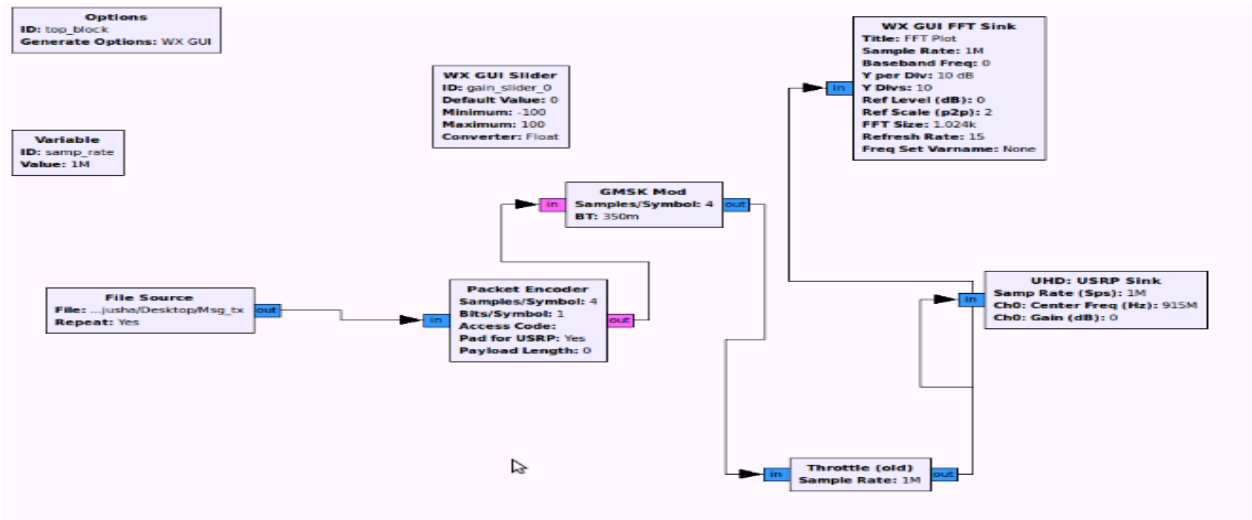

Fig 23: GRC Flow Graph Of A Transmitter

Now we describe some special features that were implemented in our research for the transmitter.

- File Source: The text message which we want to transmit is written in a file and is read from the file in this block to act as a source to this flow graph. The 'Repeat' option is set to enable in order to transmit the text continuously. A continuous stream of Character ‘h’ is sent; which looks like ‘hhhhhhhhhhhhhhhhhh’. We used this

known sequence of repeating characters to check for Character error rates at the receiver. [16]

- Packet Encoder: The text data we have is in the digital format. In order for the signal to be transmitted over the air the signal has to be made into packets. The Packet encoder should always be followed by blocks like GMSK, DBPSK and QAM. The packet encoder lets you choose number of samples/symbol and bits/symbol. [17]
- GMSK Modulator: Both the packet encoder and GMSK work synchronously. GMSK takes the 0's and 1's transforms them into -1 and +1's and runs the data through frequency modulator. The output of GMSK is always a phase modulated signal and the inputs are symbols. The input is a byte stream (unsigned char) and the output is the complex modulated signal at baseband. [18]

  A GMSK block is composed of three blocks:

  1. NRZ line coding: A stream of bytes is converted into +/- 1 symbols.

  2. Gaussian Filter: It is a finite impulse response filter (FIR) which multiplies the product of two numbers and adds that number to the accumulator. This procedure is called MAC (Multiply-Accumulate). MAC is used in convolution if filter taps with a stream of signals. Gaussian filter is used for filter shaping.

  3. Frequency Modulator: It accumulates the phase changes of incoming sample and outputs of I/Q samples. This block provides the modulation.

The packet passes through these blocks within a GMSK modulator.

Fig 24: GMSK Modulator Block Diagram

We use GMSK over other modulation techniques due to:

- It's simplicity to implement in GNU Radio
- High spectral efficiency
- The tradeoff between demodulation accuracy and spectral efficiency can be set with the help of a parameter called Bandwidth Time (BT) product, which describes how much smoothing, can occur. A BT of 0.3 works fine for our experiment.
- GMSK modulation again arises from the fact that none of the information is carried as amplitude variations. This means that is immune to amplitude variations and therefore more resilient to noise, than some other forms of modulation, because most noise is mainly amplitude based.

- Throttle: A Throttle block is used to avoid the computer from freezing.
- USRP Sink: The analog signal generated by the blocks above is sent over the USRP Sink for the transmission over air on the prescribed center frequency. We chose to transmit the signal at 915MHZ since the daughter boards RFX 900 works best in the range of 902-928 MHZ ISM band.

Considerable changes have been made in Python code, in order to view the number of packets being transmitted and also transmit a packet number with the

payload. Cooperative relaying combines or choose the best of the multiple signals it receives. We need a packet number so that we can compare the same packets that are received over these multiple path. These changes are described in the following sections.

Figure 25 illustrates the signal at the Transmitter

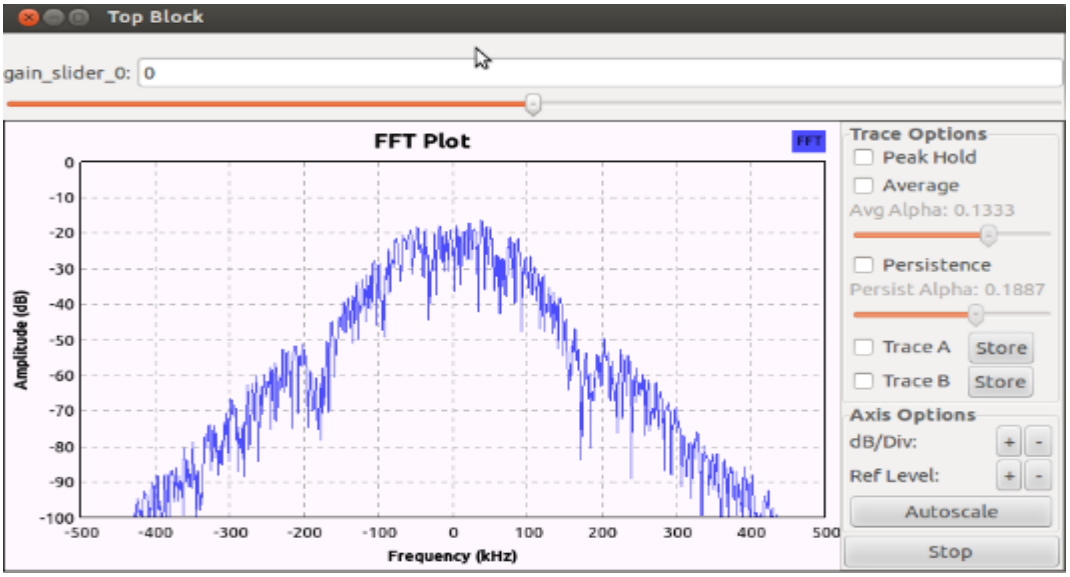

Fig 25: Signal At The Transmitter

Data sent from the transmitter is a stream of ‘hhhhhhhhhhhhhhhhhhhh’.

3.1.2 Receiver

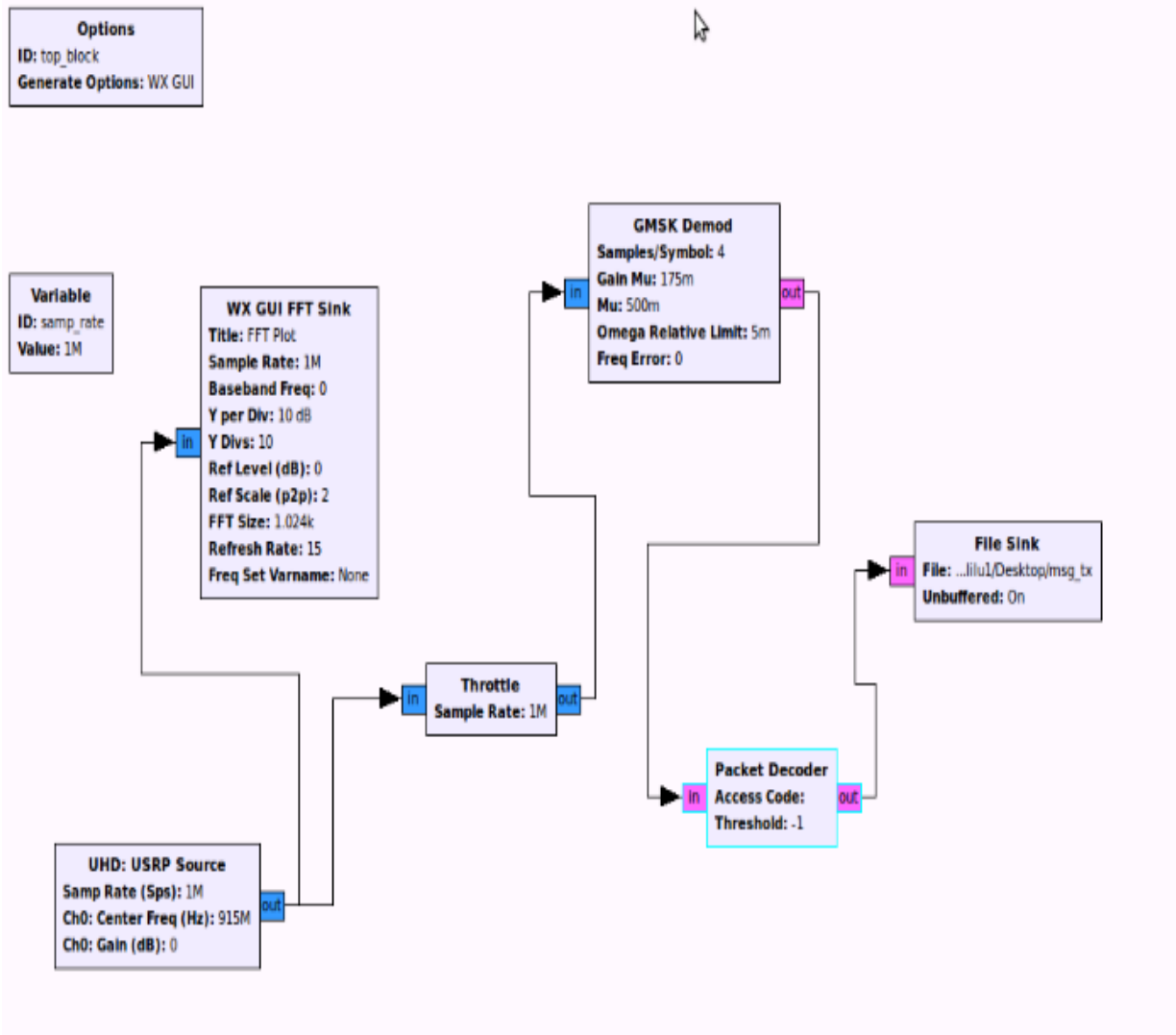

Fig 26: GRC Flow Graph Of A Receiver

Now we describe some special features that were implemented in our research for the transmitter.

- USRP Source: The transmitted signal is received with this block. The center frequency of transmitter and the receiver should always be same. The signal received is analog and further blocks are used to convert this signal into the readable form.

- WX GUI FFT: It is used to view the received signal.
- GMSK Demodulator: It is reverse of a GMSK modulation. The input is the complex modulated signal at baseband and the output is a stream of bits packed 1 bit per byte (the LSB).

GMSK has three blocks:

1. Quadrature demodulator: it reversed the operation of frequency modulation. It processed the incoming packets and estimates their phase information.

2. Clock Recovery: Is used to synchronize the symbols using Mueller and Muller (M&M) discrete error tracking synchronizer. M&M tracks the symbol clock, re samples as needed and gives an output in soft symbols.

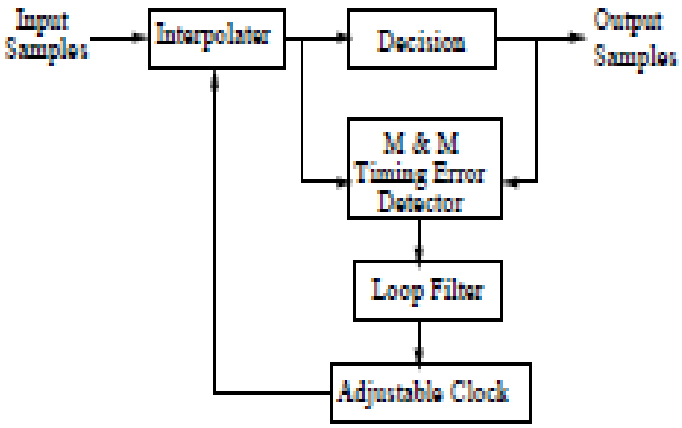

Fig 27: M&M Discrete Error Tracking Synchronizer

3. Slicer: For symbol decision

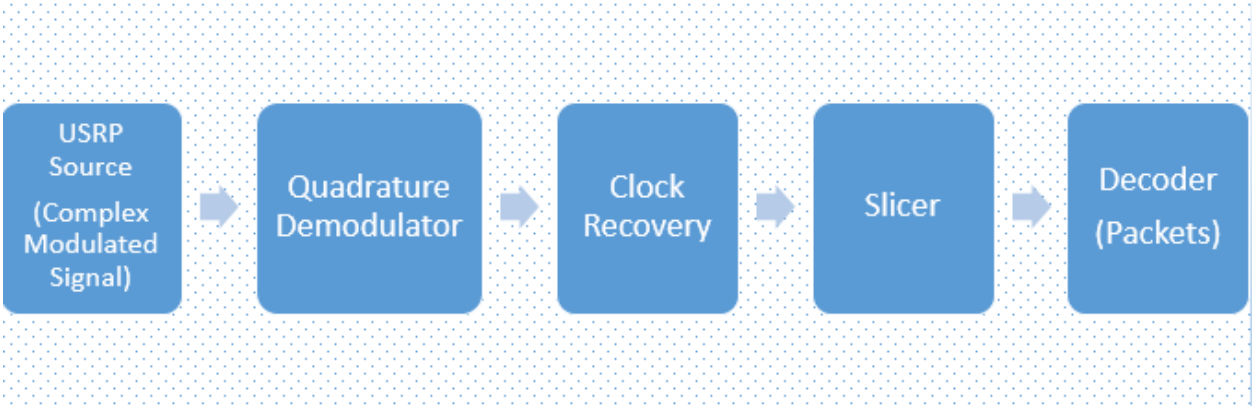

Fig 28: GMSK Demodulator Block Diagram

- Packet Decoder: All the packets received are broken down to extract the data out of the entire packet.
- File Sink: The extracted payload/data is sent to a file. The output file shows only the text being sent but gives no information about the number of packets being dropped during transmission. So, few changes have also been made in the receiver which are discussed in the following sections.

Here is the plot that is seen of the Received signal:

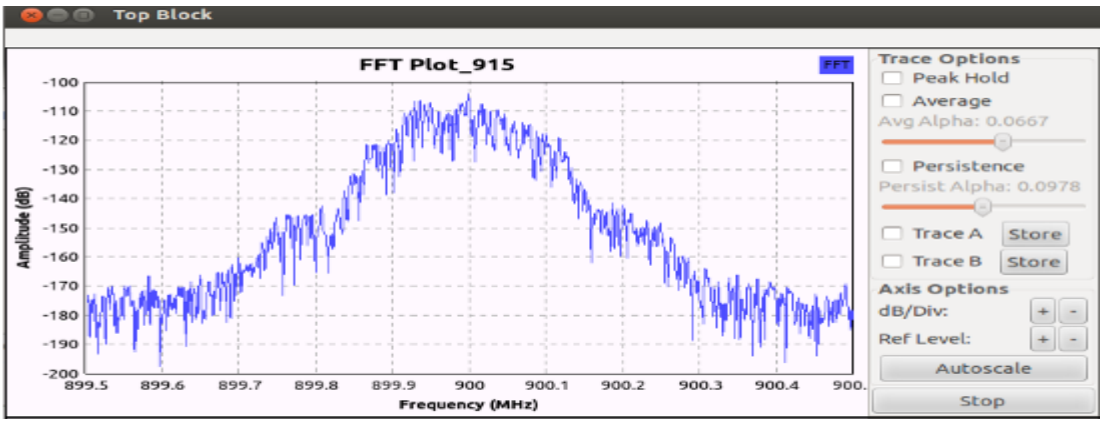

Fig 29: FFT Plot At Receiver

Figure 30 shows the terminal output that shows the Received Data for each packet and the statistics.

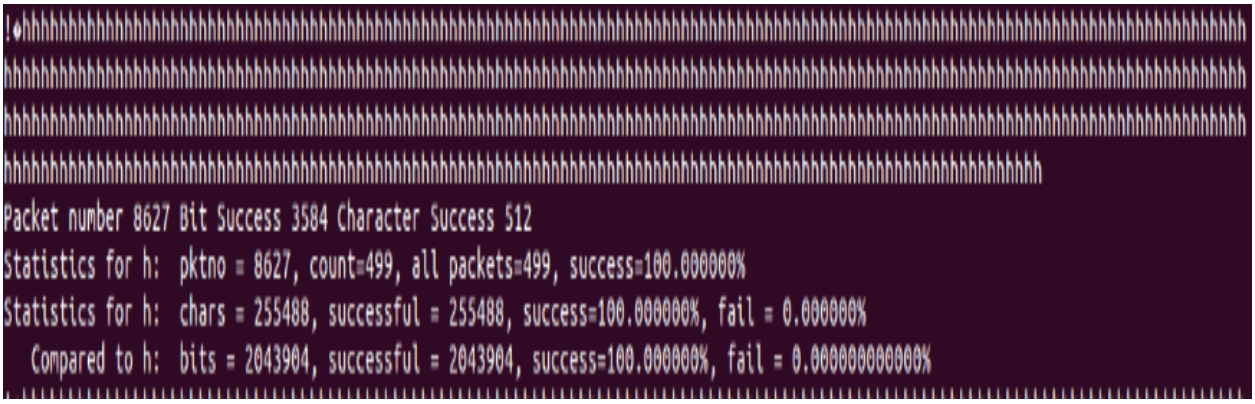

Fig 30: Terminal Output Of The Data Received

3.2 Implementation Of Cooperative Relays

We used three USRP1s for our experiment. One Transmitter, One Receiver and other acts a Relay. See again Figure 21. All are placed at variable distances.

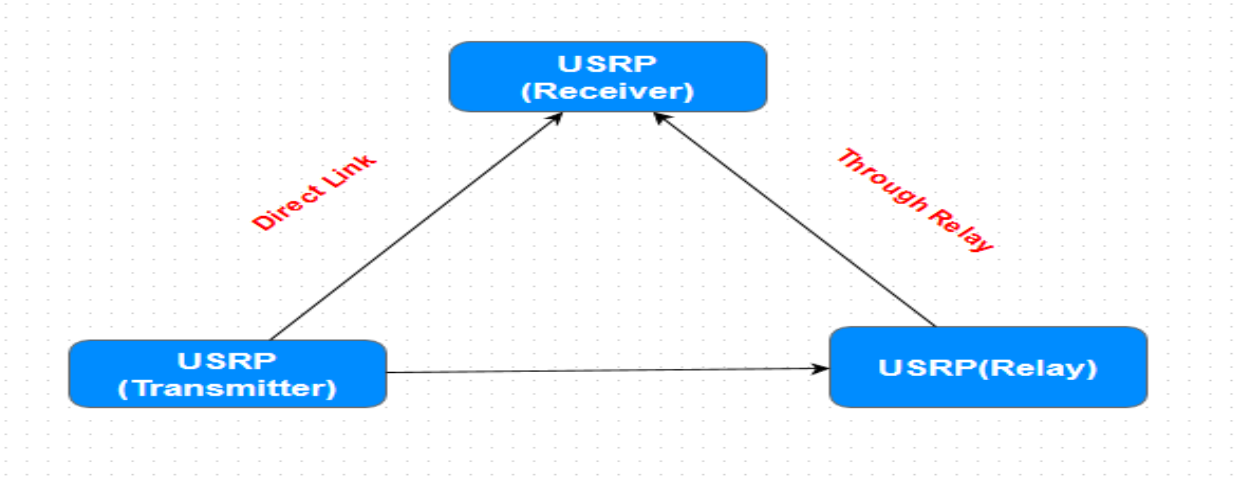

Figure 31: Cooperative Relaying

The Transmitter is the same as the one used for Basic Transmitter (explained in detail above).

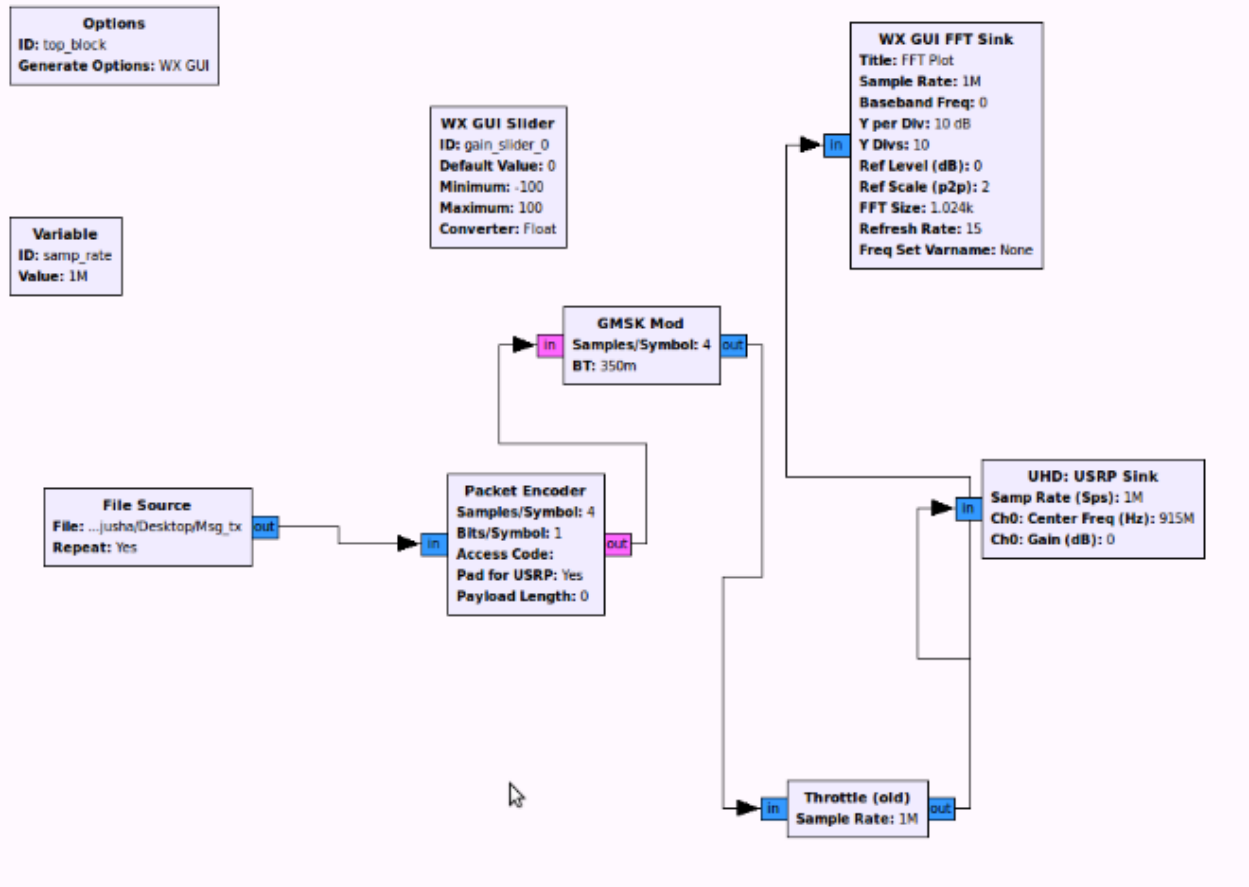

Fig 32: GRC File For Transmitter

3.2.1 A Relay

3.2.1.1 Amplify And Forward

This is a simple method that is useful to understand the concept of Cooperative communication. The receiver receives a noisy version of two signals coming directly from the transmitter and other from the relay. As the name implies, the signal along with

the noise from the channel is amplified and retransmitted at the relay. The base station then takes a decision of choosing the best or combining packets from these paths. [19]

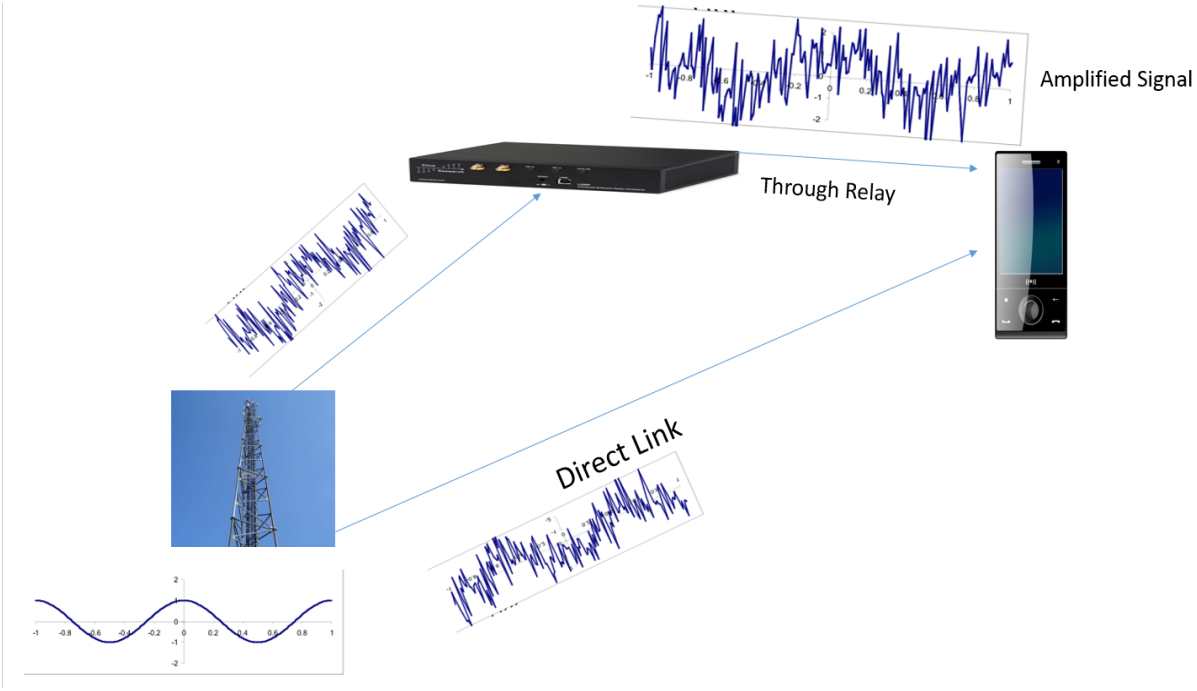

Fig 33: Amplify And Forward Implementation

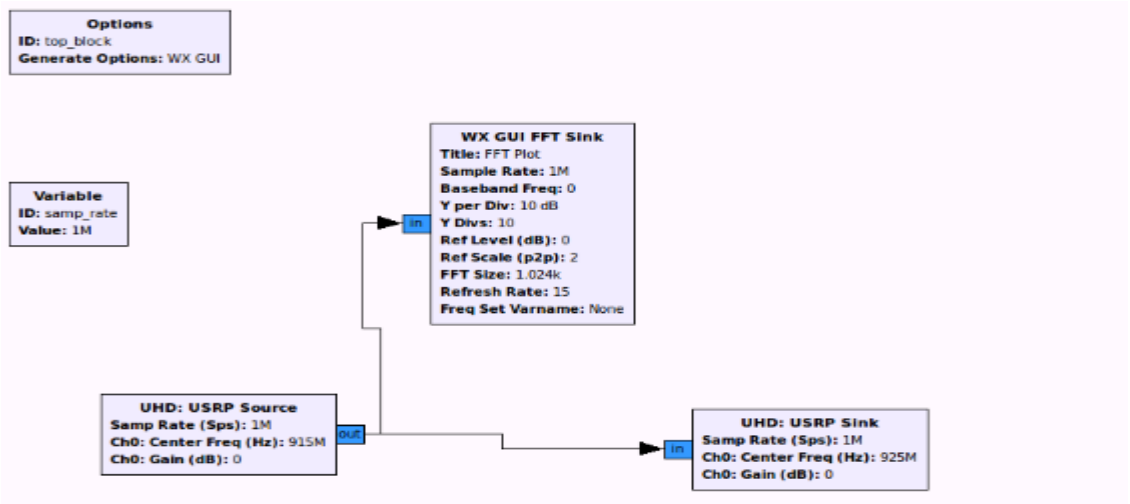

Fig 34: GRC Graph Implementation Of A&F

All that is needed for GRC and the USRP is to receive the signal and then send the signal. Both the source and sink have amplifier gains; we study their impact on performance.

3.2.1.2 Decode and Forward

In this method, the signal from the Transmitter is decoded and then re-encoded in an attempt to achieve better throughput. [19] We show in our results when this is and is not the case.

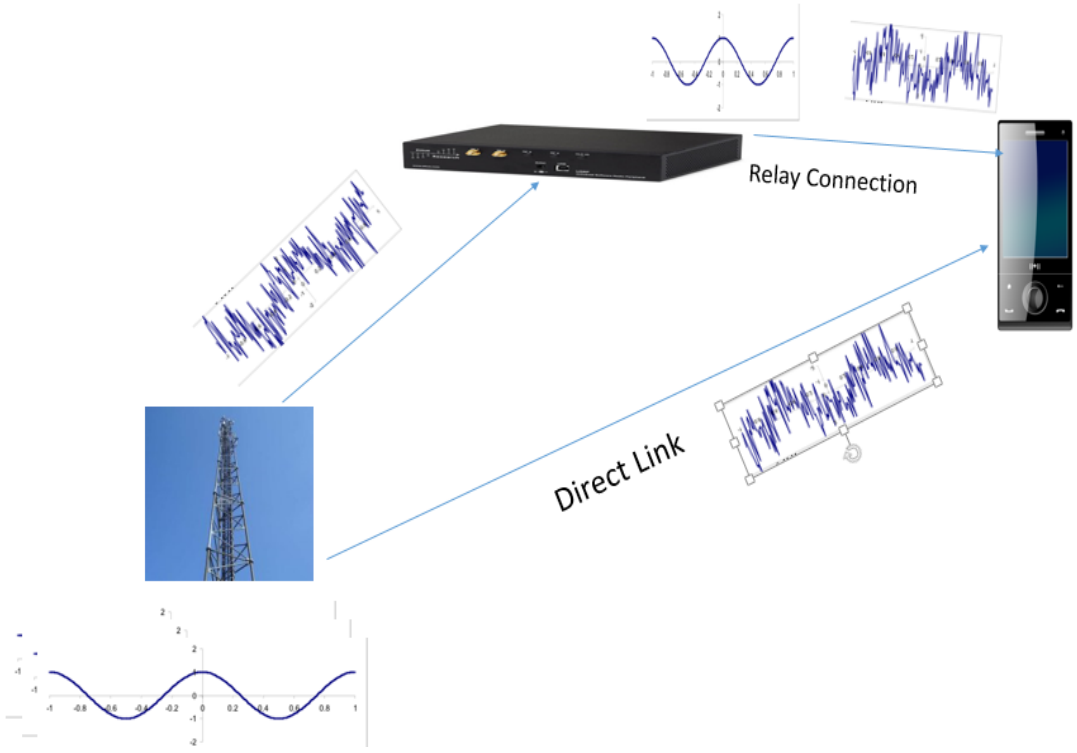

Fig 35: Decode And Forward Implementation

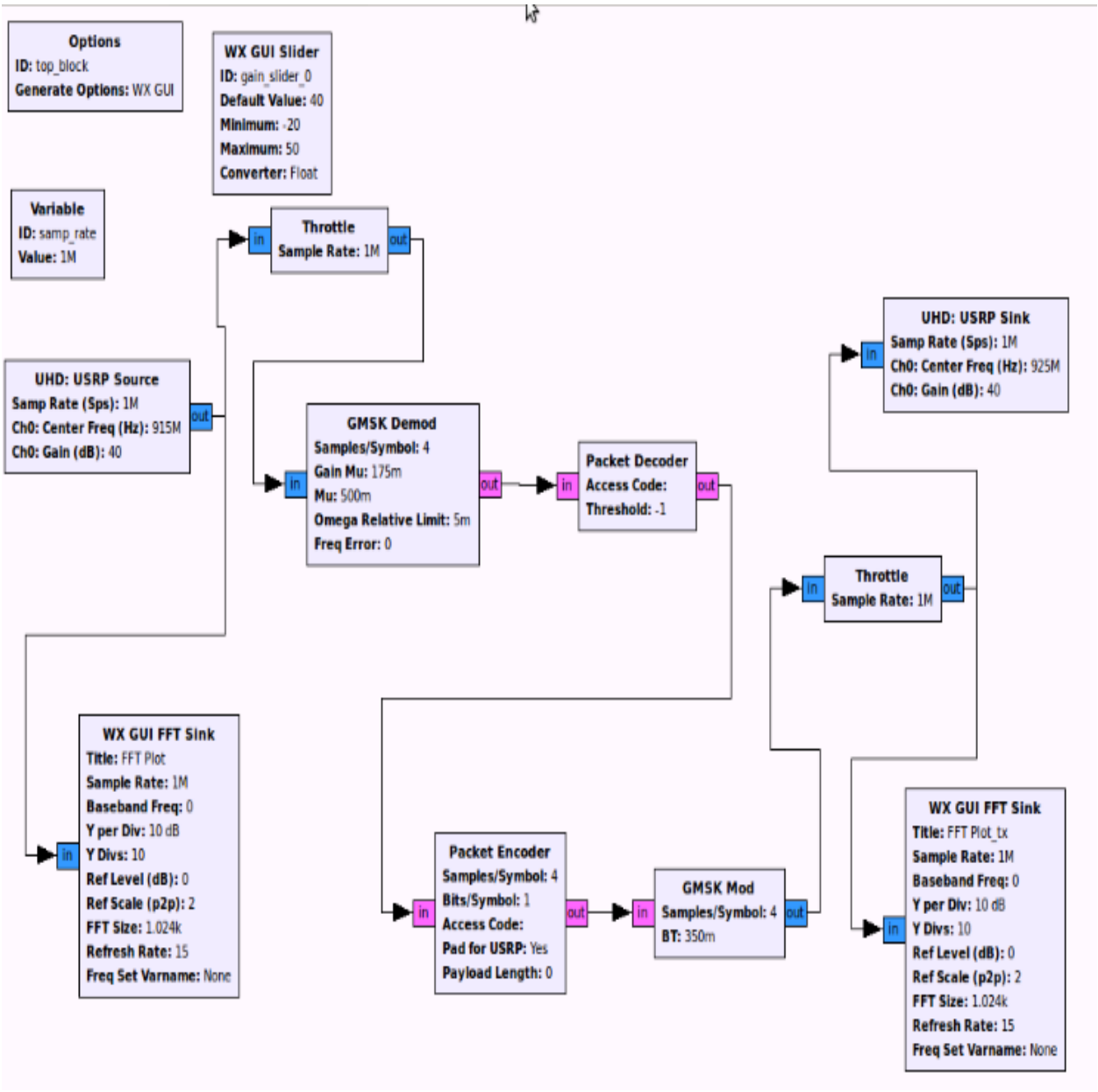

Fig 36: GRC Flow Graph of D&F Relay

3.3 Final Destination/Receiver

Figure 37 shows the GRC implementation of the receiver.

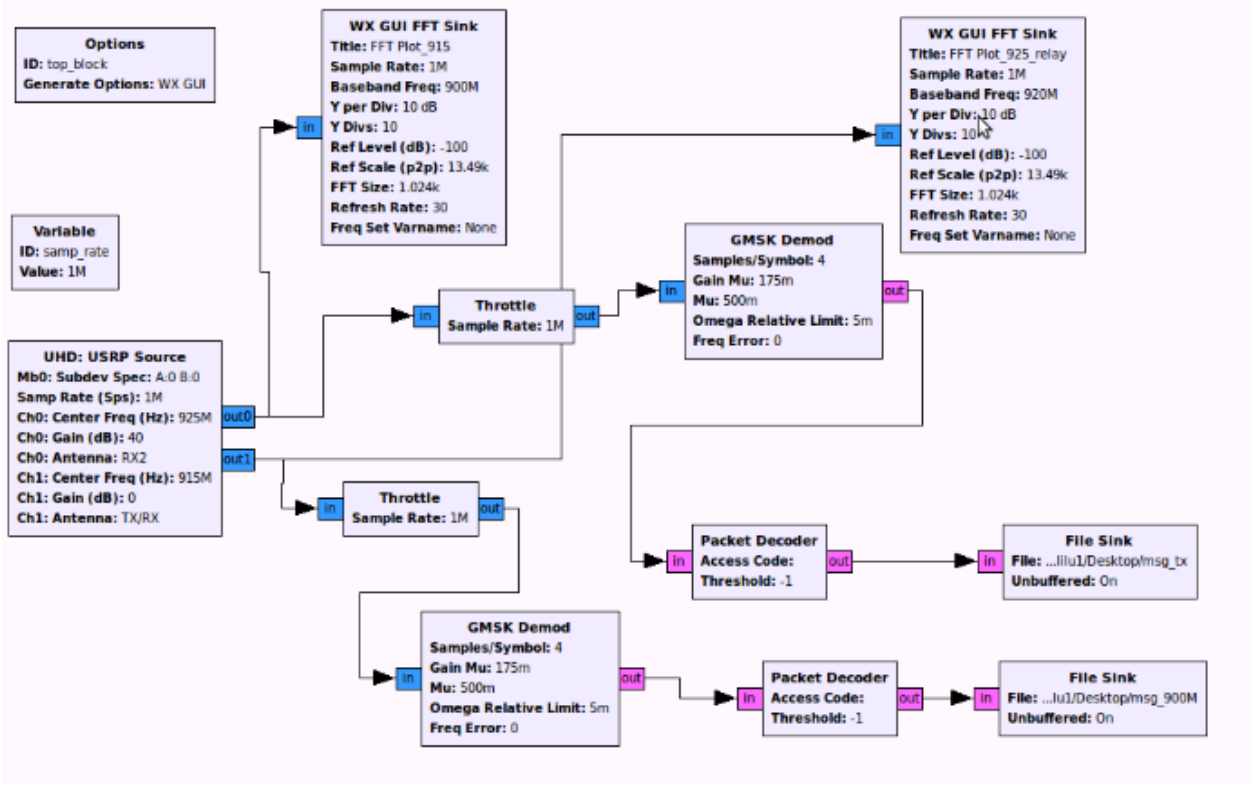

Fig 37: GRC Flow Graph Of Final Destination

The plot is similar to the receiver's plot in a basic Transceiver [20] [21]. But the USRP source is set to receive signals on two channels (one from Transmitter and other from relay) at different frequencies. These come from separate daughterboards in the receiver. The demodulation and decoder processes are done individually for both the signals and stored to files.

3.4 Detailed Explanation Of The Cooperative Relaying Process

We made several changes to the Python code that operates below the GRC implementations. This is described in detail here.

Prior to explaining our Transmission and Reception procedure of Cooperative Relays, we need to know how the data is packed into packets and transmitted over air.

3.4.1 Packet Structure

After application of all the techniques like amplify, modulation etc., the data is passed through a GNU Radio function called 'make_packet' where a complete packet is formed. [22]

For this function, a packet consists of the following, as is also seen in Figure 38.

1. Preamble – 2 Bytes in length and used for maintaining synchronization (in frequency and time) in between the Transmitter and the Receiver.
2. Access Code – 8 Bytes in length and positioned at the start of the packet. Both Preamble and Access Code correlate with the Receiver for better synchronization.
3. Header – 4 Bytes in length and has the information about length of the payload. When a stream of sampled data comes to the receiver, all the data are discarded until the beginning of the packet is signaled either by the stream tag or a trigger signal from the coming inputs. Once the start of the packet is detected, the preamble and header are copied to the output by the de-multiplexer. The header is then demodulated with any of the GRC blocks.
4. CRC-32 – 4 Bytes in length and is used for error detection for the payload.
5. Whitener Offset – Ranges from 0 to 15 Bytes in length. The payload is 'whitened' which means an XOR operation is done with a PN code generated by the Linear

feedback shift register (LFSR) in order to avoid phase error during the transmission. This offset determines where to begin the whitening.

6. Payload – The length of the payload ranges from 1 to 4092 Bytes.
7. End Byte – 1 Byte in length which tells the receiver that the transmission is complete.

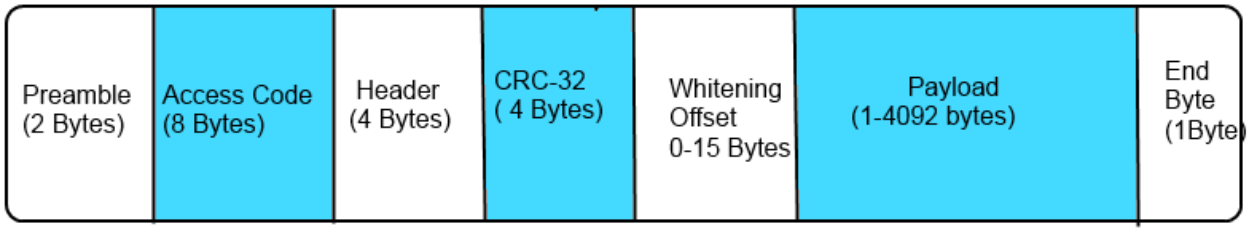

Fig 38: Packet Structure

The main functions in Python of a Transmitter, Relay and a Receiver are explained below.

3.4.2 At Transmitter

After the text from the file source is forwarded to the Packet encoder, the complete packet structure is formed.

Before encoding the packets, creation of Header and the Packet is necessary. This is done within packet_utils.py which is located in /usr/local/lib/python2.7/dist-packages/gnuradio/digital. The function used is make_header where the length of the payload and whitener offset values are packed. The packet is framed within the function make_packet. This function packs the packet with the payload (payload+packet number+packet number), sampling rate, bits per symbol, preamble etc.

```
converted_pkt_no=struct.pack('!H', pkt_no & 0xffff)
payload2=converted_pkt_no+converted_pkt_no+payload
```

We add the packet number to the payload and transmit it over the air so we can to calculate the errors that occurred due to interferences. This packet number is added twice to the payload in order to increase the efficiency of our results. The packet numbers which match with each other at the receiver will only be processed to calculate the statistics of the system. Otherwise, errors have occurred in the packet numbers and This function of checking the packet number match is done in relay_process.py file programmed exclusively for our project.

Packet transmission is done through the function send_packet in the Packet.py file which is located in /usr/local/lib/python2.7/dist-packages/grc_gnuradio/blks2. The send_packet function is a part of packet_encoder where the packet data is pushed onto the message queue for transmission. After the complete packet structure is formed, it is sent to our next block, the GMSK modulator. This function is located in /usr/local/lib/python2.7/dist-packages/gnuradio/digital, where the entire procedure of GMSK takes place. The working of the GMSK modulator was explained in the above sections.

Finally the message queue is passed over to the USRP Sink where the digital signal is processed to an analog signal with the help of D/A converters and transmitted over air.

The signal transmitted can be viewed (on the terminal) by printing the payload as seen below.

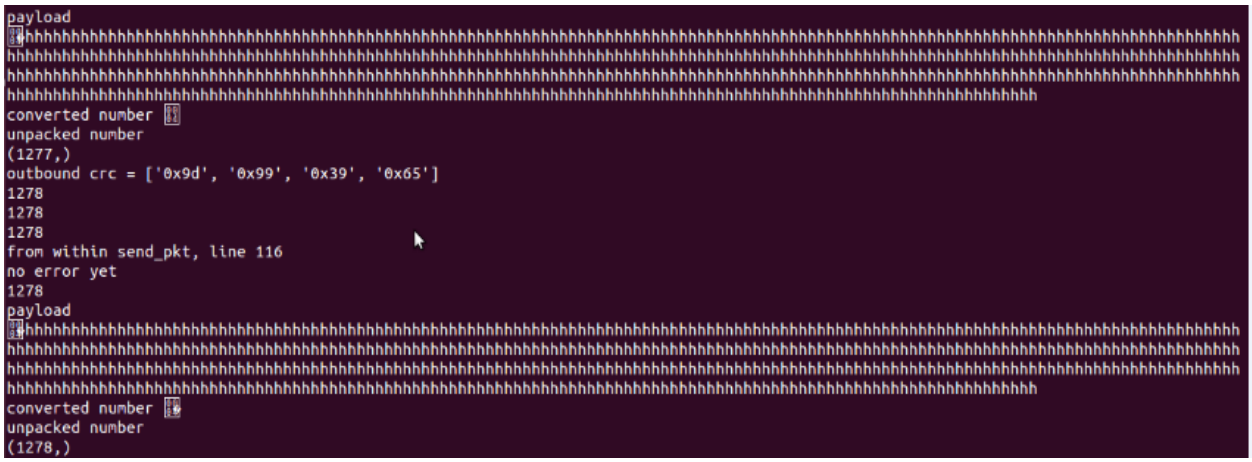

Fig 39: Terminal Output At The Transmitter

The first few characters of the stream 'hhhhhhhhhhh' are the hexadecimal representation of the packet numbers. The original packet numbers can also be seen above.

3.5 At The Relay

The USRP source is used to receive the signal. The center frequency is set to a frequency on which the signal is being transmitted. Gains are kept variable to see the performance of the signal. These results are analyzed in the next chapter.

3.5.1 Amplify And Forward

This just forwards the received signal onto another frequency. No additional Python code changes are necessary in order to implement this scenario. The data along with the packet numbers will automatically be forwarded to the receiver.

Output At A&F Relay

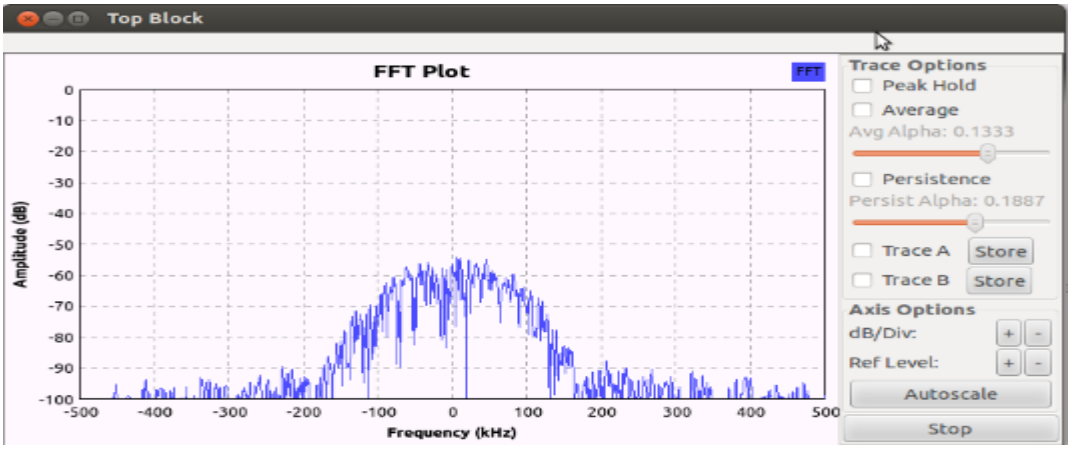

Fig 40: Output For A&F Implementation

3.5.2 Decode And Forward

The signal received has both entities of the payload and the two packet numbers. While the signal is decoded, the payload and the packet numbers are separated. So, to retransmit the payload accompanied with the packet number, we had to add the received packet numbers with the payload again. To meet this requirement, a few changes have been made in packet.py and packet_utils.py.

The header is first separated from the packet in the packet_decoder_thread function located in packet.py. Packet number is extracted out from the payload in packet.py as shown below.

```
(pkt_no,) = struct.unpack('!H', payload[0:2])
```

The packet numbers are re added to the payload in make_packet function located in packet_utils.py. Also, the payload is converted to upper case, just to create the difference in between the payload coming from relay and the payload coming from the

Transmitter at the Final Destination. This allows the receiver to easily distinguish the direct and relayed packets.

payload =converted_pkt_no +converted_pkt_no +payload.upper()

Output At D&F Relay

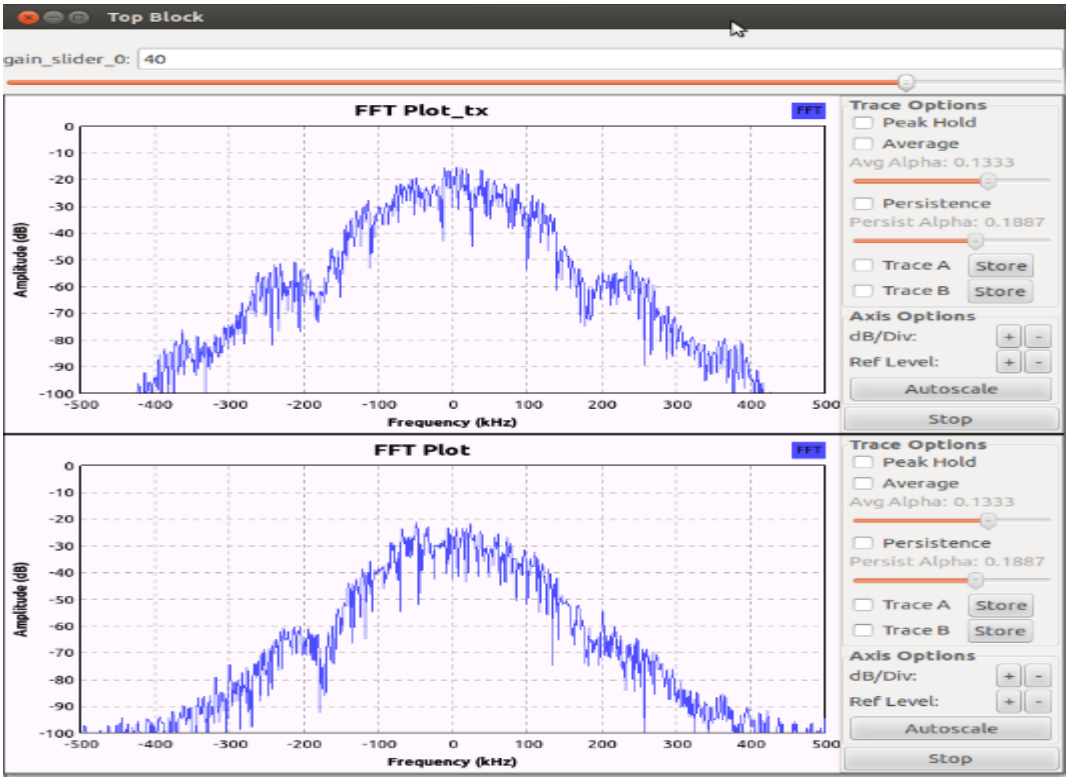

Fig 41: FFT Plot At The Relay

Output In The Terminal

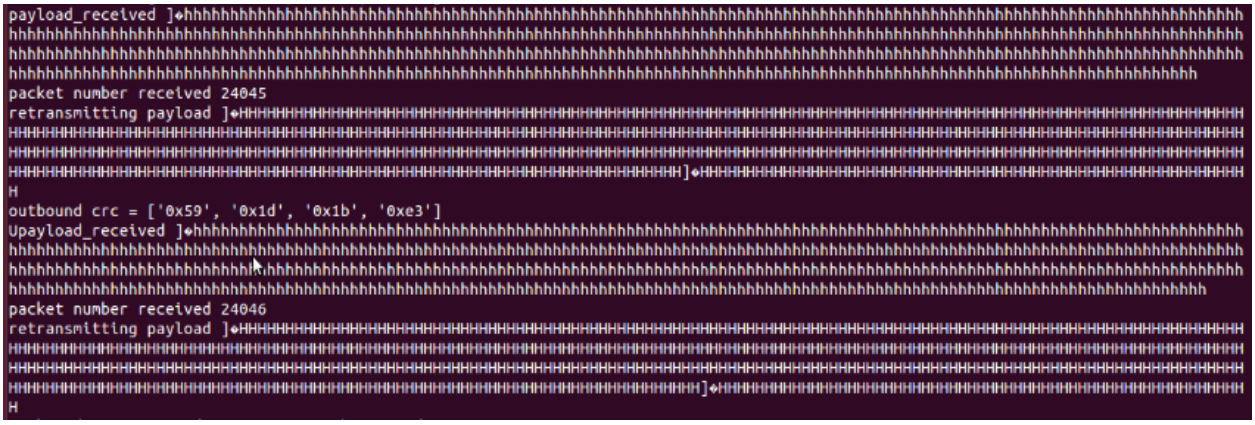

Fig 42: Terminal Output At Relay

The stream of "hhhhhhhhhhhhhhhhhhhhhh" is the received payload. The stream of 'HHHHHHHHHHHHHHHHHHHHHH' is the retransmitted payload. The received packet numbers are also observed above.

3.5.3 At The Receiver

The USRP Source receives the input from two channels. One, directly from the transmitter called Direct Link and other from the Relay. These two analog signals are converted to digital (in frames/packets) in the mother board within USRP and forwarded to the GMSK demodulator for converting complex modulated signal to the packets. The gains are set variable in the Source to see the performance of both these signals. Some additional details are as follows.

- GMSK_Demod: The complex baseband signal is sent to the Demodulator to produce a stream of bits packed 1 bit per byte. The functionality of a GMSK Demodulator is explained in the above sections.

- Packet Decoder: Used to unpack and decode the information. Firstly, we need to unpack the packets before proceeding to decode function. This function is located in packet_utils .py in /usr/local/lib/python2.7/dist-packages/gnuradio/digital.

- Unmake Packet: This function is used to unmake the packet. This separates the payload from other parameters like whitener offset dewhitening etc. After all the conditions are verified, the data is now stored in "payload" variable.

- The payload is now forwarded to Packet.py located in /usr/local/lib/python2.7/dist-packages/grc_gnuradio/blks2

- Packet_Decoder_thread: This function decodes the data and calculates the Bit error Rates, Character Error Rates and Packet Error Rates coming from both the Direct link and through the Relay.

After the packet numbers and the number of successful bits are written to file, another Python file, relay_process.py, is run to calculate the Bit error rates of the entire system. This implements the equivalent of a selection diversity relay combining strategy. If both packets are received, the best packet is chosen. This is the one with the fewest bit errors, which can be determined since we know the payload should either be all 'h' or 'H' characters. If only one packet is received, that one is chosen even if it comes from the path (direct or relayed) that might in general be worse.

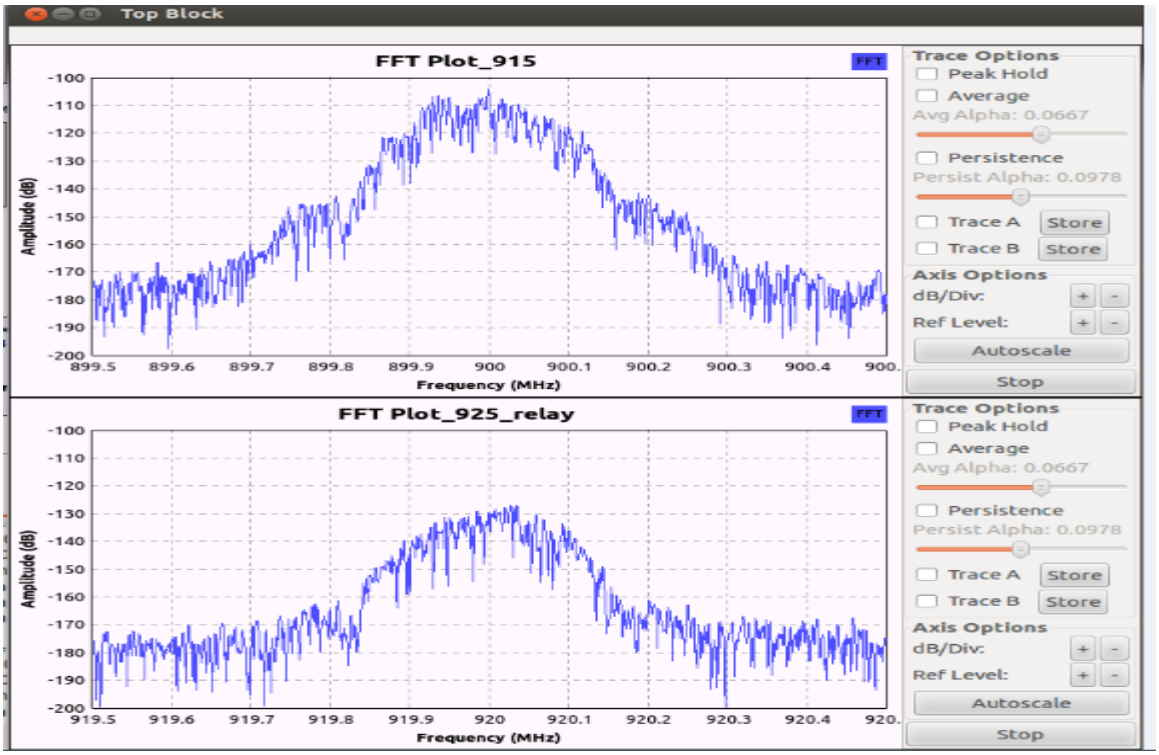

Fig 43: FFT Plot At The Receiver

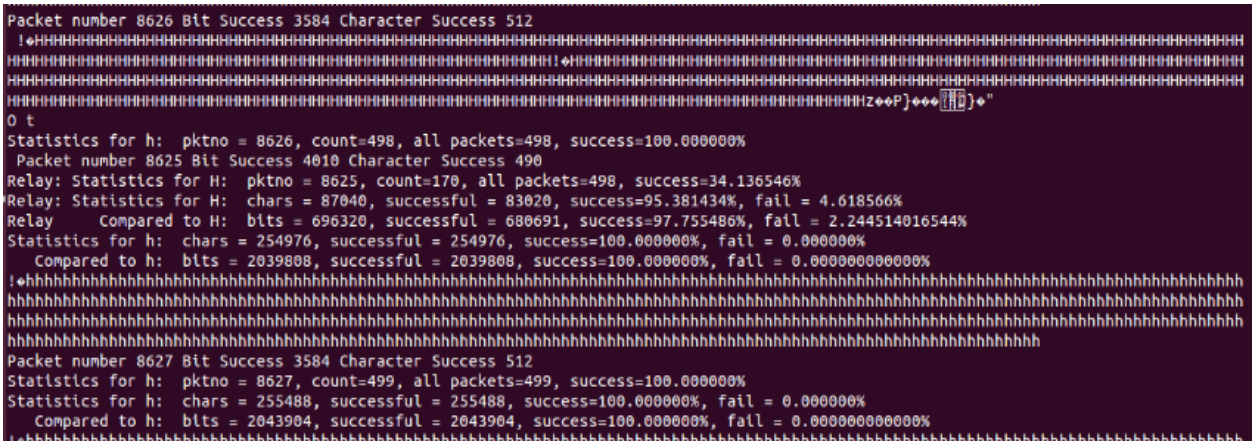

Fig 44: Terminal Output At The Receiver

The statistics shown above are picked from a batch of outputs produced at the Receiver when the USRPs are placed in some random positions. Now that the packet

numbers and the bit success rates are entered into the files Direct_path.txt and Relay.txt, the Overall BER of the system is calculated from relay_process.py.

CHAPTER 4

RESULTS AND ANALYSIS

Given below are statistics are obtained at random gains for the Relay receiver, Relay transmitter, and Source transmitter (i.e., Direct Link). BER in Direct Link and Relay Link are calculated individually from packet.py .The Overall BER is calculated from relay_process.py were the best packets are chosen. As observed from the table, the Overall BER is substantially improved in most of the conditions.

Gain at Relay (dB)	Gain at Direct link (dB)	Gain at Relay (dB)	Packet Success Rate (Direct Link)	BER (Direct Link)	CHAR error rate (Direct Link)	Packet Success Rate (Relay)	BER (Relay)	CHAR error rate (Relay)	Overall BER	D&F or A&F
20	8	20	0.937	0.002	0.021	1.0	0.0023	0.0115	0.00193	D&F
20	12	20	0.9633	0.0013	0.0123	0.9881	0.0007	0.00076	0.00039	A&F
20	20	5	0.99	0	0.0039	0.999	0.0034	0.012	0.06	D&F
20	20	10	1.0	0	0.0038	0.884	0.0041	0.029	0	A&F
10	13	20	0.847	0.118	0.069	1	0.0040	0.0115	0.005	D&F
10	10	20	0.807	0.01	0.0688	1	0	0.00389	0	A&F
10	20	10	1	0.003	0.0115	0.994	0.00305	0.0038	0.00006	D&F
10	20	2	1	0	0.00389	0.813	0.0084	0.00595	0.00	A&F
0	11	20	0.946	0.0002	.0180	1	0.0038	0.0115	0.00215	D&F
0	13	20	0.984	0.0005	.0074	1	0	0.00389	0	A&F

The following plots show a more systematic investigation. The plots obtained by varying the gains at the Relay and Receiver (both Relay and the Direct Link)

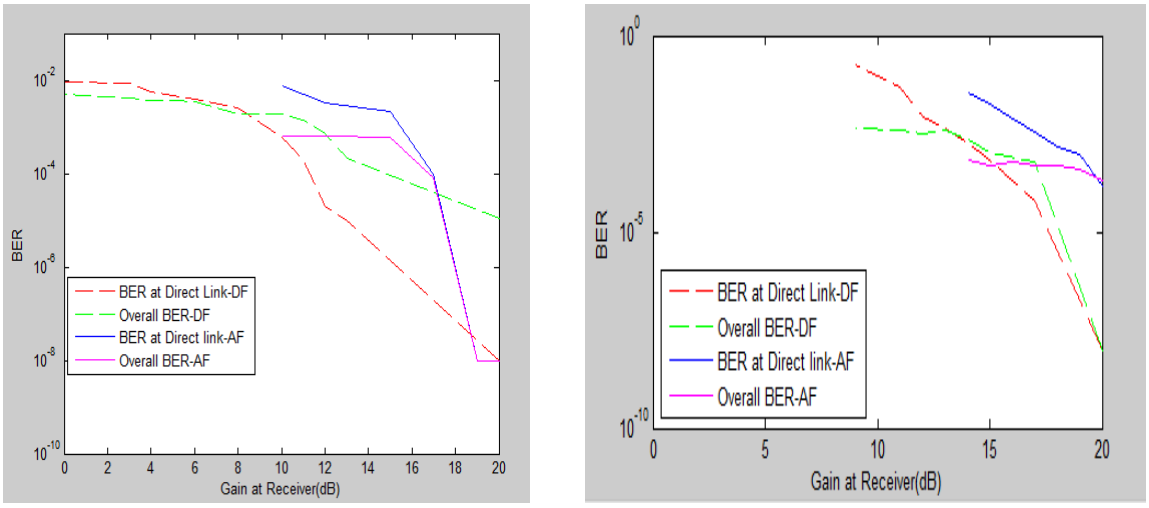

Fig 45: AF&DF (LOS On Left And Obstruction Environment On Right), 20 dB Gain At Relay, Variable Gain At Direct Link-RX

The above plots are obtained by applying 20 dB at the Relay and at the Receiver for the Relay link. Gains at the receiver for the Direct Link are varied, to observe the performance of the system. Plots show how the BER performance of the Direct Link and Overall perform. The curves for the relay are not shown since they do not change and would clutter the figure. In a LOS (Line of Sight) environment (shown in the plot on the left), starting from the right at 20 dB moving left the BER in A&F works similar to each other (for Direct Link and Overall) until close to 18 dB and then as the gain decreases, the BER at Direct link increases faster. This means that above 18 dB the receiver would choose the Direct Link, but at less than 18 dB it would choose the Relay path. The relay continues to work better and all the packets from the Relay are processed at this time maintaining lower Overall BER. The same happens with D&F. Interestingly, D&F is sometimes better than A&F, sometimes not. The system performance in an obstruction

environment (seen on the right) is definitely worse in comparison with the LOS but provides good Overall system performance in its own conditions.

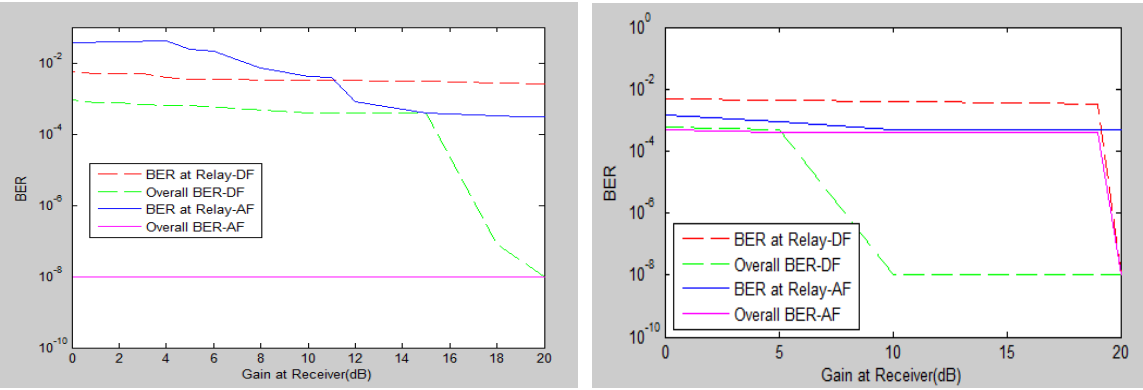

Fig 46: AF&DF (LOS On Left And Obstruction Environment On Right), 20 dB Gain At Relay, Variable Gain At Relay Link-RX

Now we fix the received gain at the receiver for the Direct Link and vary the gain at the receiver for the Relay link.. The above plot is obtained by maintaining 20dB at the Relay and 20 dB at the receiver for the Direct Link. In the LOS scenario, the Overall BER is always at its minimum value in A&F conditions. There is also drastically better performance for the Overall BER in the D&F method since the Direct Link is working on high gain. In the Obstruction environment, D&F performs better than the A&F method. A&F doesn't show much improvement between BER and Overall BER.

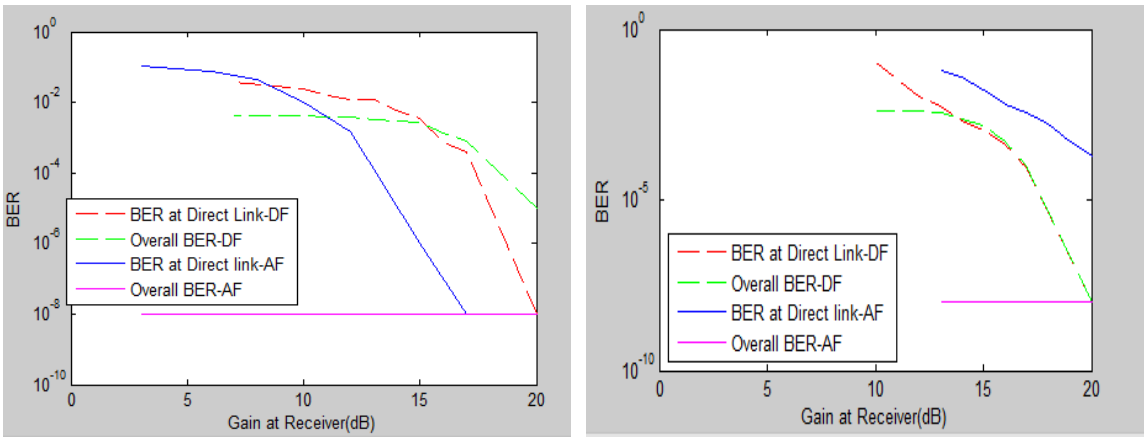

Fig 477: AF&DF (LOS On Left And Obstruction Environment On Right), 10 dB Gain At Relay, Variable Gain At Direct Link-RX

The above plot is obtained by maintaining 10dB at Relay, 20 dB at Relay Link at RX. Gains on the Direct Link are kept varying to observe the performance. In the LOS scenario, BERs in A&F are at minimum values from 20 dB down until 17 dB; from there the BER on the Direct link gets worse. BER and Overall BER performance in A&F is much better than D&F in most of the scenarios. In the Obstruction environment, the BER can only be calculated from 20 dB down to 12 dB. Thereafter, there are constant errors and CPU congestions that cause consecutive over runs. There are too many errors to process.

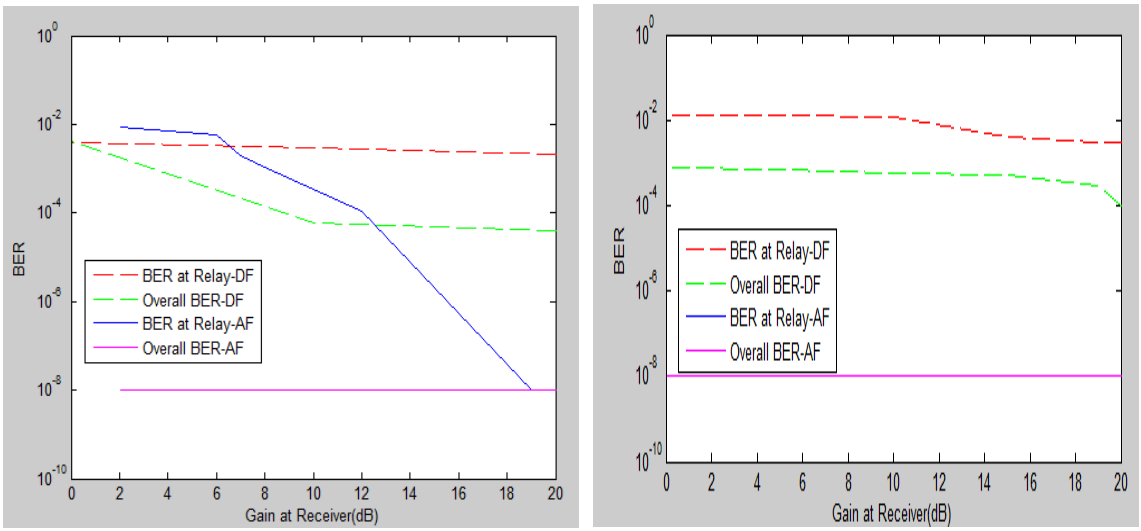

Fig 488: AF&DF (LOS On Left And Obstruction Environment On Right), 10 dB Gain At Relay, Variable Gain At Relay Link-RX

The above plot is obtained by maintaining 10dB at the Relay, 20 dB at the receiver for the Direct Link. Gains on the Relay Link are varied to observe the performance. In the LOS scenario, the Overall BER in A&F remains constant at its minimum value. Overall BER in D&F remains almost constant and continues to provide better performance until it is lowered down below 10 dB. In the obstruction environment, both the BER performances (A&F method) at the Relay are excellent since the Direct Link is operating at high gain and also Relay's gain is good enough to draw the signal without errors.

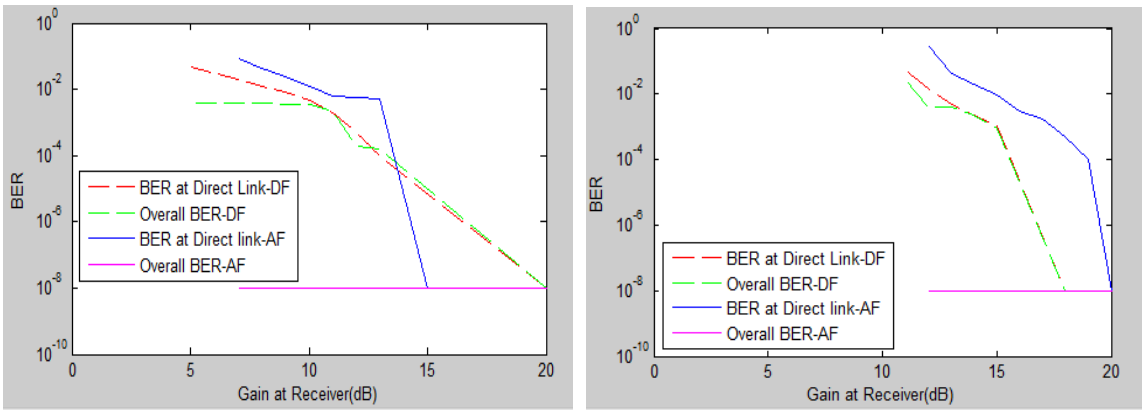

Fig 499: AF&DF (LOS On Left And Obstruction Environment On Right), 0 dB Gain At Relay, Variable Gain At Direct Link-RX

The above plot is obtained by maintaining 0dB at the Relay, 20 dB at the receiver for the Relay Link. Gains on the Direct Link are varied to observe the performance. In LOS, A&F still continues to provide reliable Overall system performance although there is 0 dB gain provided at the Relay. There is not much difference in the BER performance in case of D&F which indicates that the overall performance of the BER in D&F is worse in lower gain conditions. In the obstruction environment, A&F provides minimum overall BER because the Relay provides high gain.

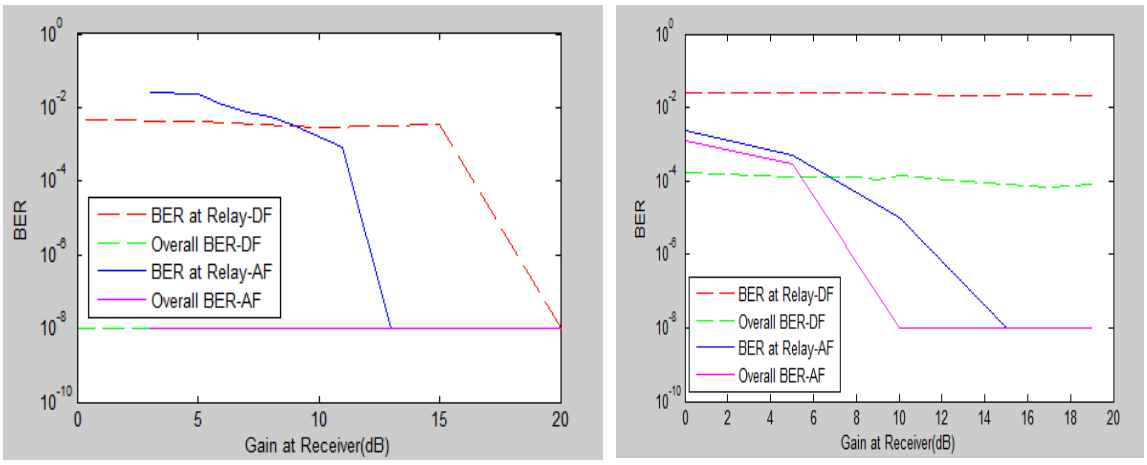

Fig 50: AF&DF (LOS On Left And Obstruction Environment On Right), 0 dB Gain At Relay, Variable Gain At Relay Link-RX

The above plot is obtained by maintaining 0dB at the Relay, 20 dB at the receiver for the Direct Link. Gains on the Relay Link are varied to observe the performance. In LOS, since the Direct Link is operated at high gain, the overall BER rates are maintained on the minimum level. In the obstruction environment, BERs in A&F remain constant and perform better until 15 dB and begin to show their difference as the gain is reduced. D&F also delivers good and consistent performance.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In our research, we have chosen to develop some fundamental cooperative communication techniques like Amplify & Forward and Decode & Forward strategies. We studied these using a hardware-based SDR laboratory. We also analyzed and compared them to interpret the best choice. GNU radio has been added with additional programming codes and reprogrammed to study the characteristics of the signal. This research project not only led us to learn the concepts of communication systems but also gave us a chance to learn techniques that can improve signal quality in real-world noisy environments.

5.2 Future Work

In the future, cooperative Relays can be implemented using multiple TXs, RXs, and Relays. The Relays can be made to forward the data only in the links that are more reliable. The links having higher SNR ratio will be chosen to carry the signal. Of course, this requires a lot of network coding and network protocols for the Relay to switch links.

Also, the Relay has been tested in stationary environments until now. In the future, a mobile Relay can be implemented by mounting a Relay on vehicles used for emergency purposes. Also, the MRC (Maximum Ratio Combining) concept at the Receiver can also increase spectrum diversity; we have only tested selection combining.

Optimal position of relay placement and relay selection can be implemented where the location of a relay in a cooperative relay network is important to take advantage of the diversity characteristics of a wireless network. The CSI (Channel state information) between the source and the final destination plays an important role to select a relay that transmits a signal from the source to the destination.

Cross-layer cooperative networking can be implemented so that this approach is not limited to one particular layer. More scope of research can be done in application of Network coding in cooperative networks.

APPENDIX

```
########################################

Packet.py on receiver end

(Extra code that is added to solve the purpose of our Project)

#####################################################

## Packet Decoder
#####################################################
class _packet_decoder_thread(_threading.Thread):
def __init__(self, msgq, callback):
_threading.Thread.__init__(self)
self.setDaemon(1)
self._msgq = msgq
self.callback = callback
self.keep_running = True
self.start()
def run(self):
start_pkt_no = -1
pkt_counter = 0
char_success_counter = 0
char_total_counter = 0
bit_success_counter1 = 0
bit_success_counter2 = 0
bit_total_counter = 0
fileout=open('DirectPath.txt','w')
while (self.keep_running & (pkt_counter !=1000)):
#os._exit(0)
```

```
msg = self._msgq.delete_head()
ok, payload = packet_utils.unmake_packet(msg.to_string(), int(msg.arg1()))
# print payload
thislen = len(payload)
(pktno,) = struct.unpack('!H', payload[0:2])
(pktno_check,) = struct.unpack('!H', payload[2:4])
if start_pkt_no == -1:
start_pkt_no = pktno
pkt_counter+=1
char_total_counter = char_total_counter + thislen-2
thispacket_bit_success_counter1=0
thispacket_bit_success_counter2=0
thispacket_char_success=0
for charindex in range(4, thislen):
c=payload[charindex]
if c in ['H','h']:
thispacket_char_success+=1
char_success_counter+=1
bitdiff1=(  ord(c) ^ ord('H') )
bitdiff2=(  ord(c) ^ ord('h') )
bitdiff_string1=bin(bitdiff1)
bitdiff_string2=bin(bitdiff2)
numerrors1 = 0
for charindex in range(1,len(bitdiff_string1)):
if bitdiff_string1[charindex]=='1':
numerrors1+=1
numerrors2 = 0
for charindex in range(1,len(bitdiff_string2)):
if bitdiff_string2[charindex]=='1':
numerrors2+=1
```

```
bit_total_counter+=8

bit_success_counter1=bit_success_counter1+(8-numerrors1)

bit_success_counter2=bit_success_counter2+(8-numerrors2)

thispacket_bit_success_counter1=thispacket_bit_success_counter1+(8-numerrors1)

thispacket_bit_success_counter2=thispacket_bit_success_counter2+(8-numerrors2)

success_ratio=(float)(pkt_counter)/(pktno-start_pkt_no+1)*100

success_ratio2=(float)(char_success_counter)/(char_total_counter)*100

if bit_success_counter1>=bit_success_counter2:

print "Packet number %d Bit Success %d Character Success %d" % (pktno, thispacket_bit_success_counter1,thispacket_char_success)

print "Statistics for h:  pktno = %4d, count=%d, all packets=%d, success=%f%% " % (pktno,pkt_counter,pktno-start_pkt_no+1,success_ratio)

success_ratio=(float)(char_success_counter)/(char_total_counter)*100

print "Statistics for H:  chars = %d, successful = %d, success=%f%%, fail = %f%%" % (char_total_counter,char_success_counter, success_ratio, 100-success_ratio)

success_ratio=(float)(bit_success_counter1)/(bit_total_counter)*100

fail_ratio=(float)(bit_total_counter-bit_success_counter1)/(bit_total_counter)*100

print "   Compared to H:  bits = %d, successful = %d, success=%f%%, fail = %.12f%%" % (bit_total_counter,bit_success_counter1, success_ratio, fail_ratio)

if pktno == pktno_check:

fileout.write("Packet number %d Bit Success %d\n" % (pktno, thispacket_bit_success_counter1))

else:

print "Packet number %d Bit Success %d Character Success %d" % (pktno, thispacket_bit_success_counter1,thispacket_char_success)

print "Statistics for h:  pktno = %4d, count=%d, all packets=%d, success=%f%% " % (pktno,pkt_counter,pktno-start_pkt_no+1,success_ratio)

success_ratio=(float)(char_success_counter)/(char_total_counter)*100

print "Statistics for h:  chars = %d, successful = %d, success=%f%%, fail = %f%%" % (char_total_counter,char_success_counter, success_ratio, 100-success_ratio)

success_ratio=(float)(bit_success_counter2)/(bit_total_counter)*100

fail_ratio=(float)(bit_total_counter-bit_success_counter2)/(bit_total_counter)*100
```

```
print "   Compared to h:  bits = %d, successful = %d, success=%f%%, fail = %.12f%%"
% (bit_total_counter,bit_success_counter2, success_ratio, fail_ratio)

if pktno == pktno_check:

fileout.write("Packet number %d Bit Success %d\n" % (pktno,
thispacket_bit_success_counter2))

if self.callback:

self.callback(ok, payload)

fileout.close()

os._exit(0)

#####################################################

## Packet Decoder

#####################################################

class _packet_decoder_thread2(_threading.Thread):

# This is for the relay, when the threshold is = -2

print 'just enterede'

def __init__(self, msgq, callback):

_threading.Thread.__init__(self)

self.setDaemon(1)

self._msgq = msgq

self.callback = callback

self.keep_running = True

self.start()

def run(self):

start_pkt_no = -1

pkt_counter = 0

char_success_counter = 0

char_total_counter = 0

bit_success_counter1 = 0

bit_success_counter2 = 0

bit_total_counter = 0
```

```
fileout=open('/home/aklilu1/gnuradio/grc/Project/relays/relay.txt','w')
print fileout
while (self.keep_running & (pkt_counter !=1000)):
#os._exit(0)
msg = self._msgq.delete_head()
# print 'dtep'
ok, payload = packet_utils.unmake_packet(msg.to_string(), int(msg.arg1()))
# print payload
thislen = len(payload)
(pktno,) = struct.unpack('!H', payload[0:2])
(pktno_check,) = struct.unpack('!H', payload[2:4])
if start_pkt_no == -1:
start_pkt_no = pktno
pkt_counter+=1
char_total_counter = char_total_counter + thislen-2
thispacket_bit_success_counter1=0
thispacket_bit_success_counter2=0
thispacket_char_success=0
for charindex in range(4, thislen):
c=payload[charindex]
#print c
if c in ['H','h']:
#print "hello"
char_success_counter+=1
thispacket_char_success+=1
bitdiff1=(  ord(c) ^ ord('H') )
bitdiff2=(  ord(c) ^ ord('h') )
#print bitdiff
bitdiff_string1=bin(bitdiff1)
bitdiff_string2=bin(bitdiff2)
```

```
numerrors1 = 0
for charindex in range(1,len(bitdiff_string1)):
if bitdiff_string1[charindex]=='1':
numerrors1+=1
numerrors2 = 0
for charindex in range(1,len(bitdiff_string2)):
if bitdiff_string2[charindex]=='1':
numerrors2+=1
bit_total_counter+=8
bit_success_counter1=bit_success_counter1+(8-numerrors1)
bit_success_counter2=bit_success_counter2+(8-numerrors2)
thispacket_bit_success_counter1=thispacket_bit_success_counter1+(8-numerrors1)
thispacket_bit_success_counter2=thispacket_bit_success_counter2+(8-numerrors2)
success_ratio=(float)(pkt_counter)/(pktno-start_pkt_no+1)*100
success_ratio2=(float)(char_success_counter)/(char_total_counter)*100
if bit_success_counter1>=bit_success_counter2:
print "Packet number %d Bit Success %d Character Success %d" % (pktno, thispacket_bit_success_counter1,thispacket_char_success)
print "Relay: Statistics for H:  pktno = %4d, count=%d, all packets=%d, success=%f%% " % (pktno,pkt_counter,pktno-start_pkt_no+1,success_ratio)
success_ratio=(float)(char_success_counter)/(char_total_counter)*100
print "Relay: Statistics for H:  chars = %d, successful = %d, success=%f%%, fail = %f%%" % (char_total_counter,char_success_counter, success_ratio, 100-success_ratio)
success_ratio=(float)(bit_success_counter1)/(bit_total_counter)*100
fail_ratio=(float)(bit_total_counter-bit_success_counter1)/(bit_total_counter)*100
print "Relay      Compared to H:  bits = %d, successful = %d, success=%f%%, fail = %.12f%%" % (bit_total_counter,bit_success_counter1, success_ratio, fail_ratio)
if pktno == pktno_check:
fileout.write("Packet number %d Bit Success %d\n" % (pktno, thispacket_bit_success_counter1))
else:
```

```
print "Packet number %d Bit Success %d Character Success %d" % (pktno, thispacket_bit_success_counter1,thispacket_char_success)

print "Relay  Statistics for h:  pktno = %4d, count=%d, all packets=%d, success=%f%% " % (pktno,pkt_counter,pktno-start_pkt_no+1,success_ratio)

success_ratio=(float)(char_success_counter)/(char_total_counter)*100

print "Relay  Statistics for h:  chars = %d, successful = %d, success=%f%%, fail = %f%%" % (char_total_counter,char_success_counter, success_ratio, 100-success_ratio)

success_ratio=(float)(bit_success_counter2)/(bit_total_counter)*100

fail_ratio=(float)(bit_total_counter-bit_success_counter2)/(bit_total_counter)*100

print "Relay      Compared to h:  bits = %d, successful = %d, success=%f%%, fail = %.12f%%" % (bit_total_counter,bit_success_counter2, success_ratio, fail_ratio)

if pktno == pktno_check:

fileout.write("Packet number %d Bit Success %d\n" % (pktno, thispacket_bit_success_counter2))

if self.callback:

self.callback(ok, payload)

fileout.close()

os._exit(0)

class packet_decoder(gr.hier_block2):

print 'i am into loop'


"""

Hierarchical block for wrapping packet-based demodulators.

"""

def __init__(self, access_code='', threshold=-1, callback=None):

"""

packet_demod constructor.

@param access_code AKA sync vector

@param threshold detect access_code with up to threshold bits wrong (0 -> use default)

@param callback a function of args: ok, payload

"""
```

```
#access code
if not access_code: #get access code
access_code = packet_utils.default_access_code
if not packet_utils.is_1_0_string(access_code):
raise ValueError, "Invalid access_code %r. Must be string of 1's and 0's" % (access_code,)
self._access_code = access_code
print 'into packet_decoder'
print 'Threshold = %d ' % (threshold)
thisthreshold = threshold
#threshold
if threshold < 0: threshold = DEFAULT_THRESHOLD
self._threshold = threshold
#blocks
msgq = gr.msg_queue(DEFAULT_MSGQ_LIMIT) #holds packets from the PHY
print 'msqq',msgq
correlator = digital.correlate_access_code_bb(self._access_code, self._threshold)
framer_sink = gr.framer_sink_1(msgq)
#initialize hier2
gr.hier_block2.__init__(
self,
"packet_decoder",
gr.io_signature(1, 1, gr.sizeof_char), # Input signature
gr.io_signature(0, 0, 0) # Output signature
)
#connect
self.connect(self, correlator, framer_sink)
#start thread
if thisthreshold == -1:
# Thread for Direct Path
```

```
_packet_decoder_thread(msgq, callback)
else:
# Thread for Relay path
_packet_decoder_thread2(msgq, callback)

###################################
```

Code for relay_process.py

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Top Block
# Generated: Wed Jul 17 12:56:08 2013
##################################################
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import uhd
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from gnuradio.wxgui import forms
from grc_gnuradio import blks2 as grc_blks2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx
class relay_process():
fr=open('relay.txt','r')
fd=open('DirectPath.txt','r')
# This parameter makes sure that pktno values are close to the previous value
#  It specifies how large of a gap can be seen in packet numbers and still be allowed.
allowed_pktno_gap = 200
```

```
# Read the first Relay packet
str=fr.readline()
str=str.split()
fr_start_pktno=int(str[2])
fr_bit_success=int(str[5])
print "first packet number from the relay",fr_start_pktno
# Read the first Direct Path packet
str=fd.readline()
str=str.split()
fd_start_pktno=int(str[2])
fd_bit_success=int(str[5])
print "first packet number from the direct path",fd_start_pktno
while fr_start_pktno < fd_start_pktno:
# Read from fr until it matches fd
# Loop until a valid packet is found
notvalidpacket = 1
old_fr_pktno = fr_start_pktno
while notvalidpacket==1:
str=fr.readline()
str=str.split()
fr_start_pktno=int(str[2])
fr_bit_success=int(str[5])
# A valid packet has a pktno larger than the previously processed packet, and is not
#   too much larger in pktno (by allowed_pktno_gap)
if (fr_start_pktno < old_fr_pktno) or (fr_start_pktno > (old_fr_pktno+200)):
notvalidpacket=1
print 'Not a valid fr packet = %d, old_pktno = %d' % (fr_start_pktno,old_fr_pktno)
else:
notvalidpacket=0
this_fr_pktno = fr_start_pktno
this_fd_pktno = fd_start_pktno
```

```
# Initialize counters
bit_success_counter = 0
bit_total_counter = 0
fr_only_pktcount = 0
fd_only_pktcount = 0
fr_best_pktcount = 0
fd_best_pktcount = 0
# Assuming collection of 500 packets, process first 300
for i in range(1,300):
# print 'Iteration %d' % i
# If the fr packet number is less than the fd packet number, search through the fr file
# and collect fr packet statistics until the latest fr packet in the file has a packet number
# equal or greater than fd
while (this_fr_pktno < this_fd_pktno):
# Read and collect statistics from fr until it matches fd
bit_success_counter = bit_success_counter + fr_bit_success
bit_total_counter+=4096
success_ratio = (float) (bit_success_counter)/(bit_total_counter) * 100
error_ratio=100-success_ratio
fr_only_pktcount += 1
print '%d Choose Relay: Overall bit error ratio after picking best packets is %f%%' % (
this_fr_pktno, error_ratio )
# Read from fr
# Loop until a valid packet is found
notvalidpacket = 1
old_fr_pktno = this_fr_pktno
while notvalidpacket==1:
# Read the next fr packet
str=fr.readline()
str=str.split()
this_fr_pktno=int(str[2])
```

```
fr_bit_success=int(str[5])
# A valid packet has a pktno larger than the previously processed packet, and is not
#   too much larger in pktno (by allowed_pktno_gap)
if (this_fr_pktno < old_fr_pktno) or (this_fr_pktno > (old_fr_pktno+allowed_pktno_gap)):
notvalidpacket=1
print 'Not a valid packet = %d ' % (this_fr_pktno)
else:
notvalidpacket=0
# print 'Fr packet = %d' % (this_fr_pktno)
# print 'Fd packet = %d' % (this_fd_pktno)
# if this_fd_pktno < this_fr_pktno:
# print 'Looking in DirectPath.txt for: %d' % this_fr_pktno
# If the fd packet number is less than the fr packet number, search through the fd file
# and collect fd packet statistics until the latest fd packet in the file has a packet number
# equal or greater than ff
while (this_fd_pktno < this_fr_pktno):
# Read and collect statistics from fd until it matches fr
bit_success_counter = bit_success_counter + fd_bit_success
bit_total_counter+=4096
success_ratio = (float) (bit_success_counter)/(bit_total_counter) * 100
error_ratio=100-success_ratio
print '%d Choose Direct Path: Overall bit error ratio after picking best packets is %f%%'
% ( this_fd_pktno,error_ratio )
fd_only_pktcount += 1
# Read from fr
notvalidpacket=1
old_fd_pktno = this_fd_pktno
while notvalidpacket==1:
str=fd.readline()
str=str.split()
```

```
this_fd_pktno=int(str[2])
fd_bit_success=int(str[5])
if (this_fd_pktno < old_fd_pktno) or (this_fd_pktno > (old_fd_pktno+allowed_pktno_gap)):
notvalidpacket=1
else:
notvalidpacket=0
# This is the case where the packet numbers are the same
if this_fd_pktno == this_fr_pktno:
# Compare bit success numbers and choose the best
if fd_bit_success <= fr_bit_success:
bit_success_counter = bit_success_counter + fr_bit_success
bit_total_counter+=4096
success_ratio = (float) (bit_success_counter)/(bit_total_counter) * 100
error_ratio=100-success_ratio
print '%d Match -- Choose Relay: Overall bit error ratio after picking best packets is %f%%' % (this_fr_pktno,error_ratio)
fr_best_pktcount += 1
else:
bit_success_counter = bit_success_counter + fd_bit_success
bit_total_counter+=4096
success_ratio = (float) (bit_success_counter)/(bit_total_counter) * 100
error_ratio=100-success_ratio
fd_best_pktcount += 1
print '%d Match -- Choose Direct Path: Overall bit error ratio after picking best packets is %f%%' % (this_fd_pktno,error_ratio)
# Read the pktno values from both files, making sure to only use valid packets
# Read from fr
notvalidpacket=1
old_fr_pktno = this_fr_pktno
while (notvalidpacket==1):
```

```
str=fr.readline()
str=str.split()
this_fr_pktno=int(str[2])
fr_bit_success=int(str[5])
# print "FR: Packet number = %s" % this_fr_pktno
if (this_fr_pktno < old_fr_pktno) or (this_fr_pktno > (old_fr_pktno+allowed_pktno_gap)):
notvalidpacket=1
else:
notvalidpacket=0
# Read from fd
notvalidpacket=1
old_fd_pktno = this_fd_pktno
while (notvalidpacket==1):
str=fd.readline()
str=str.split()
this_fd_pktno=int(str[2])
fd_bit_success=int(str[5])
# print "FD: Packet number = %s" % this_fd_pktno
if (this_fd_pktno < old_fd_pktno) or (this_fd_pktno > (old_fd_pktno+allowed_pktno_gap)):
notvalidpacket=1
else:
notvalidpacket=0
print 'Relay only packets = %d, Direct Path only packets = %d' % (fr_only_pktcount,fd_only_pktcount)
print 'When matching: Relay best packets = %d, Direct Path best packets = %d' % (fr_best_pktcount,fd_best_pktcount)
```

BIBLIOGRAPHY

[1] R.H. Hosking, Software Defined Radio Handbook, New Jersey: Pentek,2011.

[2] Wiki,"Wikipedia,"[Online].

Available:http://en.wikipedia.org/wiki/Radio_broadcasting.

[3] Wikipedia,"Wiki," [Online]. Available:

http://en.wikipedia.org/wiki/Downsampling.

[4] Dawei Shen, "The USRP Board," August,2005. [Online].

Available: astro.square7.ch/Datenblaetter/gnuradiodoc-4.pdf

[5] Firas Abbas Hamza, "The USRP under 1.5 X Magnifying Lens," Bosten,June 2008. [Online]

Available:microembedded.googlecode.com/files/USRP_Documentation.pdf.

[6] Matt Ettus, "USRP User's and Developer's Guide," Ettus Research LLC.[Online].Available:http://www.olifantasia.com/gnuradio/usrp/files/usrp_guide.pdf.

[7] Matthias Fähnle, "Software-Defined Radio with GNU Radio and USRP2 Hardware Frontend," Germany,2009. [Online]. Available: http://repo.zenk-security.com/Others/Software-Defined%20Radio%20with%20GNU%20Radio%20and%20USRP-2%20Hardware%20Frontend:%20Setup%20and%20FM-

GSM%20Applications.pdf.

[8] "Ettus," National Instruments, [Online]. Available: www.ettus.com. [Accessed 2013].

[9] "Ettus," Ettus, [Online]. Available: https://www.ettus.com/product/details/USRPPKG.

[10] Ettus, "Ettus research," [Online]. Available: https://www.ettus.com/product/details/RFX900.

[11] Josh Blum, "Ettus Research," [Online]. Available: https://www.ettus.com/product/details/VERT900.

[12] Ettus, "Ettus Research," [Online]. Available: https://www.ettus.com/product/details/SMA-Bulkhead.

[13] GNURADIO, "GNU Radio," [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki/InstallingGR.

[14] GnuRadio, "GRC," [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion.

[15] Naveen Manicka, "GNURadio Testbed," 2007. [Online]. Available: http://www.eecis.udel.edu/~manicka/Research/NaveenManicka_Thesis.pdf.

[16] Maruganti Murali Krishna, "EXPERIMENTAL STUDY OF COOPERATIVE COMMUNICATION USING SDR," Cleaveland, 2007. [Online]. Available:

http://www.csuohio.edu/engineering/ece/research/theses/2010/Marunganti%20Murali%20Krishna.pdf.

[17] Jon Petter Skagmo, Per Eid Fjuk,Kristian Rygh Jerndahl, "Digital Voice Communication with USRP2 & GNU Radio," February 2011.[Online]. Available:http://folk.ntnu.no/fjuk/mydoc/TTT4145_Radiocommunication.pdf

[18] Feng Ge, C. Jason Chiang, Yitzchak M. Gottlieb, Ritu Chadha, "GNU Radio-Based Digital Communications," Telcordia Technologies, New Jersey.

[19] Aria Nosratinia, Todd E. Hunter, "Cooperative Communication in Wirelss Networks," IEEE Communications Magazine, vol.42,no.10,pp.74,80,Oct 2004.

[20] "USRP users mailing list," Ettus, [Online]. Available: http://comments.gmane.org/gmane.comp.hardware.usrp.e100/4313.

[21] GNURadio, "Mailing list Discuss-gnuradio@gnu.org," [Online]. Available: http://gnuradio.4.n7.nabble.com/.

[22] Jhony Lee, "A Bidirectional Two-hop Relay Network Using Gnuradio And Usrp",NorthTexas,2011"[Online].Available:http://digital.library.unt.edu/ark:/67531/metadc84237/m1/1/.

VITA

Ganga Manjusha Yandamuri was born on December 07, 1988, in Vishakapatnam, Andhra Pradesh, India. She completed her schooling in Hyderabad and graduated from high school in 2006. She then completed her Bachelor's degree in Electronics and Instrumentation Engineering from Jawaharlal Nehru Technological University in Hyderabad in 2010. Upon completion of her Bachelor's, she was placed in SIFY as a Software Engineer. In August 2011, Ms.Ganga came to the United States to study Electrical Engineering at the University of Missouri-Kansas City (UMKC), specializing in Wireless Communications. During her graduation, she worked as a Graduate Teaching Assistant for a year. Upon completion of her Master's degree she will be rendering her services to RF industry.