

GNU Radio + USRP2 Implementation of a Single-Carrier Zero-Correlation-Zone CDMA System

MEENAKSHI SUNDARAM GANDHI
PRAVEEN KUMAR



**KTH Information and
Communication Technology**

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:75

Abstract:

GNU Radio is a software defined radio in which components implemented in hardware are implemented using the software. GNU Radio works along with the USRP board. The USRP board is used to transmit and receive the signal in the wireless channel.

CDMA system using the ZCZ (zero Correlation Zone) code is needed to be implemented in the GNU Radio and USRP board. Primary implementation and testing of CDMA using the ZCZ code is done with Matlab. The ZCZ come with the property of ideal impulsive auto correlation and zero cross correlation property within its ZCZ. However the Matlab implementation is not enough to make the system work using the USRP board. It is needed to be modified and implemented in GNU Radio to make the USRP board work.

Unlike the simulation done in the Matlab there are lot of problems involved in real-time implementation of the idea in GNU Radio. In Matlab there are lot of in built functions which are needed to be implemented manually in C++. In GNU Radio a chunk of data received at a particular instant may not be of same length all the time and it keeps varying, should be handled properly. For the feedback implementation such as code tracking needed to be handled carefully and all the feedback variables needed to be present in a single module. ZCZ code for single user system, that is one transmitter and one receiver has worked well, but the ZCZ code for multi user uplink had to be tested too.

This thesis aims at converting the Matlab implementation of ZCZ codes into real time GNU Radio implementation that works using the USRP board. This thesis will check the real time implementation of ZCZ code using USRP board and GNU Radio. This is done by implementing spreading of data at the sender side using ZCZ code, code synchronization at the receiver, code despreading at the receiver without code tracking and code despreading with code tracking are all done using ZCZ codes. This thesis also checks the feasibility of an asynchronous multi user communication without strict timing synchronization among the users with good results.

Acknowledgments

This thesis was carried in Nanyang Technological University (NTU) Singapore at INFINITUS lab.

I would like to thank my external supervisor Prof. Guan Young Liang for giving me this internship at his university and for his guidance.

I would like to thank my internal supervisor Prof. Svante Signell for accepting to be my internal guide at KTH and for his support.

I would like to thank Zilong Liu who designed the ZCZ code CDMA system and its implantation in Matlab.

I would like to thank Tang Xiaoyan for helping me in GNU Radio implementation and handling USRP board.

I would like to thank my parents for their care and encouragement.

I would like to thank my friends who were with me throughout my graduate studies.

Table of Contents

Abstract:.....	i
Acknowledgments.....	iii
List of Figures	ix
List of Tables	xi
Abbreviations.....	xiii
Chapter 1 Introduction	1
1.1 SDR	1
1.2 CDMA	2
1.3 CDMA codes.....	2
1.3.1 PN sequence	2
1.3.2 M-Sequence codes.....	2
1.3.3 Gold Sequence codes	3
1.4 ZCZ Codes.....	4
1.4.1 Construction of ZCZ codes	5
1.4.2 Bridge function and ZCZ codes.....	5
1.5 Motivation.....	5
1.6 Thesis Overview	7
1.6 Outline of Previous Work.....	7
Chapter 2 Software Defined Radio	9
2.1 Introduction	9
2.2 Nyquist Sampling theorem	10
2.3 Ideal RF digitalization	12
2.4 SDR digitalization	12
2.5 Digitalization of SCR	13
Chapter 3 GNU Radio	15
3.1 Introduction	15
3.2 Use of GNU Radio.....	16
3.3 GNU Radio flow graph Structure	16
3.4 Block types in GNU Radio.....	18
3.4.1 General Block	18
3.4.2 Synchronous Block	18
3.4.3 Decimation Block	18
3.4.4 Interpolation Block.....	19

3.5 Creating a signal processing block in Gnu Radio.....	19
3.5.1 Connecting signal processing block in C++ with Python	20
3.6 System design for transmitter with the GNU Radio	20
3.7 System design at the receiver for GNU Radio.....	21
3.8 Running the transmitter and receiver in GNU Radio	22
3.9 GNU Radio Tools and utilities	22
Chapter 4 USRP	25
4.1 Introduction	25
4.2 USRP 1	25
4.3 USRP2.....	27
4.4 USRP Network Series	27
4.4.1 USRPN200	27
4.4.2 USRPN210	28
4.5 Daughterboard RFX2400.....	30
4.6 GPS module.....	30
Chapter 5 System Overview.....	31
5.1 System specification	31
5.2 System design with USRP Hardware and GNU Radio	31
5.3 Differential encoding and Spreading at Transmitter	32
5.4 Differential decoding and despreading at receiver	32
5.5 Code Aquisition of the System.....	33
5.6 Code Tracking.....	35
5.7 ZCZ property.....	36
5.7.1 Auto correlation property of User 1 code.....	36
5.7.3 Cross Correlation Property of User 1 and User 2 code	38
Chapter 6 System implementation problem in GNU Radio and multi user transmission	41
6.1 Introduction	41
6.2 Problem of handling Stream of data.....	41
6.3 Problem with feedback functions	42
6.4 Implementing the built in functions in Matlab	43
6.5 Modifying the existing block.....	43
6.6 Synchronizing different Users in the uplink.....	43
Chapter 7 System Implementation in Gnu Radio	45
7.1 System design with GNU Radio.....	45

7.2 Transmitter without the spreading block	45
7.2.1 Modulator without spreading block	45
7.3 Receiver without the spreading block	46
7.3.1 Demodulator	46
7.4 System design at sender with spreading	47
7.4.1 Spreading Block.....	47
7.5 System design at receiver with despreading	48
7.5.1 CODE Aquisition	49
7.5.2 Code despreading without code Tracking	50
7.5.3 Decoding with CODE Tracking.....	52
7.5.4 Code Tracking.....	53
7.5.5 Code Tracking and despreading in single module	54
7.6 UP Link	54
7.6.1 Linux Server Time:.....	55
7.6.2 Division of Time into Slots:.....	55
Chapter 8 Lab set Up and results	57
8.1 One transmitter and one receiver setup and results:.....	57
8.2 Uplink (Two transmitters and one receiver):.....	59
Conclusions	62
Future Work	62
Appendix A	63

List of Figures

Fig 1.1: General Functional architecture of SDR	1
Fig 1.2: Shift Register Implementation	3
Fig 1.3: Shift registers implementation of Gold Sequence	4
Fig 1.4: Auto correlation Function and its zone ZACZ	4
Fig 1.5: Cross correlation Function and its Zone ZCCZ	5
Fig 1.6: Auto Correlation Function of ZCZ code	6
Fig 1.7: Cross correlation Function of ZCZ code	7
Fig 2.1: Block diagram for Software Defined Radio	10
Fig 2.2: Sine wave at 1Hz	11
Fig 2.3: Sine wave at 1Hz and sampling rate of 2Hz	11
Fig 2.4: Sine wave at 1Hz and Sampling at 3Hz	11
Fig 2.5: Sine wave at 1 Hz and sampling at 1.5Hz	12
Fig 3.1: GNU Radio with RF hardware	15
Fig 3.2: Flow graph for the Ringtone	17
Fig 3.3: Flow graph for hierarchical block	17
Fig 3.4: Inheritance structure of general block	18
Fig 3.5: Inheritance structure of general block	18
Fig 3.6: Inheritance structure of decimation block	18
Fig 3.7: Inheritance graph of Interpolator	19
Fig 3.8: Data transmission using GNU Radio and USRP	20
Fig 3.9: Flow graph for the Transmitter in GNU Radio	21
Fig 3.10: Flow graph for receiver in GNU Radio	22
Fig 4.1: USRP1 with Mother board and Two daughter boards for transmitter and receiver	26
Fig 4.2: USRP1 Block Diagram	26
Fig 4.3: USRP2 board front view	27
Fig 4.4: USRPN200 with daughter board	28
Fig 4.5: USRP N210 Motherboard, FPGA and GPS module	29
Fig 4.6: USRPN210 with daughterboard	29
Fig 4.7: Daughterboard RFX2400	30
Fig 5.1: System design at USRP hardware and GNU Radio	31
Fig 5.2: Differential Encoding and Spreading at transmitter	32
Fig 5.3: Differential decoding and despreading at the receiver	33
Fig 5.4: Peak position corresponding to the code aquisition	34
Fig 5.5: Constellation diagram for ideal system after despreading	35
Fig 5.6: Code Tracking of the Code aquisition position	35
Fig 5.7: Auto correlation property of code 1	37
Fig 5.8: Auto correlation property of User 2 code	38
Fig 5.9: Cross correlation of Code 1 and Code 2	39
Fig 6.1: Screen shot for Number of Input items	41
Fig 6.2: Stream data Flow in GNU Radio	42
Fig 6.3: CDMA Decoding block data flow with code	42
Fig 6.4: Data flow in GNU Radio for Block with loopback	43
Fig 6.5: Code Aquisition for Multi user without any time control	44

Fig 7.1: System Design with USRP board and GNU Radio.....	45
Fig 7.2: Flow graph for modulator	46
Fig 7.3: Flow graph for demodulator	46
Fig 7.4: Flow graph for modulator with Spreading	47
Fig 7.5: Flow chart for the spreading block.....	48
Fig 7.6: Flow chart for code aquisition.....	48
Fig 7.7: Flow chart for code synchronization	49
Fig 7.8: Code aquisition for real time data.....	50
Fig 7.9: Flow chart for code despreading without code tracking.	51
Fig 7.10: Flow chart for code despreading with code-tracking	52
Fig 7.11: Flow chart for code tracking.....	54
Fig 7.12: The graph for cross correlation for code 1 and code 2 when they are repeated thrice	55
Fig 7.13: Time division of time into slots from the reference time.	56
Fig 8.1: Lab set up with one transmitter and one receiver.....	57
Fig 8.2: Constellation diagram after despreading without code tracking in real-time scenario	58
Fig 8.3: Constellation diagram for code tracking algorithm	58
Fig 8.4: Lab set up for the Uplink two transmitters and one receiver	60
Fig 8.5: Code Aquisition diagram after two user start at time slots	60
Fig 8.6: GNU Radio plot of constellation diagram and time-domain display for single user	61
Fig 8.7: GNU Radio plot of constellation diagram and time-domain display for multi user uplink	61

List of Tables

Table 4.1: Comparisons of different USRP boards.....	30
Table 5.1: System specification for the test bed.....	31

Abbreviations

ADC	Analog to Digital Converter
AM	Amplitude Modulation
BER	Bit Error Ratio
CDMA	Code Division Multiple Access
CIC	Cascade Integrator Comb
CR	Cognitive Radio
DAC	Digital to Analog Converter
DBPSK	Differential Binary Phase Shift Keying
DDC	Digital Down Converter
DSA	Dynamic Spectrum Access
DS-CDMA	Direct Sequence Code Division Multiple Access
DUC	Digital Up Converter
FHSS	Frequency Hopping Spread Spectrum
FLL	Frequency Locked Loop
FM	Frequency Modulation
FPGA	Field Programmable Gate Array
GCP	Golay complement pairs
GPS	Global Positioning System
GRC	GNU Radio Companion
Kbps	Kilo bits per second
LFSR	Linear Feedback Shift Register
mHz	milli Hertz
MIMO	Multiple Input Multiple Output
M-sequence	maximal-length sequence

NTP	Network Time Protocol
PLL	Phase Locked Loop
RF	Radio Frequency
RRC	Root Raised Cosine
SDR	Software Defined Radio
SRAM	Static Random Access
SWIG	Simplified Wrapper and Interface Generator
TDMA	Time Division Multiple Access
UHD	USRP Hardware Driver
USRP	Universal software Radio peripheral
XOR	Exclusive OR
ZCZ	Zero Correlation Zone

Chapter 1 Introduction

1.1 SDR

There is an exponential growth in number of ways and the means people communicate with each other. They use for example voice, video and text messages, SDR (Software Defined Radio) provides a cost effective and flexible way to drive the communications forward. It provides lot of benefits to the service providers and to the end users.

SDR is a radio in which physical layer functions are achieved using software. IEEE P1900.1 group has worked to establish a definition and overview of SDR. SDR is a collection of hardware and software where some or all the radio functions are implemented through software or firmware which are programmable. These devices include FPGA (Field Programmable Gate Array), DSP, GPP or other programmable processors. With the SDR new technology can be added to existing technology without much change in hardware. Fig 1.1 shows the general functional architecture of SDR. GNU Radio used in this thesis is a type of SDR [12].

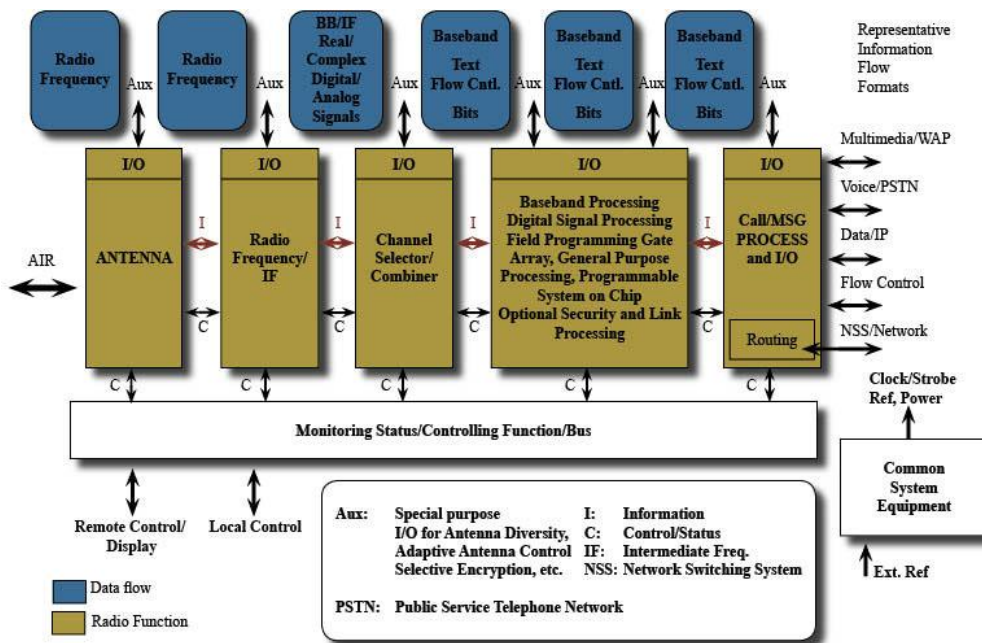


Fig 1.1: General Functional architecture of SDR [12]

1.2 CDMA

Using the spread spectrum multiple access techniques it is possible for signals with same RF (Radio Frequency) bandwidth to be transmitted simultaneously without any interference. CDMA (Code Division Multiple Access) is a type of spread spectrum technique. CDMA system using codes for different users are known as DS-CDMA (Direct Sequence Code Division Multiple Access). In this System if there are N users each user will be given their own code, $g_i(t)$, where $i=1,2,\dots,N$. Generally Codes used are orthogonal so that the cross correlation between two codes is zero. The advantage of using the CDMA system is that all users can share the full spectrum asynchronously [1].

1.3 CDMA codes

CDMA codes are the spreading sequence shared by the transmitter and receiver. The input data is multiplied by the spreading sequence. The bit rate of the spreading sequence is higher than the bit rate of the actual data. When the signal is received it is synchronised with the code and multiplied with the codes to remove the spreading and this process is known as despreading. The spreading codes should look like noise. That is the codes should have equal number of ones and zeros. The spreading codes for a CDMA system should have good correlation properties. When multiple signals are received, each with different spreading code the receiver should be able to pick the individual signal using that particular code. The spread signal should be uncorrelated with the other signals; they should behave as noise and should not interfere with despreading of a particular signal. The spreading of signals will result in high level of redundancy which enables the signal to cope up with interference from other signals in the same bandwidth. There are different CDMA codes some of them are explained in this section [2].

1.3.1 PN sequence

The ideal scenario for having spreading sequence would be random sequence of zeros and ones. Both the transmitter and the receiver should have the same spreading sequence. In other words both the transmitter and the receiver should have a method to generate the same bit stream at transmitter and receiver yet retain the properties of random sequence. This job is carried out by PN generator. PN generator will generate a spreading code with random sequence of zeros and ones.

PN sequence is generated by taking some initial value known as seed. The algorithm will not produce a perfect random sequence but will pass many test of randomness. Such numbers are called as pseudo noise or pseudorandom numbers. It is not possible to predict the sequence if the algorithm or the seed is unknown.

Important properties of PN sequences are randomness and unpredictability. In order to generate a sequence which is random, the sequence of numbers has to be random in some statistical sense. There are two criteria to validate a sequence of numbers as random, they are uniform distribution and independence. The uniform distribution means the sequence of numbers should be uniform. That is the frequency of occurrence of each number should be same. The independent property means that no one value can be inferred from others [2].

1.3.2 M-Sequence codes

The spread spectrum generated by a PN generator consisting of XOR (Exclusive OR) and shift registers are called LFSR (Linear Feedback Shift Register). The LFSR is 1 bit storage devices consist of an output line which gives the value of currently stored device and an input line. At a discrete time instant value in the storage device is replaced by the value in the discrete line. This

causes 1 bit shift in the entire register. If LFSR contains n bits there could be 1 to $(n-1)$ gates. The presence or absence of a gate is represented by presence or absence of a polynomial term. General equation for LFSR by XOR terms is represented as

$$B_n = A_0 B_0 \oplus A_1 B_1 \oplus A_2 B_2 \oplus \dots \oplus A_{n-1} B_{n-1}$$

For $A_i=0$ corresponding XOR circuit will be removed. Fig 1.2 shows the 4 bit shift register for the equation $B_3 = B_0 \oplus B_1$. The sequence generated by LFSR has advantages such as the sequence is random with long periods. They can be easily implemented in hardware at high speed. The speed is important because the spreading rate has to be higher than the data rate. It can be proved that LFSR has a period of $N=2^n-1$. The all zero sequence will be obtained only if the all the co-efficient are zero or the initial content of LFSR is zero. This LFSR always generate a sequence of period N and those sequences are called m-sequences (maximal-length sequence) can be used in FHSS (Frequency Hopping Spread Spectrum) but not in CDMA system this is due to their low cross-correlation property [11].

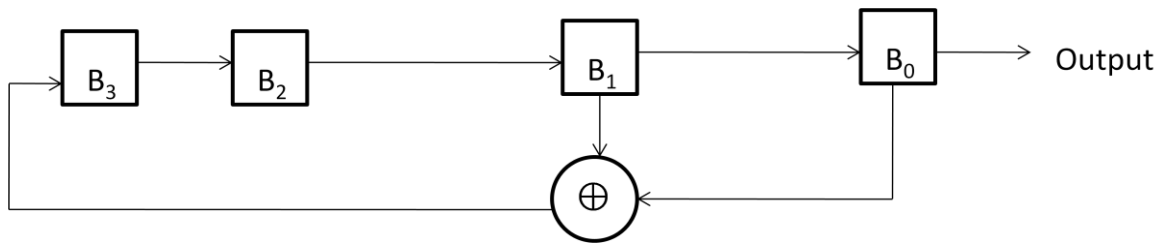


Fig 1.2: Shift Register Implementation

1.3.3 Gold Sequence codes

The m-sequences cannot be used in CDMA system since m-sequences for each user doesn't have good cross correlation property. However the gold sequence does satisfy the cross correlation property. The gold sequence is constructed by XOR of two m-sequences at same clocking. Fig 1.3 shows two pairs of shift registers generating two m-sequences and they are bitwise XORed. The expression for the LFSR at the top of the Fig 1.3 is $B_4 = B_3 \oplus B_0$ and the expression for LFSR at the bottom is $C_4 = C_3 \oplus C_2 \oplus C_1 \oplus C_0$. The m-sequence from these two LFSR is bitwise XORed to generate the gold sequence. The resulting sequence is not maximal. In order to design a desired gold code preferred pairs of m-sequences are used. The preferred pair of m-sequences are selected from table of pairs or generated from algorithm [11].

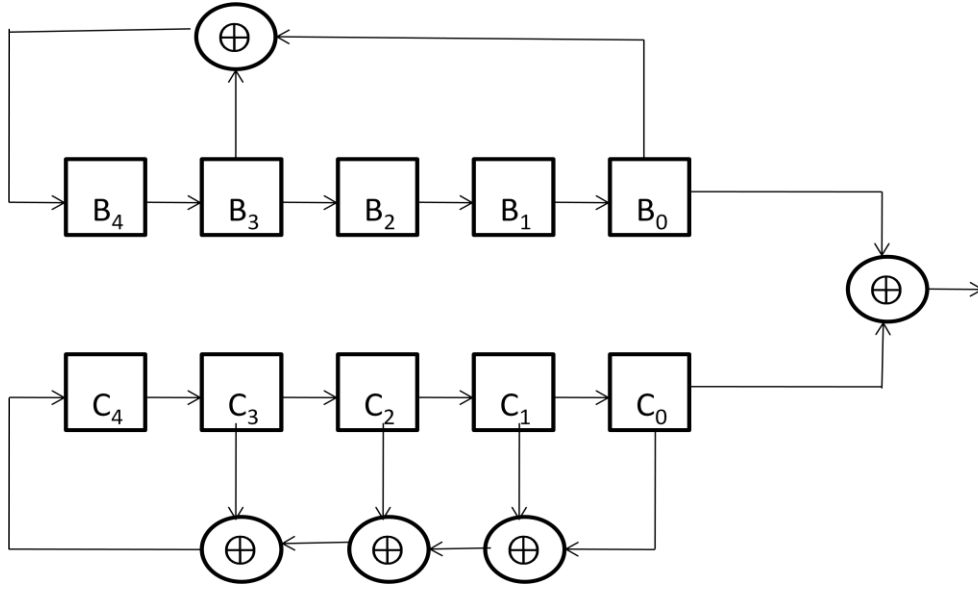


Fig 1.3: Shift registers implementation of Gold Sequence

1.4 ZCZ Codes

The ideal DS-CDMA system should have very low auto-correlation side lobes and very low cross correlation side lobes. It is proven that it is impossible to develop ideal impulsive auto correlation function and ideal zero cross correlation function. That is auto correlation has a single peak at $t=0$ and zero or almost zero values at other values of t . For auto correlation codes will have zero amplitude at all values of t , but it is not possible and this will lead to co-channel interference in CDMA systems. Though it is not possible to develop zero cross correlation function and impulsive auto-correlation, but it is possible to develop ZCZ (Zero Correlation Zone) in auto and cross correlation. Z_{ACZ} is a zone where all the amplitude of auto-correlation function will be zero except at $t=0$ and outside the zone there will be non-zero amplitude. Similarly Z_{CCZ} is a zone where there is zero amplitude for cross-correlation at particular zone and non-zero amplitude outside the zone. Fig 1.4 and fig 1.5 shows Z_{ACZ} and Z_{CCZ} respectively [27].

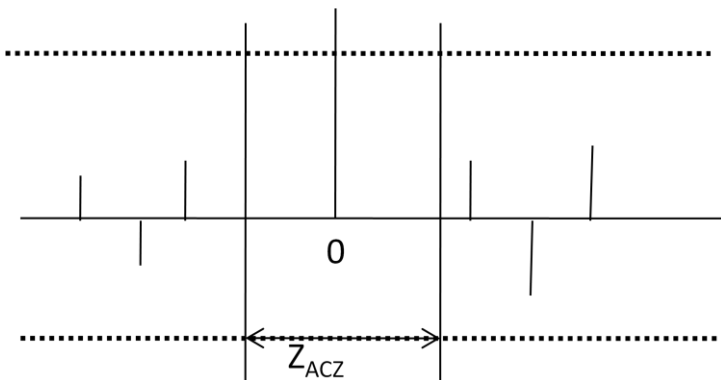


Fig 1.4: Auto correlation Function and its zone Z_{ACZ}

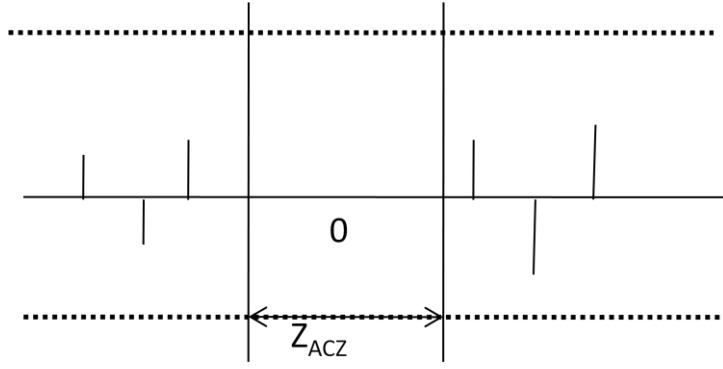


Fig 1.5: Cross correlation Function and its Zone ZCCZ

1.4.1 Construction of ZCZ codes

Construction of ZCZ code involves creation of Golay complement pairs (GCPs) which have zero cross correlation zones. Construction of GCP is given in [34]. Suppose we have a GCP (A,B), it is possible to generate another GCP (A₂,B₂) which is mutually orthogonal with (A,B) i.e., A₂ = reverse(B) and B₂ = reverse(A). With these notations in hand it is possible to generate two ZCZ codes as follows

$$Z_1 = [A, O, B, O]$$

$$Z_2 = [A_2, O, B_2, O]$$

Where O is the vector of zeros whose length is equal to the width of ZCZ. There are two types of ZCZ codes such as periodic ZCZ and aperiodic ZCZ. In this thesis work aperiodic ZCZ is used. To create periodic ZCZ cyclic prefix has to be inserted. Cyclic prefix is not inserted for construction of aperiodic ZCZ code.

1.4.2 Bridge function and ZCZ codes

Both the bridge function and ZCZ are ternary. Bridge function is used in synchronous multicarrier CDMA [35]. However ZCZ codes are used for asynchronous single-carrier CDMA which follows time domain (as opposed to frequency domain) signal processing algorithm.

1.5 Motivation

With the extended usage of mobile devices, the scope of wireless communication has increased drastically. The spread spectrum technology was initially used for military communication to resist jamming and to achieve low probability of detection. However they are now extensively used in commercial cellular and personal communication systems. The codes in CDMA system play a vital role in multiple access systems.

In section 1.3 and 1.4 we see different kind of codes. There are lot of research actively carried out on this CDMA code. There is a kind of code known as complementary code that was first found in the year 1961 by Golay where the auto correlation property is zero at all the values of t except at t=0. This complementary code has generated lot of usability scope like in MC-CDMA [29] and Radar system [30]. Lot of research are currently going on about the complementary codes to achieve better correlation with better sequence length such as quasi-complementary sequence set and weighed complementary sequence set [31] [32]. There is another kind of code known as Event shift orthogonal code constructed from complementary codes which exhibits aperiodic correlation function, which

takes zero-values for all even shifts. There is also a new kind of orthogonal sequence that has zero aperiodic correlation function for all shifts [33].

Like these codes ZCZ is also a special code which exhibits zero cross correlation and impulsive auto correlation function but restricted to its zone. The main motivation of ZCZ is that, even different users are asynchronous (which is typical in uplink CDMA channel), provided that they have small inter-user-delays which are within the ZCZ window, one can always achieve the zero multi-user interference (MUI) and zero multi-path interference (MPI). Its property is briefly discussed in section 1.4. This code is used in this project. Though the other codes such as quasi-complementary sequence set, weighed complementary sequence and new kind of orthogonal sequence exhibits idealistic behaviour they are not tested for real-time implementation. The ZCZ code has been put into research for real time experiments using USRP (Universal software Radio peripheral) boards and GNU Radio.

The initial implementation of the ZCZ code is carried out in the Matlab. Based on the Matlab implementation of ZCZ code, real-time implementation is carried out using USRP board and GNU Radio test bed, where the USRP board is the RF hardware and GNU Radio is a SDR. Fig 1.6 represents the idealistic impulsive auto correlation function but within a zone but outside the zone there is non-zero amplitude. Fig 1.7 represents cross correlation of ZCZ code, where there is zero cross-correlation within the zone which is an idealistic behaviour.

This project explores the feasibility of this idealistic ZCZ code implementation in USRP board and GNU Radio. Then the performance of the ZCZ code in the real-time is compared with Matlab implementation. Then research study, about the feasibility of multi-user uplink implementation of ZCZ code and find a method to implement the multi-user uplink.

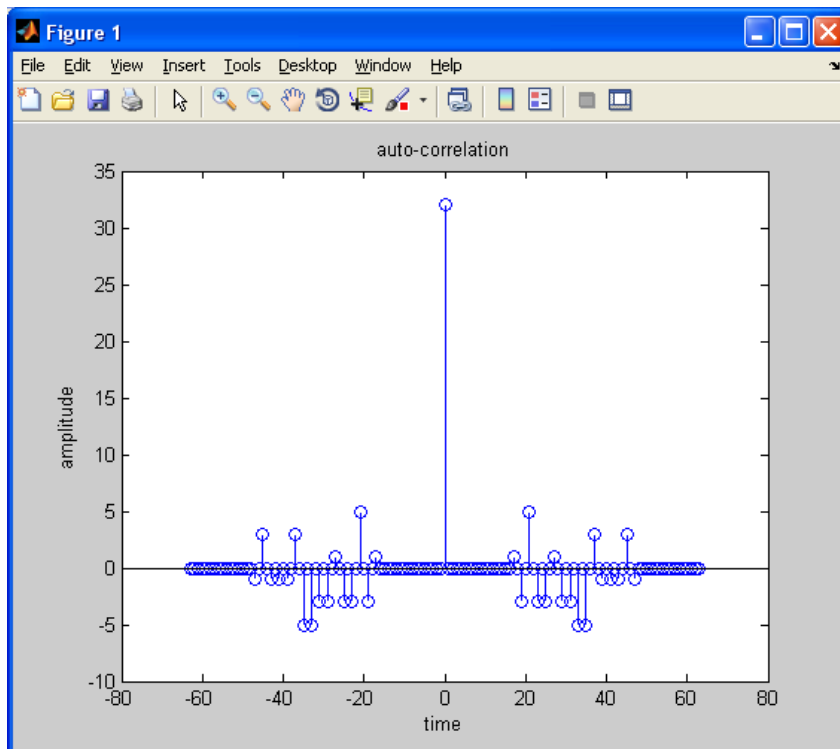


Fig 1.6: Auto Correlation Function of ZCZ code

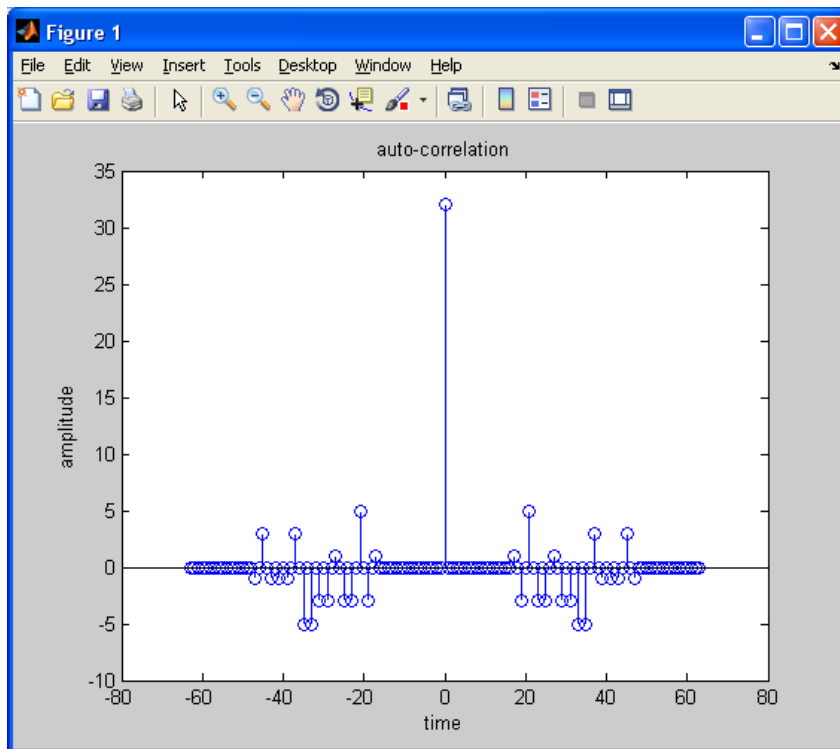


Fig 1.7: Cross correlation Function of ZCZ code

1.6 Thesis Overview

The general description about SDR, general structure of SDR is presented in Chapter 2. A brief description of GNU Radio (a kind of SDR) and transmission and reception using GNU Radio is given in chapter 3. In chapter 4 description about the USRP board which is used as the front end in this project, in this chapter we also see different types of USRP boards available and their variations.

In chapter 5 we see the general system description. In this chapter we will also see about the modulation type, about the data-rate, code tracking, code acquisition and briefly see the properties of ZCZ code which is the code used in this project for CDMA systems. In chapter 6 we will deal with the general difficulties in implementing the GNU Radio compared to a general implementation in Matlab and the problem of multi user uplink. Chapter 7 gives detailed description of implementation of spreading, despreading, code tracking signal processing block in C++, multi user uplink support and its time synchronization.

1.6 Outline of Previous Work

ZCZ codes are ternary spread spectrum with zero aperiodic autocorrelation and cross correlation properties. Within the prescribed time delay users can achieve zero multi path and multi user interference. The aim of the test bed is to implement the CDMA system using ZCZ codes in USRP boards.

GNU Radio is the SDR used with the USRP board. GNU Radio consists of several in built signal processing blocks and new signal processing blocks can also be used. GNU Radio consists of Python and C++. The signal processing blocks are written in C++ and Python to do the

implementation. Using Python data processing pipeline is created. GNU Radio is implemented in the Unix based system.

The previous work in the test bed involved the creation of the CDMA system for ZCZ in Matlab, CDMA code spreading and despreading without code tracking was implemented by editing the existing modules. The disadvantage of editing the existing module is that the existing module's characteristic would be altered which could be a problem in later implementation. In the implementation of decoding without code tracking there is no mechanism to check the validity of the start position of the symbol. After some duration there could be a possibility that the start position could be slipped away and hence the code tracking will be essential if the start position gets slipped. This implementation is done by having the decoding module alongside the code tracking. Everything is done for single user i.e. there will be a single transmitter and receiver.

The work of this thesis was to create spreading and despreading as a separate module. This thesis also involved creating the despreading module with the code tracking implementation and to develop an uplink transmission where different users can join the system at different times but the receiver should distinguish them uniquely.

Chapter 2 Software Defined Radio

2.1 Introduction

A software defined radio is a software program which does the functionality of hardware. The digital signal processing blocks are now incorporated into the software. This gives more flexibility, since if the functionality of the radio has to be altered it can be made possible by changing the software. This gives the advantage of improving the performance by changing the software and without having to replace the hardware [22]. This gives edge for the radio that uses the software internally and do not depend the hardware to perform all the functionalities. Software defined radio does the some or all the function of physical layer. Software determines the functionality of the radio and changes the performance, functionality of the radio [23].

The term software defined radio was given in the year 1984 by team of garland Texas division of E-Systems Inc. Now there is a dedicated forum for SDR known as SDR Forum (Wirelessinnovation.org). The SDR Forum is a non-profitable forum where group of people have advanced knowledge in SDR, CR (Cognitive Radio), and DSA (Dynamic Spectrum Access) in all market will discuss about the advancement of wireless communication.

In software defined radio, processing is done in the digital domain. At the receiver signal is passed to ADC (Analog to Digital Converter) followed by the software defined radio. It is the other way around for the transmitter where data passed to DAC (Digital to Analog Converter) before transmitting it through antenna. Fig 2.1 shows the block diagram for the SDR.

The Front end of the SDR is the RF section which takes the responsibility of transmitting and receiving signals at its working frequency band. At the receive path RF section amplifies the signal and mixes it with the local oscillator to down convert it to the IF frequency. In the transmit path RF section receives the signal from the IF section convert it to the required frequency. After it converts the signal to the required frequency it uses the matching circuitry for the maximum power transfer and then it is presented to the antenna for transmission.

The IF section of the receive side do the ADC, DDC (Digital Down Converter) which does the signal demodulation and processing before passing it on to the Base Band Processing Section. Similarly at the transmission side it does the DAC, DUC (Digital Up Converter) which performs the modulation on to the carrier and converts the digital to analog signal for radio transmission.

In the baseband section signal in digital format is processed. In this section the new and complex protocols are added and hence work of this section is becoming more and more complicated.

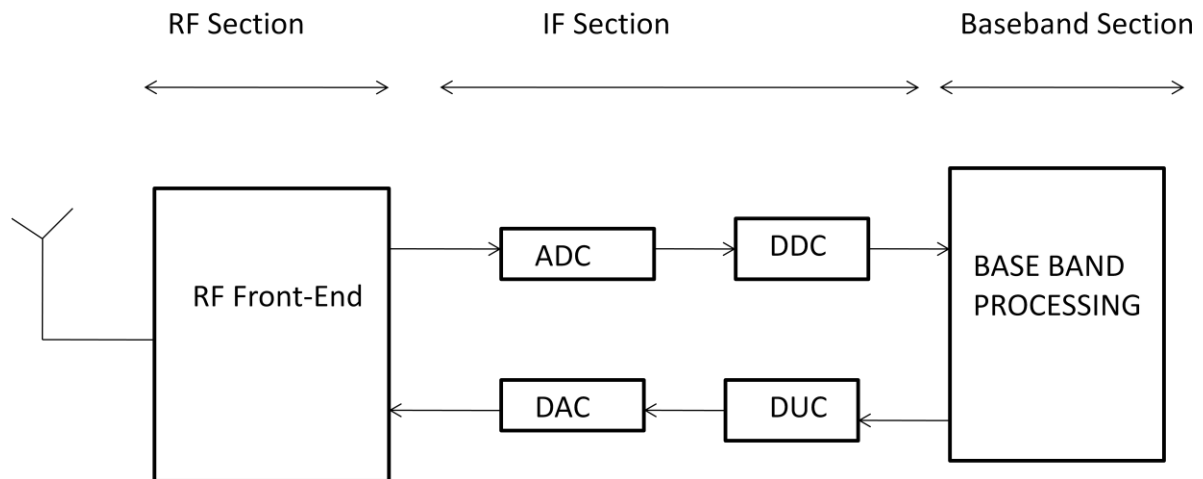


Fig 2.1: Block diagram for Software Defined Radio [24]

2.2 Nyquist Sampling theorem

Nyquist sampling theorem gives the criteria for converting the analog signal to the digital signal. The sampling theorem states the following

“Sampling frequency should be at least twice the highest frequency contained in the signal”

It can be mathematically stated as

$$f_s \geq 2f_c$$

Where f_s is the sampling frequency and f_c is the highest frequency contained in the carrier. Let us consider sine wave of 1Hz as shown in fig 2.2. Figure 2.3 shows the sine wave at 1 Hz but the sampling is done at 2Hz. In this all the peak values are noted and it has just enough data to reconstruct the sine wave. Figure 2.4 shows the sine wave at 1 Hz but the sampling is done at 3Hz. At this sampling rate more information about the sine wave is captured and it provides better reconstruction as compared to one in figure 2.3. Figure 2.5 shows the sine wave at 1 Hz but the sampling is done at 1.5Hz. The sampling rate in Fig 2.5 is less than twice the highest frequency component of the signal which is 1Hz. There are not enough samples to capture the negative peak. By sampling at this rate, it not only gives less information it provides wrong information about the signal and the signal cannot be reconstructed.

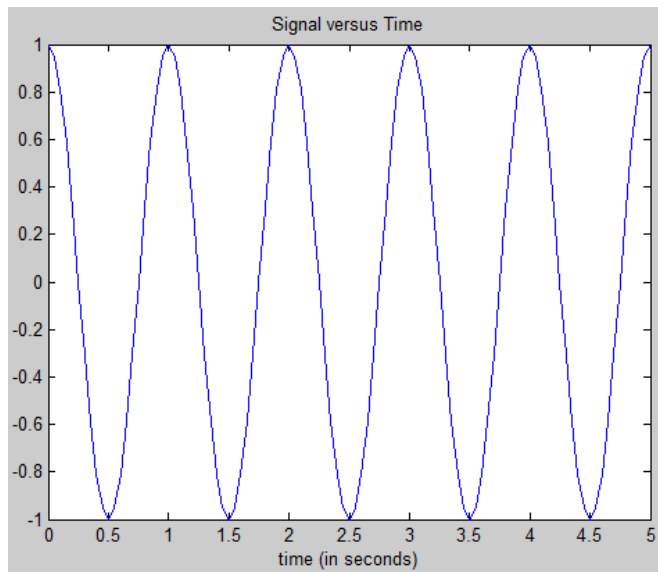


Fig 2.2: Sine wave at 1Hz

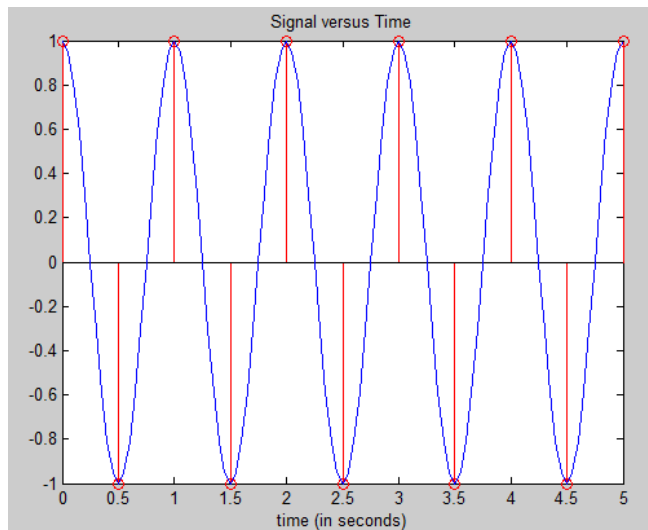


Fig 2.3: Sine wave at 1Hz and sampling rate of 2Hz

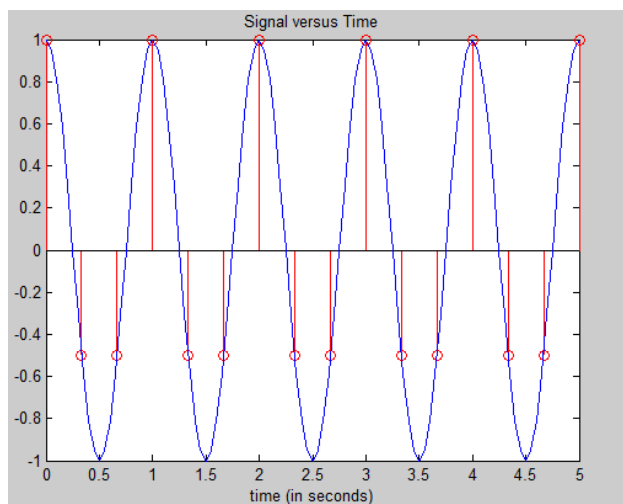


Fig 2.4: Sine wave at 1Hz and Sampling at 3Hz

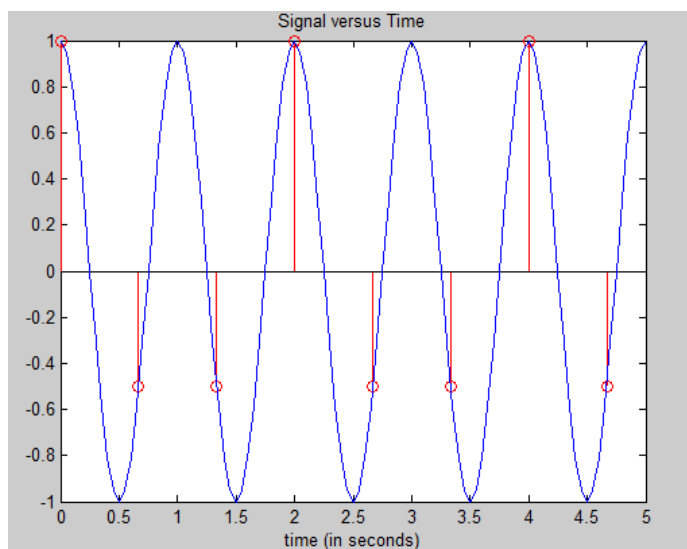


Fig 2.5: Sine wave at 1 Hz and sampling at 1.5Hz

2.3 Ideal RF digitalization

The idea behind the RF digitalization is that ADC digitalizes the signal collected at the antenna. The digitalized component is passed to signal processing software. In this way ADC, signal processing software is next to each other. There is no IF section in between. This is an ideal scenario.

The main advantage of this method is that same radio can be used for different frequency bands. However the frequency component will be very high in the order of GHz. The sampling rate should also be high to satisfy the Nyquist theorem as explained in section 2.2. The signal strength also varies which will be difficult for current ADC to handle. A lot of research is conducted currently on this field. The ideal scenario is to have the SDR as close as possible to the RF front end. However this is not possible and we have SDR after IF.

Power consumption and performance are the trade offs in the ADC. If the ADC work at high speed as it has no IF section it will consume lot of energy, higher consumption of energy will produce heat. If heat is produced then considerable cooling system has to be used. If such a system is used in the mobile devices considerable battery power is used for the cooling system. Some research has been done to increase the performance of ADCs while other research has been done to reduce the power consumption [26].

2.4 SDR digitalization

In section 2.3 the ADC is placed immediately after the signal collected from antenna. However the SDR designers have placed an IF section before the ADC. This digitalization is also known as the IF digitalization due to the presence of the IF section. This design requires a front end which consist of RF filter, RF/IF converter and IF filter. The front end converts the incoming signal to an IF signal. Before demodulation the ADC will digitalize the signal and give it to SDR.

The advantage of using this method is that due to the presence of an IF section the speed and resolution required for ADC is less. The computation required also is less since the RF filter will limit the number of received channels and no need to do software channel selection [26].

2.5 Digitalization of SCR

The traditional radio which fine tunes the demodulated signal through software is known as Software Controlled Radio. In SCR digitalization is carried out at baseband level and this digitalization is termed as baseband digitalization. This type of digitalization is common in traditional transceivers. This digitalization is used for music adjustment and equalization. This technique is used in stereo-music equipment. The SCR does not perform any of the radio function in software and hence it is not termed as SDR [26].

Chapter 3 GNU Radio

3.1 Introduction

GNU Radio is an open source SDR which runs in an UNIX environment and implements the digital signal processing blocks. The advantage of GNU Radio is that it is an open source software development tool kit. It can be used with external RF hardware such as USRP to create SDR [3]. GNU Radio has a very active community which works on and share their ideas for the improvement of GNU Radio. In order to support and develop GNU Radio Matt Ettus who is a member of GNU Radio team has started the company EETUS Research LLC [5] and built USRP.

The GNU Radio consists of Python at the front end and C++ at the back end. The signal processing implementations are done in C++. The signal processing blocks are organised and connected by Python [14] [15]. With this developers are able to develop simple to implement and rapid to use software radios.

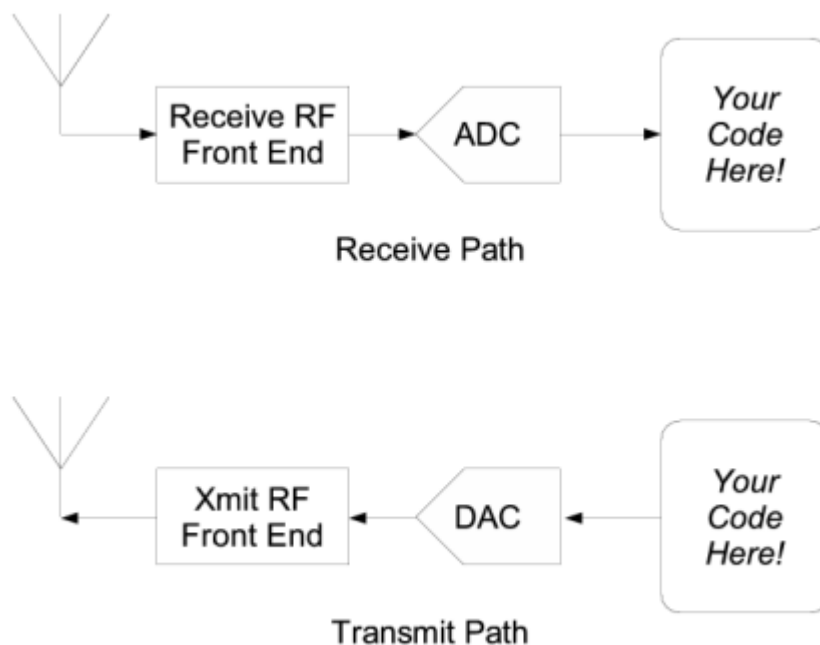


Fig 3.1: GNU Radio with RF hardware [7]

Fig 3.1 represents the implementation of GNU Radio using some RF hardware. From Fig 3.1 we can see an antenna, RF front end, ADC and some code implemented by GNU Radio. ADC bridges the real-time continuous signal and discrete signal which is manipulated in the software. Two characteristics of ADCs are sampling rate and dynamic range. Sampling rate is the measure of number of times the analog signal is measured. Dynamic range is the measure of difference between largest and smallest signal. Dynamic range is the function of number of bits in the ADC output. For example if 12 bit ADC is used it represent 4096 level, 14 bit ADC is used it represents 16384 signal levels.

According to Nyquist theorem explained in section 2.2, if the sampling rate is around 20MHz (mille Hertz) and if the signal of interest is around 100MHz then the RF front end will convert the signal to the range of 0 to 10MHz. This is explained in chapter 2 in section 2.4. The signal after passing through the ADC is manipulated by the GNU Radio. The GNU Radio has a library of signal processing block written in C++. The signal processing blocks are connected and organised by the Python front end. Attributes of the blocks include the number of input ports, number of output ports

and the data type that is flowing through the each block. Most commonly used data types are complex, float, short.

3.2 Use of GNU Radio

The GNU Radio is used both in academic and in commercial environment. The GNU Radio is used for research as well as real time radio communication purposes. Some of the GNU Radio projects which are in progress are as follows [7]

1. GNU Radio is used in TDMA (Time Division Multiple Access) waveforms
2. GNU Radio used in RADAR system takes advantage of the broadcast TV as its signal source.
3. GNU Radio is used in radio astronomy
4. GNU Radio is used in Amateur radio transceiver.
5. GNU Radio is used in distributed measurement of spectrum utilization.
6. GNU Radio is used in RFID detectors and readers.
7. GNU Radio is used with software GPS (Global Positioning System).
8. GNU Radio is used with MIMO (Multiple Input Multiple Output) processing.

3.3 GNU Radio flow graph Structure

The GNU Radio flow graph structure can be better explained by an example Python program. The example program dial_tone.py is given in appendix A. This example is used to create a dial tone similar to the dial tone in US. First two sine waves with frequency 350 and 440Hz are created using the function `gr.sig_source_f`. The `sig_source_f` is the C++ signal processing block which creates a signal with desired frequency and amplitude. The `audio.sink` is the signal processing block which writes its input to the sound card. In order to connect these blocks the `connect` method is used. The class `my_top_block` inherits from `gr.top_block` which provides the function to connect.

The `connect` function takes two parameters the source endpoint and the destination endpoint. The endpoint represent the two components namely the signal processing block and the port number. The port number specifies the input or output port. If the port number is 0 the block can be used alone without the port number for example

```
fg.connect((src0, 0), (dst0,0))
```

```
fg.connect ((src0, dst0))
```

The above two statements are the one and the same. It can be seen from the connection that it follows a graph structure. The graph structure is given in fig 3.2. It could be noted from the fig 3.2 that sine source with 350Hz from its output port 0 is connected to input port 0 of audio sink, sine source with frequency 440Hz from its output port 0 is connected to input port 1 of the audio sink. This representation of graph is commonly known as the flow-graph representation of GNU Radio. This is written in the coding as

```
src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)  
src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
```

```
fg.connect((src0,0), (dst,0))  
fg.connect((src1, 0), (dst, 1))
```


The above 4 lines of codes are present in dial_tone.py in Appendix A in the lines 11, 12, 14 and 15 respectively. The src0 and src1 are the sine signals with frequency 350Hz and 450 Hz respectively. The connect function is used to connect these two signals as explained earlier.

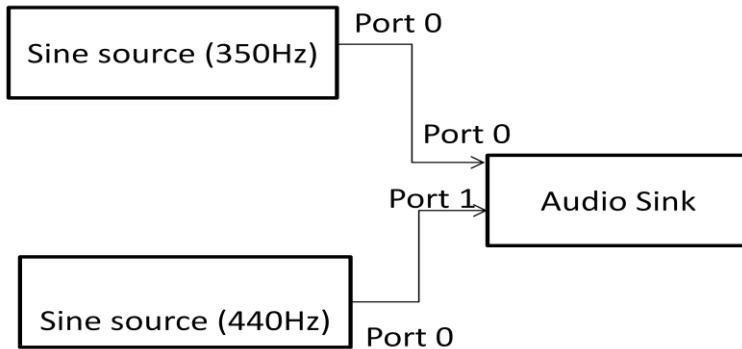


Fig 3.2: Flow graph for the Ringtone

In GNU Radio, data flows through each block like a stream. That is the data from sine source will flow into the audio sink as stream of data. In some instance it is essential to connect several blocks into a single block. Such a connection is done using the hierarchical block [17]. In order to implement the hierarchical block `gr.hier_block2` is inherited instead of `gr.top_block`. In the hierarchical block self will be used as source and sink. The block which uses hierarchical blocks will look to the outside world as a single block, but it is a collection of several blocks together. Simple graphical representation of hierarchical block is represented in fig 3.3.

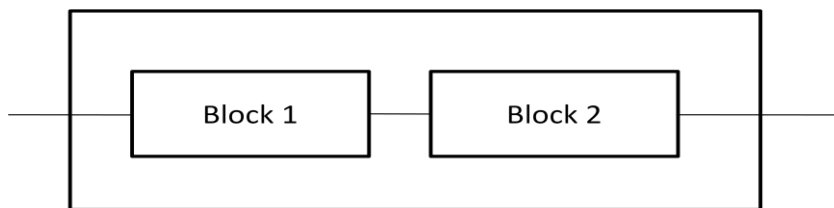


Fig 3.3: Flow graph for hierarchical block

In order to use hierarchical block in the code, first `gr.hier_block2` class is inherited into the class which is going to act as hierarchical block. In hierarchical block connect function will have self in the start and the end this is to make sure all the block between the two self is looks to the outside world as a single block. The hierarchical block is represented in the code as follows

```

class class_name(gr.hier_block2)

Block1=gr.block1(...)

Block2=gr.block2(..)

Self.connect(self,Block1,Block2,self)
  
```

In the code above `gr.hier_block2` is inherited into the class. Then two signal processing blocks Block1 and Block2 are defined and these two blocks are placed between the self in the connection function. The significance of this connection is that the functionality of these two signal processing block connected together will look to the outside world as single signal processing block.

3.4 Block types in GNU Radio

GNU Radio consists of several built-in signal processing blocks. Signal processing block in GNU Radio can be categorised based on the ratio of number of input to the number of output of the particular signal processing block. Several types of signal processing blocks are available such as synchronous, decimation, interpolation and general blocks.

3.4.1 General Block

The general block which is defined in GNU Radio as `gr_block` is an abstract C++ class. It is the base class for all the signal processing blocks. The inheritance structure of `gr_block` is given in fig 3.4. The inheritance in C++ is an object oriented programming concept in which a class can inherit properties of already existing class, in addition to its own properties. The inherited properties can also be modified if and when needed. The inheritance in C++ is used to increase the code reusability. Signal processing block which has inherited the `gr_block` will implement the signal processing block at `general_work` method. The default input and output ratio is 1:1. Which means the number of output data is same as number of input data flow into the block [17].

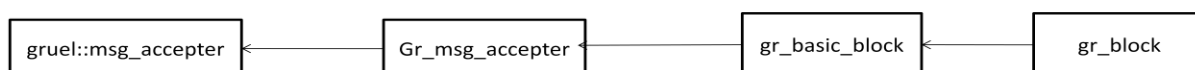


Fig 3.4: Inheritance structure of general block

3.4.2 Synchronous Block

The synchronous block which is defined in GNU Radio as `gr_sync_block` is a C++ class. The `gr_sync_block` is derived from `gr_block`. Synchronous block has 1:1 input to output ratio with optional history. Important difference between `gr_sync_block` and `gr_block` is that it defines `work` instead of `general_work`. Another difference is that it omits unnecessary `ninput_items` parameter and call `consume_each` which makes our work bit easier [18] [19]. The inheritance structure of synchronous block is shown in Fig 3.5.



Fig 3.5: Inheritance structure of general block

3.4.3 Decimation Block

The decimation block which is defined in GNU Radio as `gr_sync_decimator` is a C++ class. The `gr_sync_decimator` is derived from `gr_sync_block`. Synchronous block has N: 1 input to output ratio with optional history. That is for N input data the number of output data will be 1. The inheritance structure of decimation block is shown in the Fig 3.6.



Fig 3.6: Inheritance structure of decimation block

3.4.4 Interpolation Block

The interpolation block which is defined in GNU Radio as `gr_sync_interpolator` is a C++ class. The `gr_sync_interpolator` is derived from `gr_sync_block`. Synchronous block has 1: N input to output ratio with optional history. That is for 1 input data the number of output data will be N. The inheritance structure of interpolation block is shown in the fig 3.7.

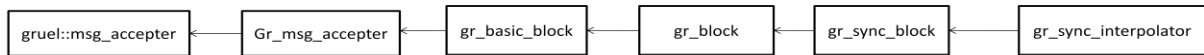


Fig 3.7: Inheritance graph of Interpolator [13]

3.5 Creating a signal processing block in Gnu Radio

The GNU Radio is a collection of signal processing blocks. It is similar to a collection of libraries in C and C++. It is possible to implement a task with the existing signal processing blocks. Sometimes the available signal processing block may not be enough to implement a task. In that case we can edit a signal processing block to our needs or create a new signal processing block. The disadvantage of editing the existing blocks is that actual use of that block will be altered. If that block is used by a person who is unaware of the modification he will be getting an undesirable result. If that block is needed to be used in a different application it will be a problem. Therefore, creating a new signal processing block will provide more flexibility.

There are two ways in creating a signal processing block. One is to use third party scripts such as `gr-modtool` [8]. This method is the easier one as it does not involve lot of editing. Another way to create an out-of-tree block is by editing `gr-howto-write-a-block` which is present in the GNU Radio folder. This method involves lot of editing, especially editing make and cmake files. The person who is interested in editing the cmake files in UNIX can try this method. In this thesis I have used the script from `gr-modtool` to create new signal processing block. The steps involved in creating them are as follows:

Step 1: Download `mbant-gr-modtool-d0fc49e.rar` file from <https://cgran.org/wiki/devtools>

Step 2: Add the `gr-modtool` path of the folder (`home/mbant-gr-modtool-d0fc49e/src`) to the UNIX environment variable.

Step 3: After adding the required folder to environment variable go to GNU Radio folder, in order to create module like `gr-digital`, type command `gr_modtool.py create module_name`

Step 4: A new folder with folder name `gr-module` will be created. In step 3 no need to type `gr-module_name` just by using `module_name` it will create `gr-module_name` folder. Go to the `gr-module` folder.

Step 5: To create the signal processing block, type the command `gr_modtool.py add -t general block-name`, `-t general` will create a block inheriting from general block. It has to be noted that module and blocks are different. A module can have more than one blocks together. To inherit from decimator or interpolator use the command `-t interpolator` and `-t decimator`. The general block, interpolator and decimator block are explained in section 3.4.

Step 6: Go to lib folder. Enter into module_block.cc file. The signal processing function is implemented in general_work. Edit the general_work function. Add extra functions based on your requirement.

Step 7: If you want to implement the GRC for your module go to grc folder and edit the .xml file accordingly. GRC (GNU Radio Companion) is a graphic interface of GNU Radio. You can drag and drop signal processing blocks in GRC. If you want your signal processing block to be working in GRC you need to edit the .xml file accordingly. In this thesis GRC is not used.

Step 8: In order to run the modules go to the gr-module give the command `cmake -i, sudo make`. After `sudo make` some compile time errors may pop up which have to be corrected. Then give `sudo make install` and `sudo ldconfig`. The `sudo ldconfig` has to be used only for the first time. For the subsequent editing or for addition of signal processing block no need to give `sudo ldconfig`.

3.5.1 Connecting signal processing block in C++ with Python

In the section 3.3 the structure of GNU Radio is explained. It is seen that GNU Radio consists of C++ and Python working together. However it is not explained how it is possible to connect C++ and Python. This connection is established by SWIG (Simplified Wrapper and Interface Generator). The SWIG file is written with an .i extension. This function gives the connection between the C++ and Python [19]. The content of the SWIG is taken care by `gr_modtool`.

3.6 System design for transmitter with the GNU Radio

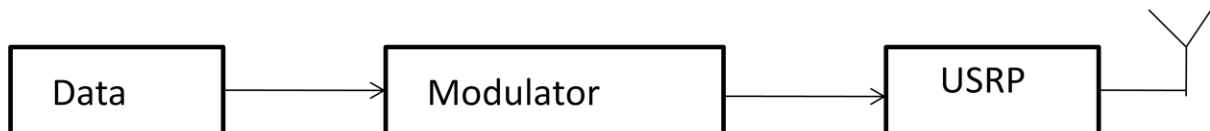


Fig 3.8: Data transmission using GNU Radio and USRP

Let us consider data transmission between two USRP boards without any spreading using GNU Radio. The GNU Radio has a built in module for digital data transmission, in GNU Radio it is present at the `gr-digital` folder. Inside the `gr-digital` folder if we go further inside the subfolders `examples` and `narrowband` we can find a file named `benchmark_tx.py`. By running the `benchmark_tx.py` the message from `benchmark_tx.py` can be sent via USRP.

The `benchmark_tx.py` has the connection link between the `transmitpath.py` and the USRP. The `transmitpath.py` is the Python file present in the same folder as `benchmark_tx.py`. The `transmitpath.py` provides the connection between module `mod_pkt` and amplifier. The `mod_pkt` is a function of `pkt.py` present in Python sub-folder of `gr-digital`. This `mod_pkt.py` provides the connection between the `message_source` and the modulator. Figure 3.9 shows the flow diagram of how data is transmitted using `benchmark_tx` in GNU Radio. The `benchmark_tx` has a flow graph that connects `transmit_path` and USRP. The `transmit_path` has flow graph that connects `mod_pkts` and amplifier. The `mod_pkt` has the flow graph which connects `message_source` and modulator. The message source will take the data from the message queue and send it to the modulator block. The data flows like `benchmark_tx`, `transmit_path`, `pkt` and USRP.

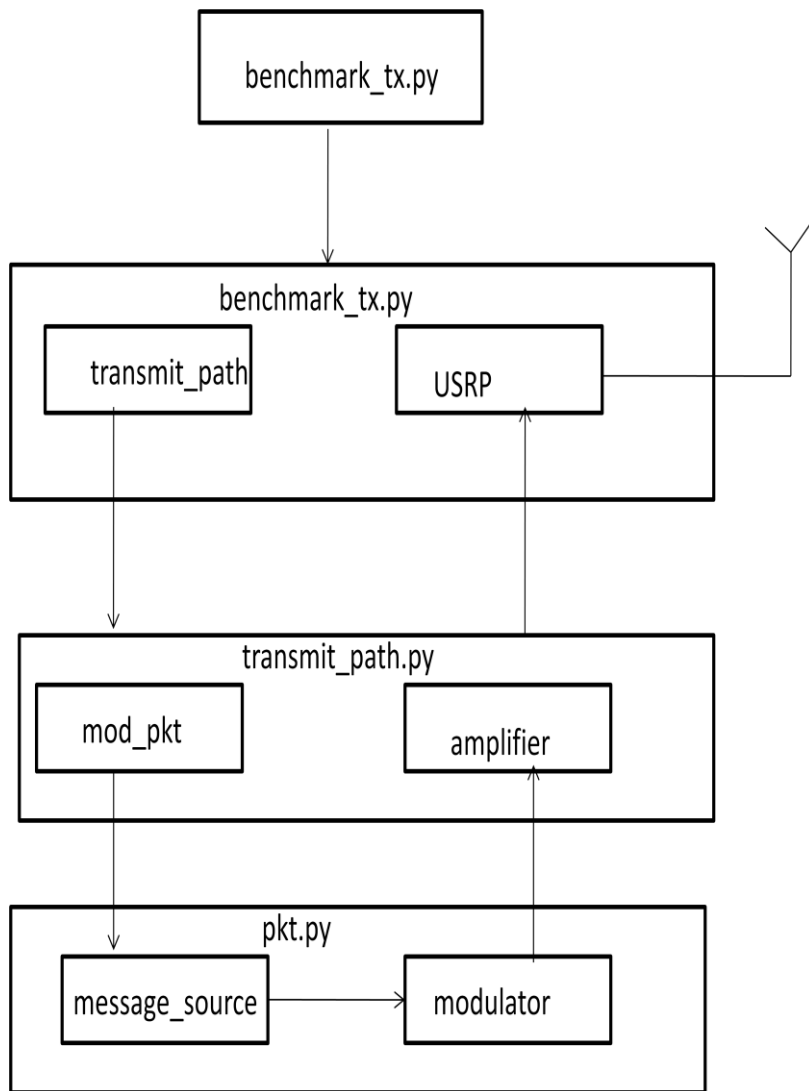


Fig 3.9: Flow graph for the Transmitter in GNU Radio

3.7 System design at the receiver for GNU Radio

The benchmark_rx.py has a connection link between USRP and the receive_path. The receive_path.py has in turn has flow graph connection between fft_filter and demod_pkt. The demod_pkt is found in pkt.py file. The demod_pkt has a flow graph connection between demodulator, correlator_access_code, frame_sink. The flow graph connection is shown in fig 3.10. The correlate_access_code block is used to determine the validity of the packet. This correlate_access_code module matches the received word with the expected one. If the match crosses the expected value it will mark start of a valid packet. The frame_sink will look for the packet header and place it in the message queue which will either be sent to display screen or any other output device.

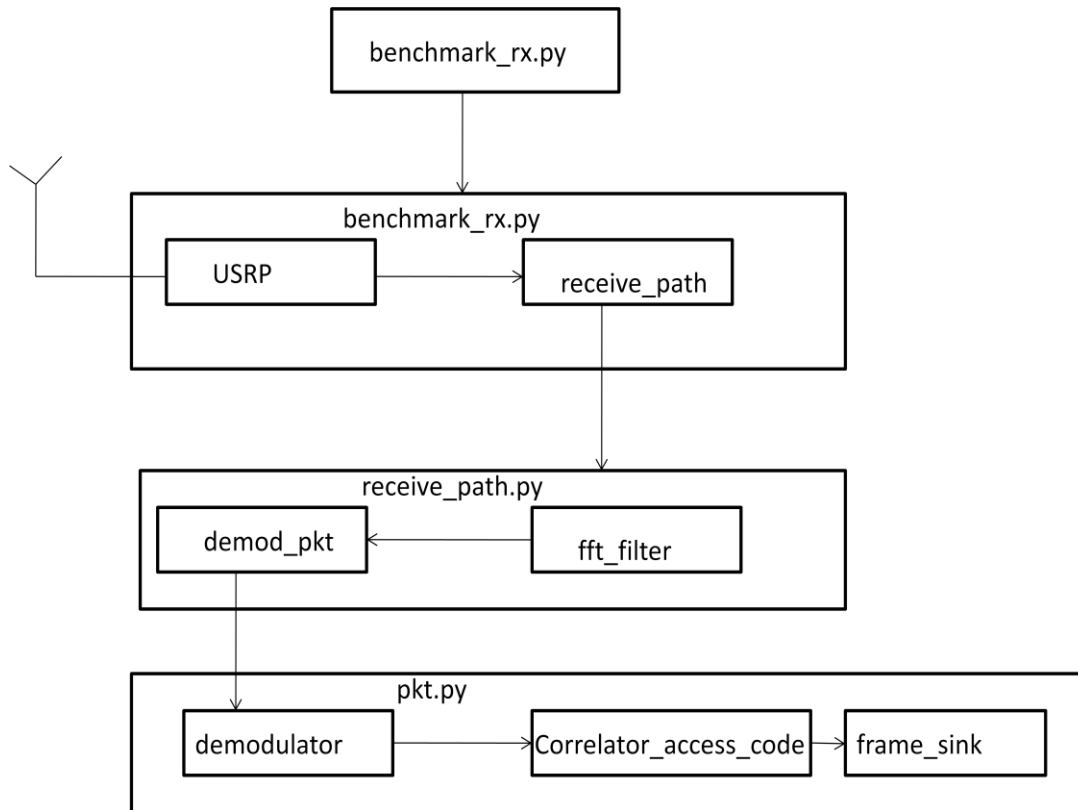


Fig 3.10: Flow graph for receiver in GNU Radio

3.8 Running the transmitter and receiver in GNU Radio

The GNU Radio uses the USRP board for the wireless transmission and reception. The USRP board is connected to the computer and at the computer the GNU Radio is installed. At the transmitter we can run the transmitter by running the following command

```
$ ./benchmark_tx.py -f 2400e6 -r 500e3 -m dbpsk -S 4
```

This command instruct the GNU Radio to do flow graph connection as seen in section 3.6 and data transmission with required signal processing with modulation of DBPSK (Differential Binary Phase Shift Keying), data rate of 500kbts/sec and with 4 samples per symbol. Since the samples per symbol is 4 (Oversampling ratio) and length of the code is 64 each symbol is represented by 256 samples. The USRP board does the necessary DAC and transmission at 2.4GHz. Similarly the command to do basic reception is done by the following command

```
$ ./benchmark_rx.py -f 2400e6 -r 500e3 -m dbpsk -S 4
```

This command instructs USRP board to listen at 2.4GHz. After the reception necessary processing and ADC is done by the USRP board before giving it to the GNU Radio where the digital signal processing done with modulation of DBPSK, data rate of 500kbts/sec and with 4 samples per symbol. Each symbol is represented by 256 samples due to length of code and samples per symbol.

3.9 GNU Radio Tools and utilities

When GNU Radio is installed it will have some utility programs along with it. The utility programs are uhd_fft, uhd_rx_cfile, uhd_rx_nogui, uhd_siggen, gr_plot. The uhd_fft is used when GNU Radio is connected with UHD (USRP Hardware Driver) devices like USRP. It is used to display the spectrum at a given frequency. The uhd_rx_cfile write IQ samples to file which can be analysed

later. The `uhd_rx_nogui` will enable you to listen the incoming signal through your audio device. This tool can also demodulate AM (Amplitude Modulation) and FM (Frequency Modulation) signals. The `gr_plot` is used with the file which stores the received signal. You can plot the signal stored in a file with the time domain representation [6]. The `uhd_siggen` is used to generate common signals such as sine, sweep, square and noise.

Chapter 4 USRP

4.1 Introduction

Matt Ettus developed the USRP board primarily for GNU Radio users, but USRP boards are also used by other software [10]. The USRP board is an integrated board which does ADC, DAC conversions along with it is the RF front-end, FPGA which does some pre processing which are computationally expensive. The USRP board is the best for GNU Radio users. There are different kinds of USRP boards available such as the Network series, Embedded series and Bus series.

4.2 USRP 1

The USRP1 board is equipped with 4 high speed 12 bit ADCs. USRP1 has a sampling rate of 64 MS/s. The Programmable Gain Amplifier is connected before ADC to amplify and to use the entire range of the ADC. At the transmitter side there are 4 DACs with 14 bit. They have sampling rate of 128MS/s. There is also a PGA present after the ADC. USRP1 has the capability of MIMO extensions. The interface used in USRP1 is USB 2.0. USRP1 consists of 2 transmitters and 2 receiver daughter board and one mother board. Fig 4.1 shows the USRP1 with four daughter boards and one mother board. Fig 4.2 shows the general block diagram of USRP1 board.

The mother board has four slots, 2 for transmitter daughter boards and 2 for receiver daughter boards. The daughter boards are used to hold the transmitter interface and the receiver interface. Each transmitter daughter board has the access to the two DACs and each receiver daughter board has access to the two ADCs. This can be clearly seen in the Fig 4.2.

In the USRP1 board all ADCs and DACs are connected to the FPGA. The FPGA plays very important role in USRP1 as it reduces the data rate which can be handled by USB 2.0. At the receiver down conversion is performed using CIC (Cascade Integrator Comb), 4 digital down converter are implemented in FPGA. Similarly the FPGA implements a digital up converter at the transmitter side. USRP1 is the first USRP board designed.

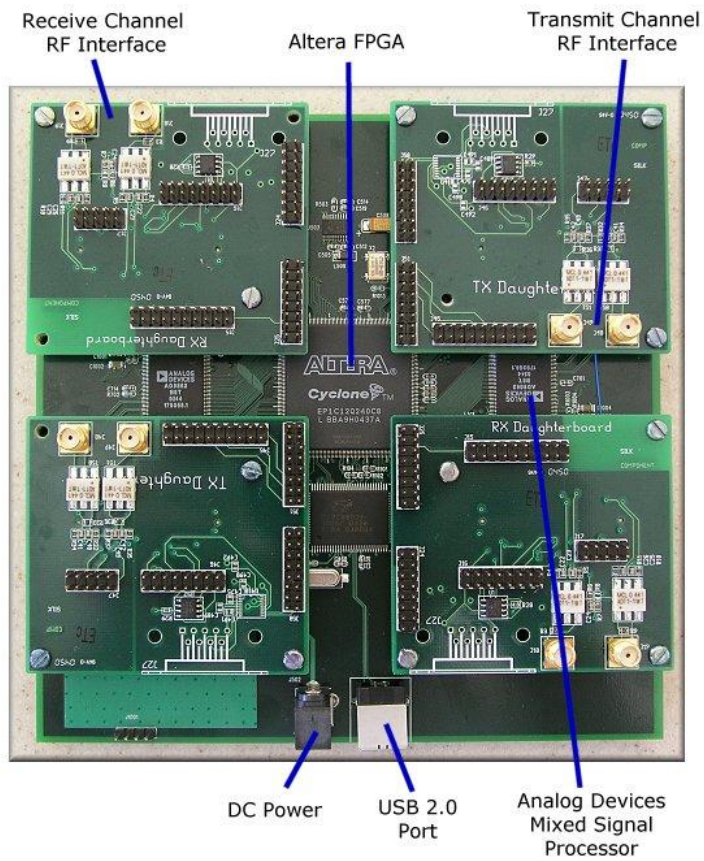


Fig 4.1: USRP1 with Mother board and Two daughter boards for transmitter and receiver [8]

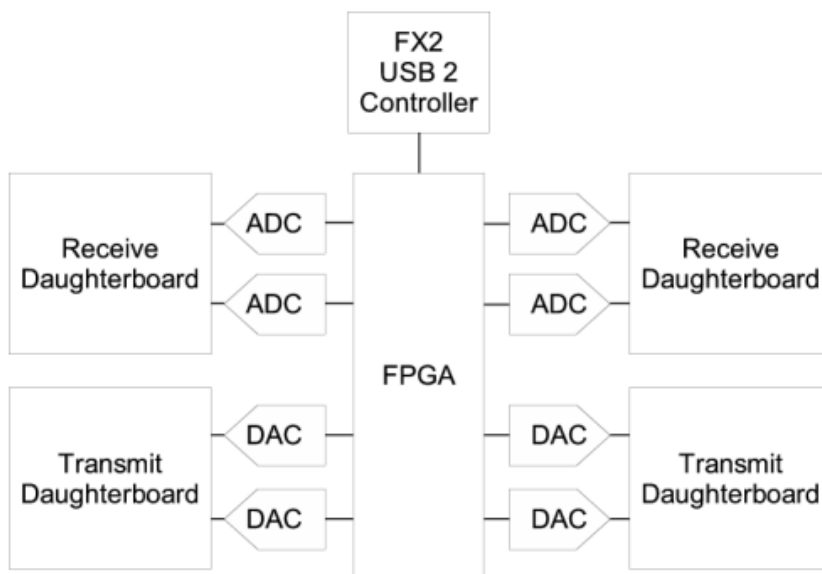


Fig 4.2: USRP1 Block Diagram [8]

4.3 USRP2

With the success of USRP1, the USRP2 board has been developed. USRP2 has high speed ADCs, DACs and wide band processing compared to USRP1. It uses Giga Bit Ethernet as shown in Fig 4.3 compared to USB 2.0 used by USRP1. USRP2 has 14 bit ADC which samples at 100 MS/s. It has 16 bit DAC which samples at 400MS/s. As shown in Fig 4.3 it has a slot for an SD card. It has full coherent MIMO expansion. It has 1Megabyte on board SRAM (Static Random Access).

In USRP 2 high speed sampling rate processing such as DDC, DUC takes place in the FPGA. Lower sampling rate can be performed both at the FPGA and at the receiving host computer. The larger FPGA allows USRP2 to be a standalone system without connecting it to the host system. USRP2 firmware is stored in the SD card which allows programming without need of any special hardware. It is possible to program DDC and DUC with different decimation and interpolation rates. As the FPGA handles high sampling rate processing it has two DDC which decimate the incoming signal. Similarly the FPGA has two DUC which interpolate the outgoing signal [20].



Fig 4.3: USRP2 board front view

4.4 USRP Network Series

USRP Network series N200 and N210 are two high performing USRP board families. This Network series is best suited for applications with high RF streaming including dynamic spectrum access advanced physical layer design, prototyping. The USRP Network series comes with high resolution ADC and DAC for demanding applications and also offer higher bandwidth and dynamic range capability. As with other USRP boards seen in section 4.2 and 4.3 it has MIMO capabilities [5].

4.4.1 USRPN200

As explained in Section 4.4 USRPN200 is a member of the high performance Networked series. USRPN200 has high performance and high dynamic range. The device is equipped with 3A DSP 1800 FPGA, 100 MS/s dual ADC, 400MS/s dual DAC and Ethernet connectivity. The device also has optional GPDSO to have the reference clock to be within 0.1ppm of worldwide GPS standard. The FPGA can be reloaded through the Gigabit Ethernet. The FPGA can process 100MHz of RF bandwidth in both transmit and receive paths. Fig 4.4 shows the internal view of USRPN200 with its daughterboard. Table 4.1 shows the comparison of USRP 1, USRP 2 and USRPN200 devices.

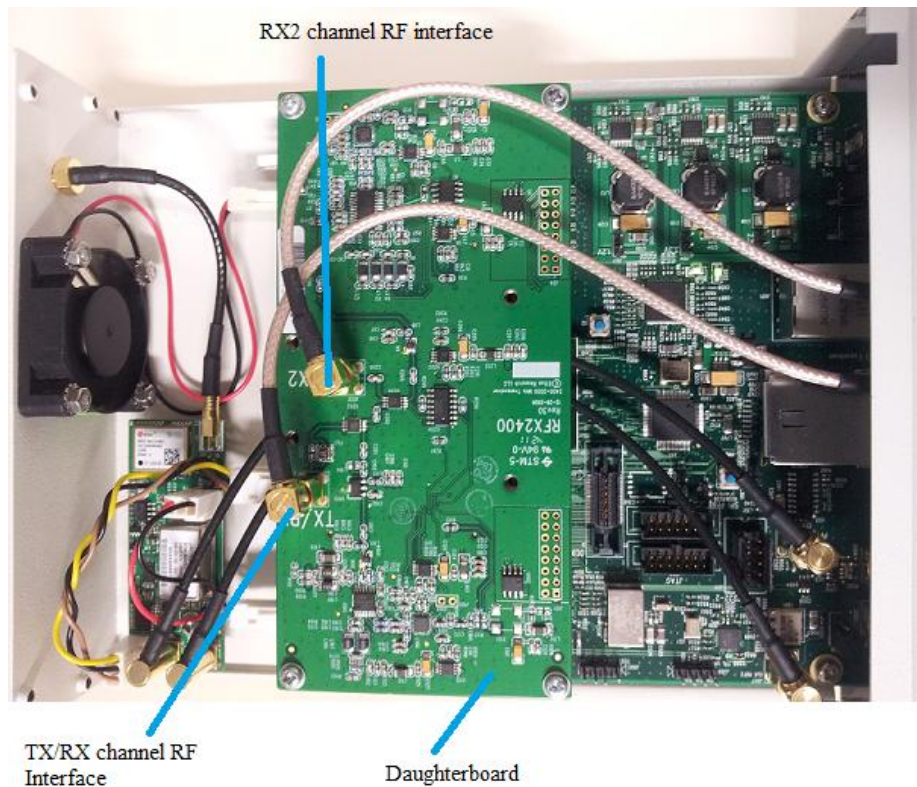


Fig 4.4: USRPN200 with daughter board

4.4.2 USRPN210

As explained in section 4.4 USRPN210 is another member of high performance networked series. USRPN210 has high performance and high dynamic range. The device is equipped with 3A DSP 3400 FPGA, 100MS/s dual DAC, 400MS/s dual ADC and Gigabit Ethernet connectivity to stream data to and from the host processor. USRPN210 has the optional GPDSO to have the reference clock to be within the 0.1ppm of worldwide GPS standard. The FPGA can be reloaded through the Gigabit Ethernet. The FPGA can process 100MS/s in both transmit and receive direction. The USRPN210 has larger FPGA compared to USRPN200. Fig 4.5 and 4.6 shows the internal view of USRPN210 with GPS module, daughterboard, FPGA and other board.

4.5 Daughterboard RFX2400



Fig 4.7: Daughterboard RFX2400

As seen in the section 4.4 USRP Networked series also has two components the Mother board and the daughter board. The mother board consists of ADC, DAC, FPGA, and host processor interface and power regulator. This mother board has more to do with the baseband signal processing. The daughter board is used in the digital operation such as DUC and DDC, filtering and other signal conditioning.

The daughter board RFX 2400 is a high performance full duplex transceiver that operates at 2.4GHz. This transceiver daughterboard has superior performance at 2.4 and 2.483GHz [24]. It has a power output of 50mW and noise factor of 8dB. The daughter board has an RX2 port that provides full range access to 2.3 to 2.9GHz. This daughter board is used in this experimental setup. Figure 4.7 show the picture of RFX2400.

Properties	USRP 1	USRP 2	USRP N200	USRP N210
Host Interface	USB 2.0	Giga Bit Ethernet	Giga Bit Ethernet	Giga Bit Ethernet
DAC	Dual 14 Bit 128 MS/s	Dual 16 Bit, 400MS/s	Dual 16 Bit, 400MS/s	Dual 16 Bit, 400MS/s
ADC	Dual 12 Bit 64 MS/s	Dual 14 Bit, 100MS/s	Dual 14 Bit, 100MS/s	Dual 14 Bit, 100MS/s
Host Band width	16 MHz	50MHz	50MHz	50MHz
FPGA	EP1C12Q240C8	A Xilinx Spartan 3-2000	A Xilinx Spartan-3A DSP 1800	A Xilinx Spartan-3A DSP 3400

Table 4.1: Comparisons of different USRP boards

4.6 GPS module

For multi user system to get good results all the USRP has to be clock synchronized. In other words all USRP takes a common clock from GPS. Mother board of the USRP is used to recognize whether USRP has synced with GPS. It is possible to get the GPS time from the motherboard sensor. In order to get and display the information programmatically we should get the information from `uhd_usrp_sink.cc` file present in `home/gnuradio/gr-uhd/lib` folder. We need to add the following code

```
1. uhd::sensor_value_t val1=_dev->get_mboard_sensor("gps_locked");
2. std::cout<<"\n\n"<<val1.to_pp_string () <<"\n\n";
3. uhd::sensor_value_t val=_dev->get_mboard_sensor("gps_time");
4. std::cout<<"\n\n"<<val.to_pp_string()<<"\n\n";
```

The first line of the code gets the GPS locked status. It gives the information whether the GPS is locked or unlocked. The second line prints the information. The third line gets the GPS time. The fourth line displays the GPS time.

Chapter 5 System Overview

5.1 System specification

Test Bed	GNURadio/USRP2
Carrier Frequency	2.4GHz
Modulation	DBPSK/Single-Carrier
Data-Rate	500Kbps (Kilo bits per second)
Chip-rate	3200kbps
Number of Users	2
RRC filter Roll-off Factor	0.35
Spreading Code	ZCZ code

Table 5.1: System specification for the test bed

Table 5.1 gives the specification used in the test bed. The test bed uses the wireless channel and needs to be transmitted at a particular carrier frequency. The carrier frequency used for the test bed is 2.4GHz. The modulation technique used for the test bed is DBPSK. The advantage of using BPSK is that it gives better performance at high noise levels. The data rate used is 500Kbps. In this test bed code length used is 64, so 64 chips corresponds to 1 symbol and the chip rate is $64 \times 500 = 320000$ Kchips/s. This test bed uses two users with two unique ZCZ codes.

5.2 System design with USRP Hardware and GNU Radio

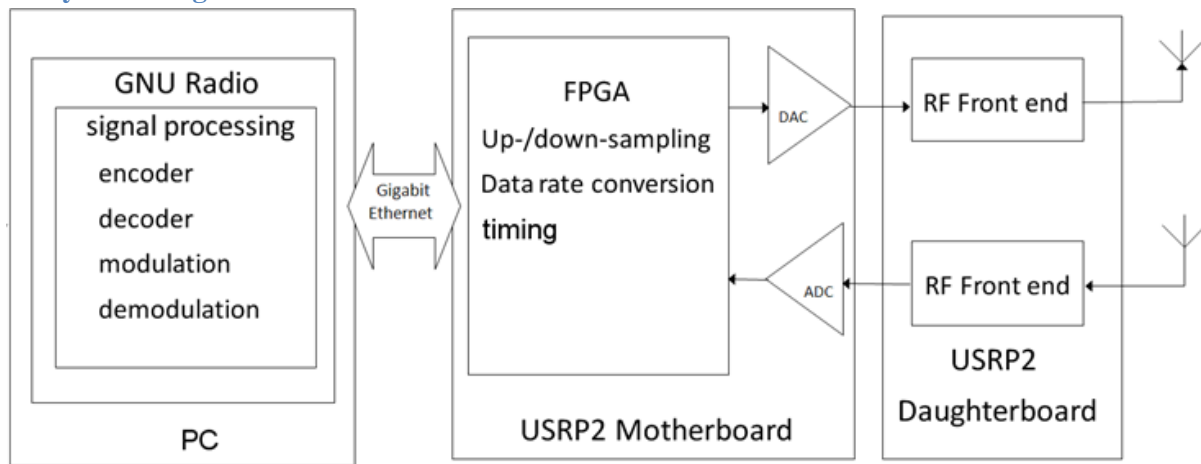


Fig 5.1: System design at USRP hardware and GNU Radio

The GNU Radio and the USRP board are the hardware and software used in this project. The carrier modulation and demodulation is carried out by the USRP board. Signal processing operation such as encoding, decoding, modulation, demodulation is done by the GNU Radio. In the test bed data to be transmitted is first differentially encoded. The differentially encoded data is then spreaded using ZCZ codes. The data spread using ZCZ codes are passed to an RRC (Root Raised Cosine) filter. In RRC filter over sampling and pulse shaping is performed. This baseband signal is then passed to the USRP board where the DAC and RF front end are used to transmit data.

At the receiver the signal is received by the RF front end and analog to digital conversion is carried out which results in the baseband signal. This baseband signal is passed to the RRC filter for down-sampling and pulse shaping. The down-sampled data is then passed to the despread module

present in the demodulation signal processing block where code synchronization and decoding is performed. Then differential decoding is performed to obtain the data. Detailed explanation of data flow is given in chapter 7.

5.3 Differential encoding and Spreading at Transmitter

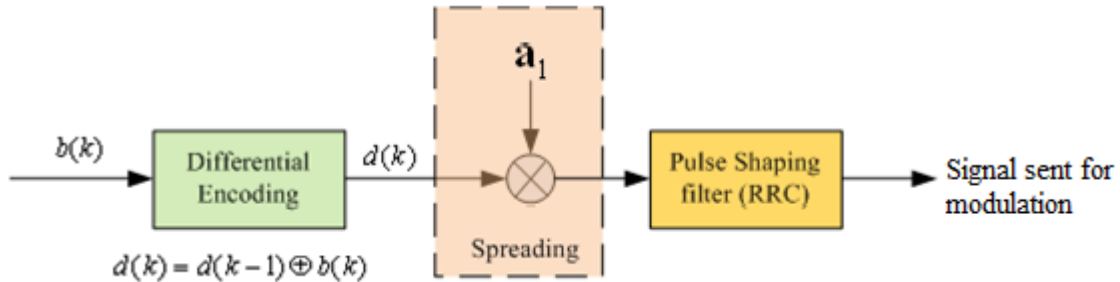


Fig 5.2: Differential Encoding and Spreading at transmitter

The fig 5.2 represents the transmitter design at the sender for differential encoding and spreading. In the fig 5.2 $b(k)$ represent the data to be transmitted. The data to be transmitted is differentially encoded as

$$d(k) = d(k-1) \oplus b(k)$$

The differentially encoded data is then spreaded using the spreading code a_1 . The spreading code used in this system is a ZCZ code with length 64. The spread data is then passed to the RRC filter for pulse shaping and then the signal is passed to the modulator. The modulated baseband signal is then passed to USRP board where DUC operation is carried out at daughter board, and the digital to analog conversion happens at the mother board. After this, the signal is converted to the required frequency, which thereafter is presented to the antenna for wireless transmission.

5.4 Differential decoding and despreading at receiver

Fig 5.3 shows the receiver function for decoding and despreading. The $r(t)$ is the received baseband signal from USRP board. The code acquisition is done to identify start of a symbol, code acquisition has to be done by the receiver. Once the start of the symbol is identified the receiver can use the code a_1 as used at the sender to do the despreading. The despreading is done by multiplying the received data with the code of the particular user. The multiplied data is integrated and sent to the differential decoder. In the differential decoder differential operation and decoding is performed to identify the data.

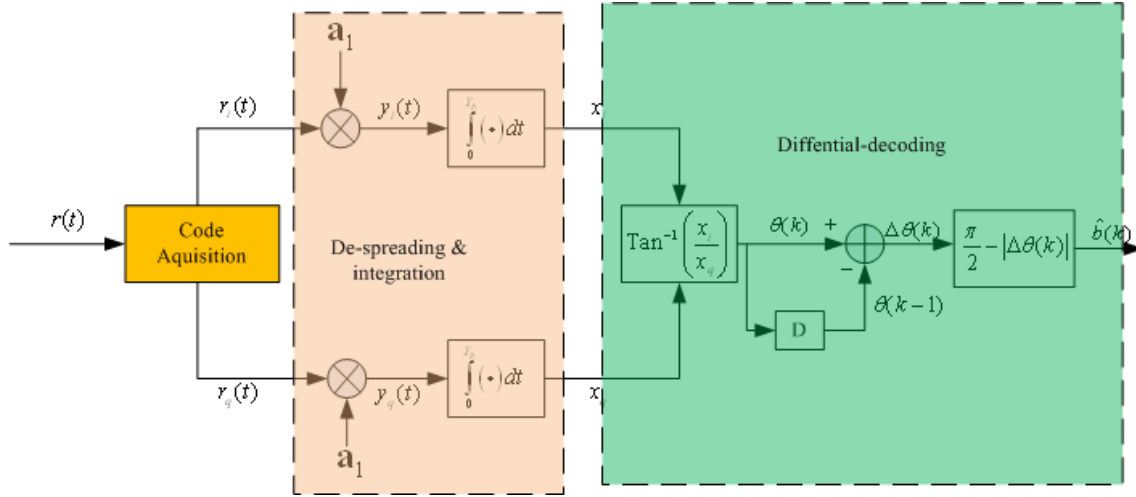


Fig 5.3: Differential decoding and despreading at the receiver

5.5 Code Acquisition of the System

Code acquisition is the procedure use to obtain the starting of a symbol. Based on the code acquisition position decoding of the symbol is performed. In our CDMA system each symbol is spreaded across code length 64. Due to oversampling of 4 each symbol is represented by 256 samples. Start of the code mark the start of the symbol. First data are collected randomly from the signals of length W symbols. If $r(t)$ is the collected symbol where $0 \leq t \leq 64 \cdot 4 \cdot W - 1$. If $a_1(j)$ is the spreading code for user 1 the function $f(t)$ can be written as

$$f(t) = \sum_{i=0}^{W-2} \left| \sum_{j=0}^{64-1} r(256i + 4j + t) a_1(j) \right|^2$$

where

$$0 < t < 255$$

The received data can be written as

$$r(t) = \sqrt{p_1} h_1 d_1(t) a_1(t) w(t) + \sqrt{p_2} h_2 d_2(t - \tau) w(t - \tau) + n(t)$$

where

p_1 = User 1 transmitted power

h_1 = User 1 Channel gain

d_1 = BPSK symbol of User 1

a_1 = Spreading code of User 1

$w(t)$ = shaping waveform

$n(t)$ = noise

p_2 = User 2 transmitted power

h_2 = User 2 Channel gain

d_2 =BPSK symbol of User 2

a_2 =Spreading code of User 2

Then the function $f(t)$ needed to be searched for the peak value to obtain the starting position of the symbol. In the Fig 5.3 creation of function $f(t)$ and searching for peak are done inside code aquisition box. After the starting position of the symbol is obtained, decoding can be performed. This is a simple code aquisition algorithm based on ZCZ auto correlation property explained in section 5.7. Since the auto correlation of ZCZ code has a unique peak at $t=0$ as compared to other positions of t it is possible to uniquely identify the start position by correlating the received signal with the same transmitted code. Even for the multi user scenario where the two users start within the ZCZ they will have zero interference and it is possible to get a unique peak. It can be seen from the Fig 5.4 there is a unique peak which represent the starting position of the symbol. Since the symbol length is 256, start position should be in one of the first 256 positions of the collected data. From the start position of the symbol we can start the despreading of the symbol by multiplying the incoming data with the code. The graph in fig 5.4 is for the idle system designed at Matlab. The constellation diagram after despreading for the ideal system is given in Fig 5.5.

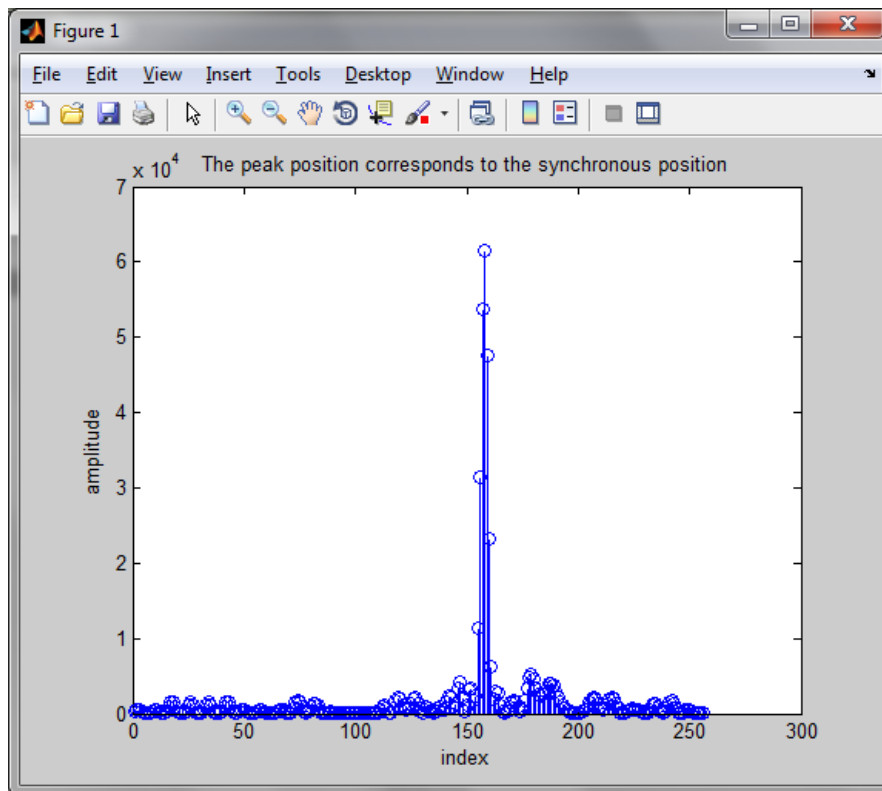


Fig 5.4: Peak position corresponding to the code aquisition

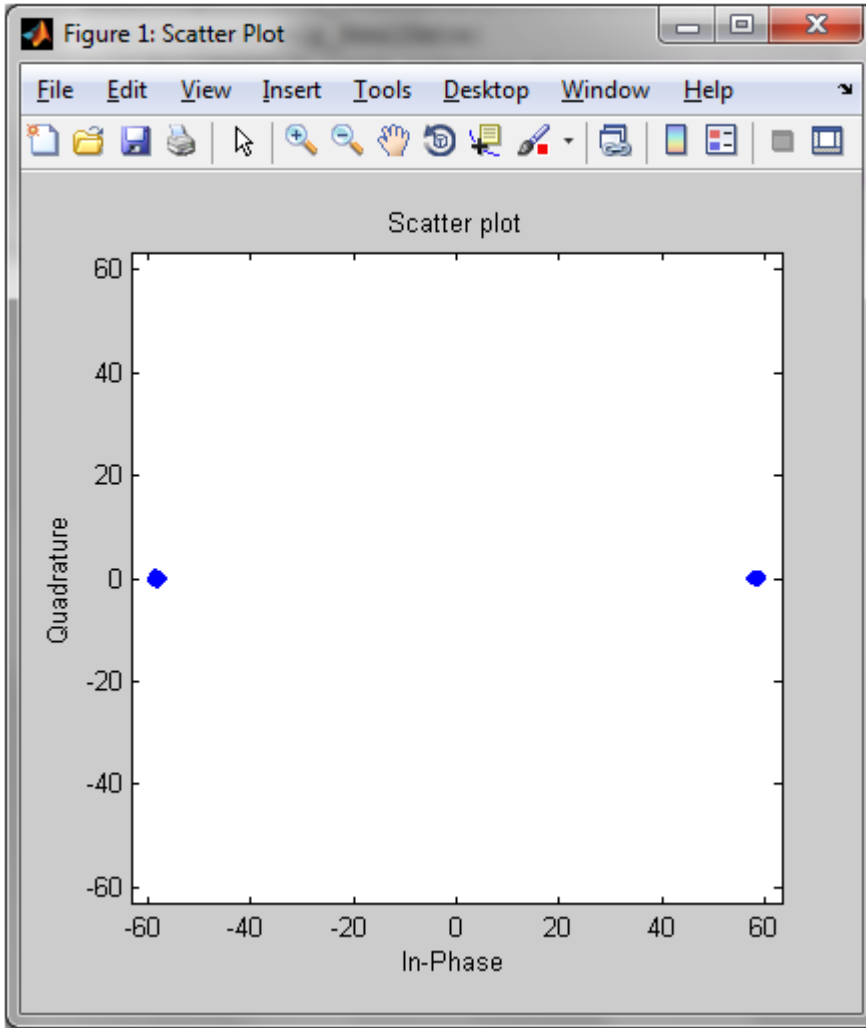


Fig 5.5: Constellation diagram for ideal system after despreading

5.6 Code Tracking

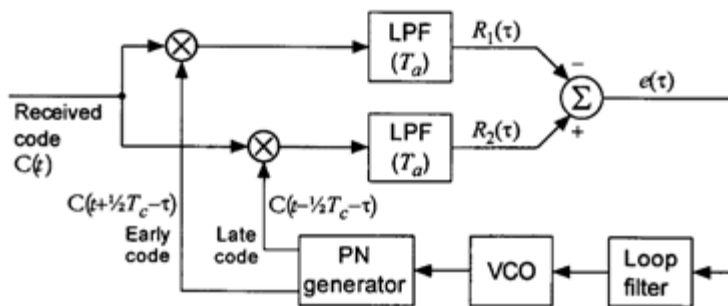


Fig 5.6: Code Tracking of the Code acquisition position [2]

As seen in section 5.5 start position of a symbol is very important as it gives a position in the received signal where to start the despreading. If we wrongly obtain the Code Acquisition position the

entire despreading will go wrong. Similarly if the code acquisition position slips away the entire code despreading after that will go wrong and hence there should be mechanism to check the acquisition position after the initial acquisition algorithm, which is done by the code tracking algorithm.

Fig 5.6 shows the diagrammatic representation of code tracking. Code tracking maintains two correlators the early correlator and the late correlator. An early correlator has code reference wave form that is advance by fraction of chip compared to the current code phase. Similarly the late correlator has code reference waveform that is delayed by fraction of chip compared to the current code phase. The difference between the early and late correlator is used to sense the deviation of acquisition position. From fig 5.6 it could be noted that input of low pass filter is multiplied by the values of early code and late code to get $R_1(\tau)$ and $R_2(\tau)$ respectively. Then an error function $e(\tau)$ which is obtained by the equation

$$e(\tau) = R_2(\tau) - R_1(\tau)$$

This $e(\tau)$ value will be used by the VCO to drive the PN generator to synchronize with correct code acquisition position.

5.7 ZCZ property

We saw briefly about the ZCZ code in chapter 1. ZCZ code will have a zone where there will be ideal impulsive auto-correlation and zero cross correlation functions. Z_{ACZ} is a zone where all the amplitude of auto-correlation function will be zero except at $t=0$ and outside the zone there will be non-zero amplitude. Similarly Z_{CCZ} is a zone where there is zero amplitude for cross-correlation at particular zone and non-zero amplitude outside the zone [27].

Z_{CCZ} means inside the zone two codes will have zero interference. That is within the zone the two users can transmit their signal without any interference. This gives the flexibility for users to have different start time that fall within the zone to have zero interference. This is applicable to more than two users. Three codes can be used to separate three different users, four codes can be used separate four users and so on. ZCZ code is used in this project. The entire project is based on ZCZ property and how well the system performs for ZCZ codes.

5.7.1 Auto correlation property of User 1 code

Let us look into the auto correlation property of code used by user 1 in this project. The ZCZ code is a ternary code that has -1, 1 and 0 as its value. The code1 is as follows

code1=[1, 1, -1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1, -1, -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Fig 5.7 shows the auto correlation property of code 1. As stated in the auto correlation property of ZCZ code it has impulsive auto correlation where there is a unique peak at $t=0$, zero at the remainder of the zone and non-zero value outside the zone. This auto correlation property is mainly used in code acquisition as explained in section 5.5. Due to the auto correlation property of unique peak, the start of the code is identified. Though there is non-zero value outside of the zone the peak value is substantially higher than the other non-zero values.

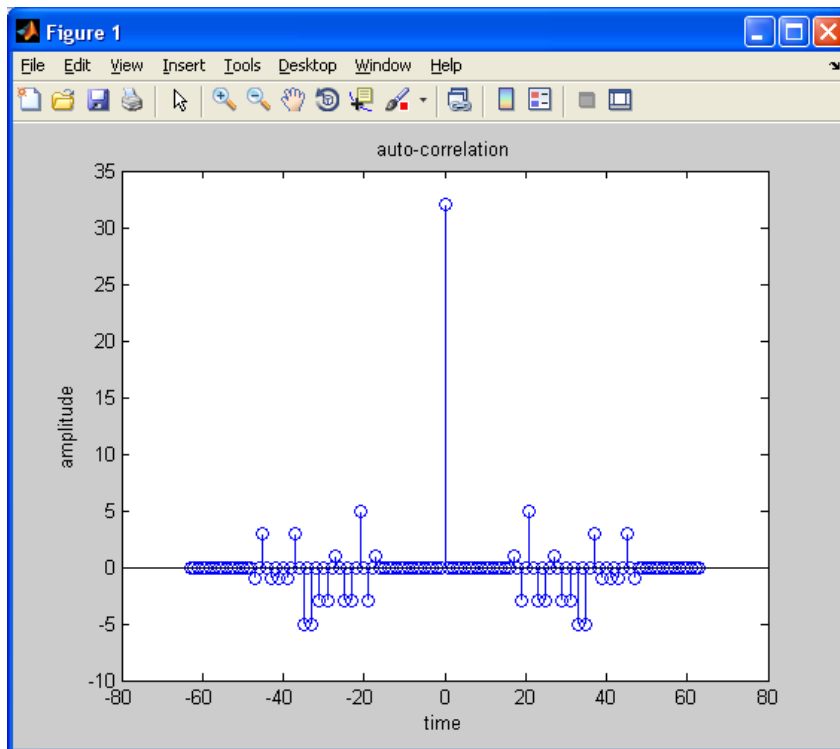


Fig 5.7: Auto correlation property of code 1

5.7.2 Auto correlation property of User 2 code

Let us look into the auto correlation property of code used by user 2 in this project. The ZCZ code for user2 is also ternary code that has -1, 1, and 0 as its value. The code2 is as follows

code2=[-1, 1, 1, -1, 1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1, 1, 1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Fig 5.8 shows the auto correlation property of code 2. As stated in the auto correlation property of ZCZ code it has impulsive auto correlation where there is a unique peak at $t=0$, zero at the remainder of the zone and non-zero value outside the zone. This auto correlation property is used in code acquisition. Due to the auto correlation property of unique peak, the start of the code is identified. Though there is non-zero value outside of the zone the peak value is substantially higher than the other non-zero values.

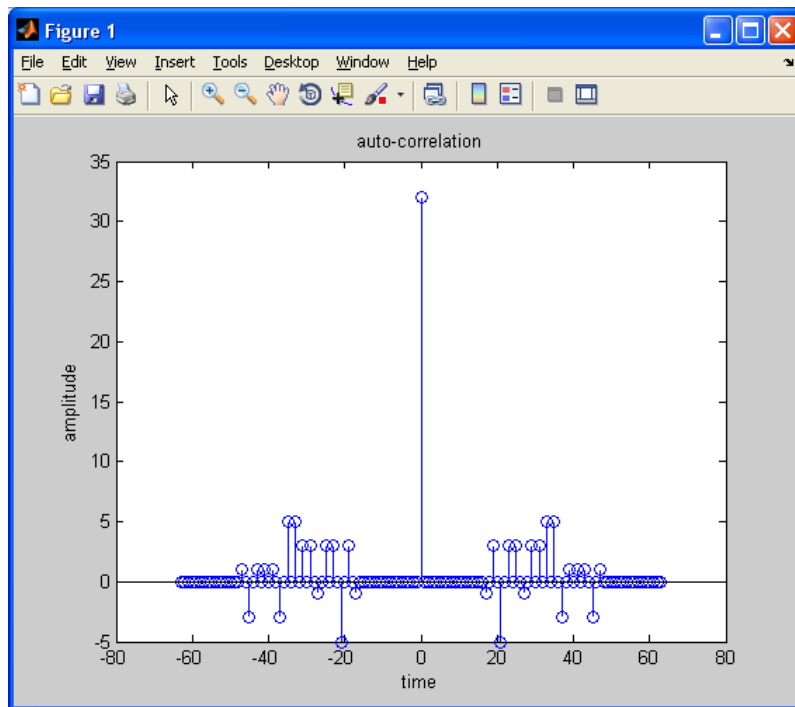


Fig 5.8: Auto correlation property of User 2 code

5.7.3 Cross Correlation Property of User 1 and User 2 code

In order to use codes in CDMA system to differentiate different users the code should exhibit good cross correlation property. That is, the codes should have zero or near zero cross correlation. Let us examine the cross correlation of code 1 and code 2. Fig 5.9 shows the cross correlation property of code1 and code 2. As stated in the cross correlation of ZCZ code, the cross correlation amplitude is zero at ZCZ and non zero value outside it. Inside the zone the two codes will have zero interference but outside the zone the code will have interference. If two users start within the zone then the users can exhibit zero interference. The ZCZ for code 1 and code 2 is 16 samples. If the number of samples between the start of code1 and code 2 is less than 16 then the interference will be zero.

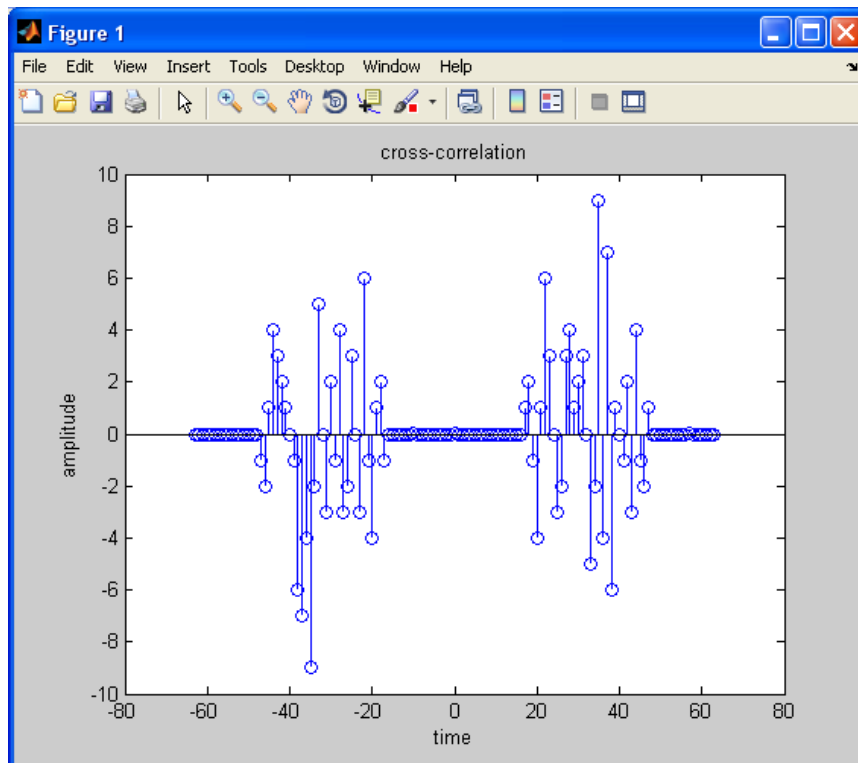


Fig 5.9: Cross correlation of Code 1 and Code 2

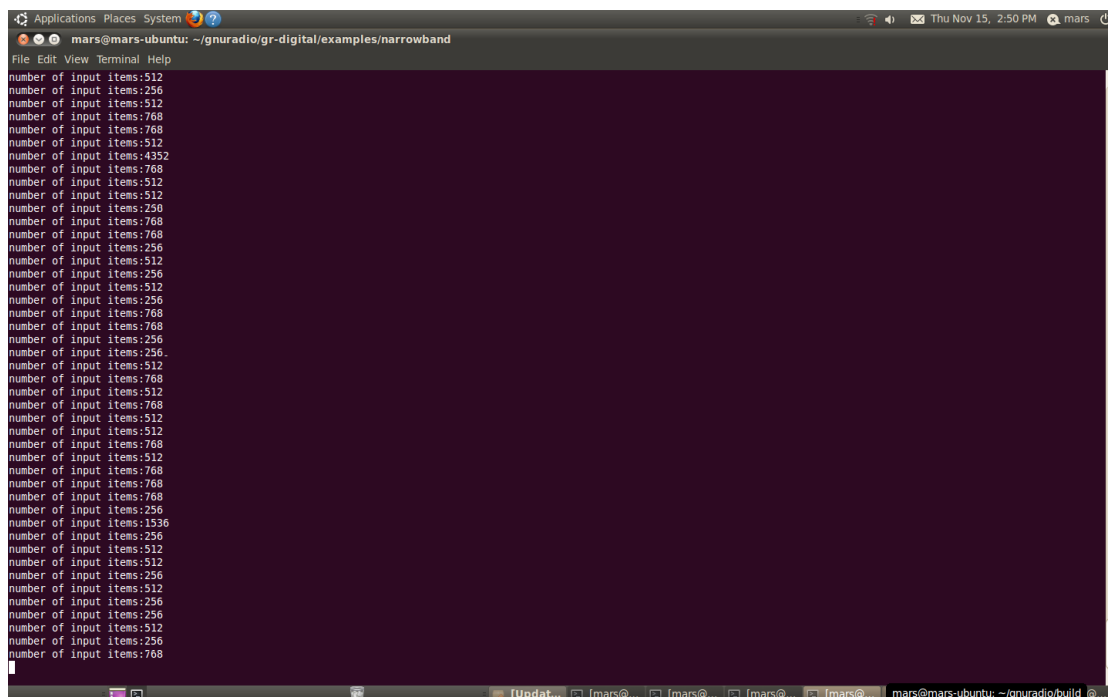
Chapter 6 System implementation problem in GNU Radio and multi user transmission

6.1 Introduction

The idea of a ZCZ code CDMA system can be simulated in Matlab. However the main idea is to test the ZCZ code CDMA system in real-time using GNU Radio and USRP boards. Unlike the Matlab simulation real time data need to be handled carefully. In GNU Radio real-time data is received as a stream of data where the length varies from time to time based the scheduler. There are certain signal processing modules with feedback mechanism such as code tracking where there will be certain restrictions which are needed to be handled. Matlab comes with lot of flexibility and in built functions which are not present in GNU Radio. These functions needed to be implemented manually. In this chapter we will discuss the problem in system implementation discussed in chapter 5 with GNU Radio and the problem of multi-user uplink transmission using ZCZ codes.

6.2 Problem of handling Stream of data

The GNU Radio is a collection of signal processing blocks. For a particular application these blocks need to be arranged in a particular manner. The data flows through these signal processing blocks as a stream of data, where the length of the stream varies from time to time based on the scheduler, so it will be very important to handle it properly. Fig 6.2 represents the stream data flow. Data is represented by a square, the colour of the square changes when it undergoes processing in a signal processing block. Data enters the signal Block 1, undergoes certain processing and enters the signal Block 2 undergoes certain processing and enters next block and so on. Fig 6.1 shows the screen shot for the number of input items that is the length of the stream of data that flows into the signal processing block. The number of inputs to a signal processing block changes for each iteration. Each stream has different length where stream length varies from 256 to 1536 as seen in fig 6.2.



```
Applications Places System
mars@mars-ubuntu: ~/gnuradio/gr-digital/examples/narrowband
File Edit View Terminal Help
number of input items:512
number of input items:256
number of input items:512
number of input items:768
number of input items:768
number of input items:512
number of input items:4352
number of input items:768
number of input items:512
number of input items:512
number of input items:256
number of input items:768
number of input items:768
number of input items:256
number of input items:512
number of input items:256
number of input items:512
number of input items:256
number of input items:768
number of input items:768
number of input items:256
number of input items:512
number of input items:768
number of input items:768
number of input items:512
number of input items:512
number of input items:768
number of input items:768
number of input items:512
number of input items:768
number of input items:768
number of input items:256
number of input items:1536
number of input items:256
number of input items:512
number of input items:512
number of input items:256
number of input items:512
number of input items:256
number of input items:256
number of input items:512
number of input items:256
number of input items:768
```

Fig 6.1: Screen shot for Number of Input items

In certain applications such as CDMA it is very important the sequence of data is maintained otherwise undesirable effects happen. As seen in the section 5.5 once the start of the symbol is known all the data is despread continuously. If there is a change in sequence of data it will affect the code acquisition position. Fig 6.3 explains the importance of maintaining the data flow sequence in CDMA systems. According to Fig 6.3 code marked with number 1 should be multiplied with data with number 1. Missing the sequence will cause misalignment of the code.



Fig 6.2: Stream data Flow in GNU Radio

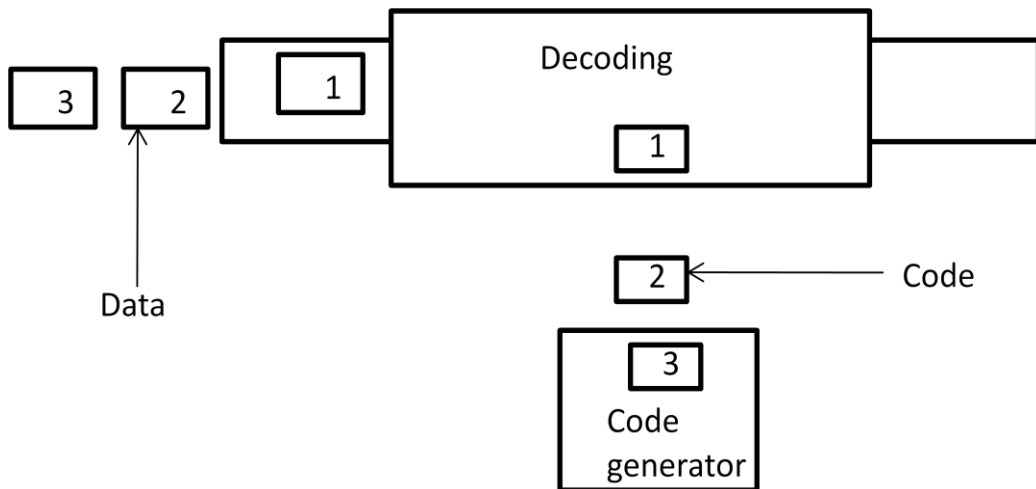


Fig 6.3: CDMA Decoding block data flow with code

6.3 Problem with feedback functions

There are certain block in GNU Radio where based on certain decision the input stream needed to be advanced or delayed, such as implementing code tracking algorithm for CDMA system in GNU Radio. The section 6.2 shows how the data flows in GNU Radio. GNU Radio block with feedback capabilities are bit different from normal block. Unlike the normal block where the stream of flow is not adjusted, for the blocks with feedback capabilities the input stream needed to be adjusted or input data needed to be modified based on certain decision taken in a block for previous input stream. As seen in section 6.2 the number input data stream is not constant so it becomes a challenge in implementing such a block. In GNU Radio it is not possible to pass variables between two or more blocks. It is another challenge in implementing such blocks if feedback operation is spread across different signal processing blocks.

Fig 6.4 shows the data flow diagram for feedback module. It can be seen that based on the decision taken in the signal processing block due to current samples the future input data or data stream is affected. From the fig 6.4 it can be noted that based on current samples 1,2,3,4 the future samples 5,6,7 are affected. As the number of input stream varies feedback implementation needed to be handled properly.

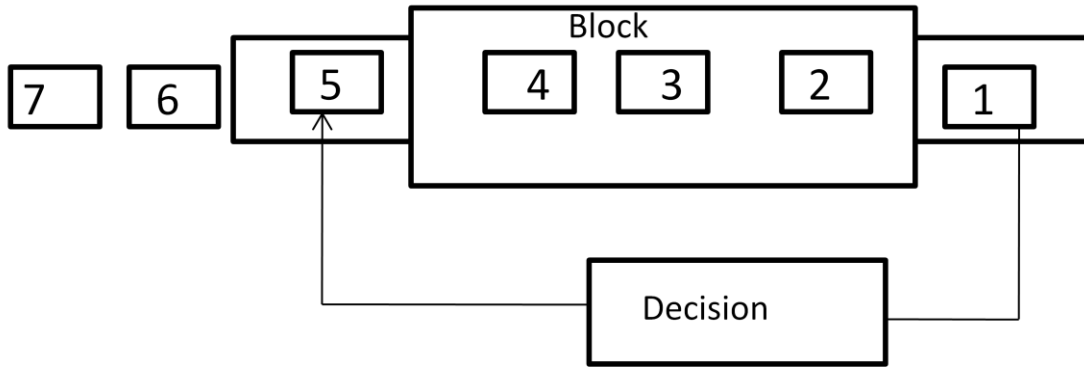


Fig 6.4: Data flow in GNU Radio for Block with loopback

6.4 Implementing the built in functions in Matlab

Matlab comes with lot of in built functions which makes life easier for developing a simulation program. Unfortunately this is not the case for GNU Radio. Though GNU Radio is powered with a lot of in-built signal processing blocks it still come short compared to mathematical functions present in Matlab. GNU Radio uses the C++ in-built standard mathematical functions while Matlab has lot of more flexible functions.

It is easy to copy data between two arrays in Matlab but not in GNU Radio. To copy data between two arrays subject to the restrictions in C++. Let us consider an example kron function in Matlab. If two arrays are passed to Kron function then it returns the kronecker tensor product of the two arrays. This will result in larger array formed by taking all possible combination between the elements of the two arrays. If X is an array with dimension m-by-n and Y is an array with dimension p-by-q, then $\text{kron}(x, y)$ will be an array of length $m \cdot p$ -by- $n \cdot q$ [21]. Such a function is not available in GNU Radio since it is not available in C++.

6.5 Modifying the existing block

In GNU Radio it is possible to implement applications with only the existing blocks. Applications exists which are not possible to implement with only the existing signal processing blocks. It is necessary to create your own blocks or edit the existing blocks. A problem with editing an existing signal processing block is that it will modify the true nature of that signal processing block. If we want to use that signal processing block for different applications it will be a problem. If we are working in a team and if the other members are not aware of your modification then he will get undesirable results.

6.6 Synchronizing different Users in the uplink

In the uplink there will be one receiver and multiple transmitters. This setting is similar to a mobile base station and mobile phones, transmitters can join the system at any time but the receiver should able to decode the message. If two users don't start at the same time the zero correlation property of ZCZ will be violated and each signal will start interfering with each other. We should find a mechanism so that if each user start at different times they should have zero correlation property.

Fig 6.5 shows the code aquisition position for a multiuser system where transmitters have random start times. Clearly due the interference the clear single peak is not obtained. Ideally there should be single peak as in fig 5.4.

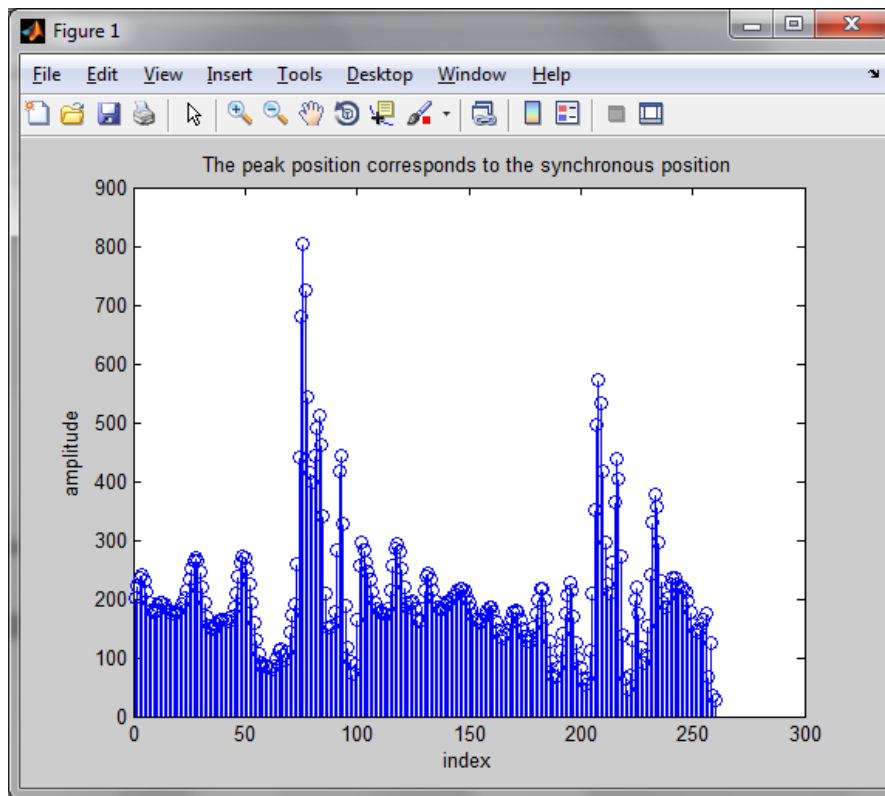


Fig 6.5: Code Aquisition for Multi user without any time control

Chapter 7 System Implementation in Gnu Radio

7.1 System design with GNU Radio

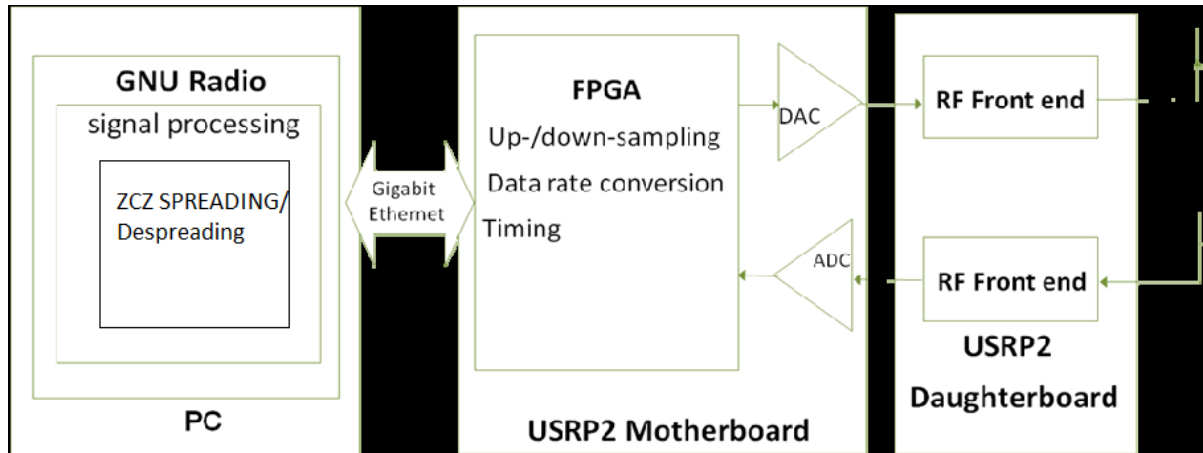


Fig 7.1: System Design with USRP board and GNU Radio

The implementation of ZCZ code for a CDMA system is done using GNU Radio and USRP boards. The USRP board provides ADC/DAC and RF, rest of the communication implementation is done in software. The ZCZ spreading, despreading other modulation and demodulation are implemented in GNU radio as mentioned in Fig 7.1. This project shows the feasibility of an asynchronous multi user communication without strict timing synchronization among the users. Most of the implementation is done in software.

In chapter 8 different lab set ups used in this project will be discussed. The basic spreading and despreading signal processing block created does not change. These spreading and despreading are explained in section 7.4 and 7.5. For the lab set up uplink in section 8.2 the start time of the transmitters is manipulated which will be explained later in this chapter.

7.2 Transmitter without the spreading block

The transmitter without the spreading block is default transmitter without any CDMA code spreading explained in section 3.6. The connection of signal processing block in benchmark_tx.py is also explained in section 3.6. The modulator is considered to be single signal processing block but it is a hierarchical block which means it has set of signal processing blocks but it looks to the outside world as a single block. Let us look into the signal processing block used in modulator and its functionalities.

7.2.1 Modulator without spreading block

The modulator is handled by generic_mod_demod.py file present in the Python folder. Now let us see the connection of different signal processing block inside modulator. The generic_mod_demod.py gives the connection between the pack_to_unpack, if differential modulation is enabled it is connected to differential, chunk_to_symbol and finally to rrc_filter. The packed_to_unpacked will convert stream of packed bytes to stream of unpacked bytes. In the differential module differential encoding is done. The chunks_to_symbol will map the stream of binary data to complex constellation point. The rrc_filter will act as pulse shaping filter. Figure 7.2 shows the flow graph for the modulator. The module for code spreading is added in this flow graph at

the appropriate position which is explained later in the section 7.4. Our system uses DBPSK so the differential encoding will be enabled and the number of constellation points will be 2.

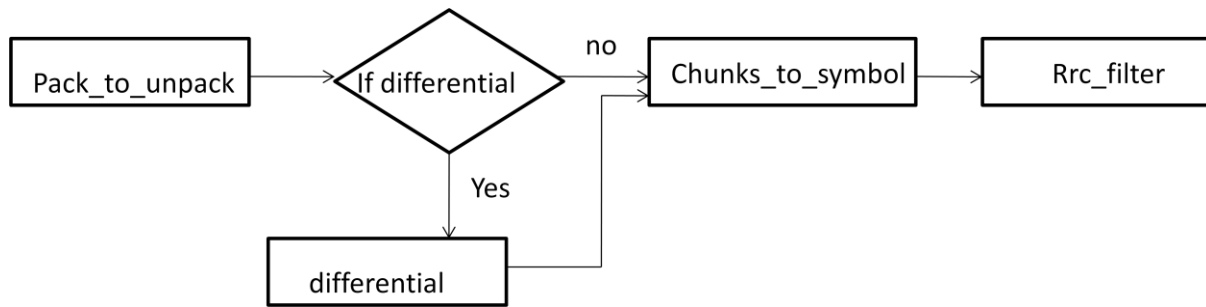


Fig 7.2: Flow graph for modulator

7.3 Receiver without the spreading block

The receiver without the spreading block is default receiver as explained in section 3.7. The connection of signal processing block in benchmark_rx.py is also explained in section 3.7. The demodulator is considered to be single signal processing block but is a hierarchical block which means it has set of signal processing blocks but it look to the outside world as a single block. Let us look into the signal processing blocks used in demodulator and its functionalities.

7.3.1 Demodulator

The demodulator is handled by generic_mod_demod.py file present in the Python folder. Let us see the connection of different signal processing block inside the demodulator. The generic_mod_demod.py gives the connection between the automatic_gain_controller, frequency_recovery, time_recovery, constellation_receiever, if differential modulation is enabled it is connected to differential decoding block, if constellation pre differential code has to be applied then symbol_mapper is added to the flow graph and finally to unpack_k_bits. The automatic gain control is the class for power control. The module for frequency control will do frequency locked loop to control slight frequency shift. The time_recovery module will do time synchronization using the poly-phase filter. The constellation_receiver will make decision on symbol value based on the constellation point used in the system. The differential decoder module will do differential decoding. Then unpack_k_bits will unpack k bit vector into stream of bits. Figure 7.3 shows the flow graph for the modulator. The module for code despreading is added in this flow graph at the appropriate position which is explained later in the section 7.5.

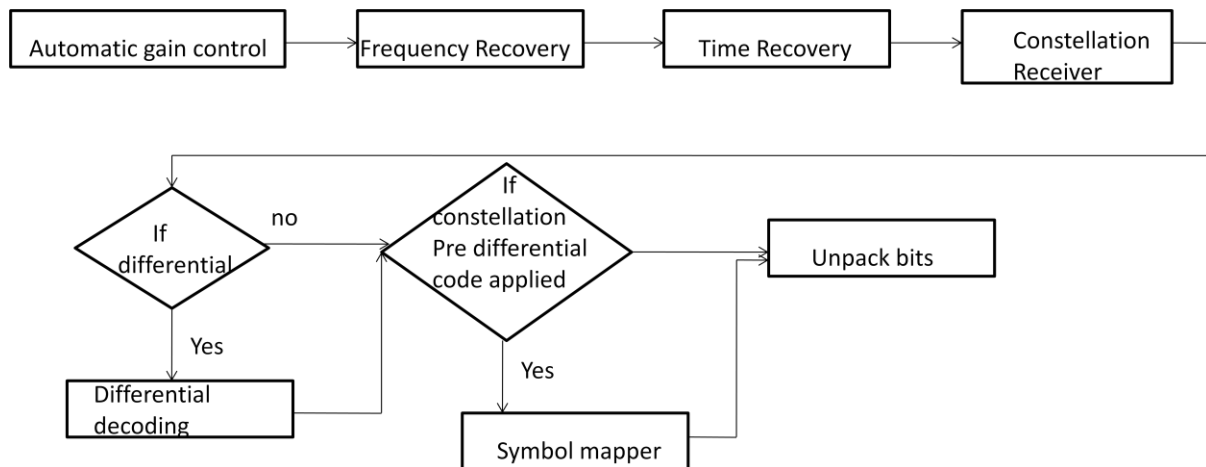


Fig 7.3: Flow graph for demodulator

7.4 System design at sender with spreading

All the components mentioned for the transmitter at section 7.2.0 remains the same for spreading. As mentioned in section 7.2.1 spreading block is added to the existing modulator block. The spreading block is added after the chunks_to_symbol and before rrc_filter block. The rrc filter will do an over sampling by a factor of 4. Each input from the chunks_to_symbol is multiplied by the 64 spreading code. The flow chart for modulator with spreading is given in fig 7.4.

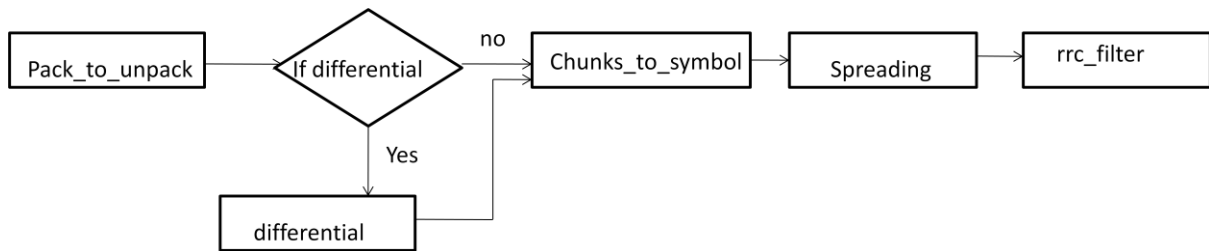


Fig 7.4: Flow graph for modulator with Spreading

7.4.1 Spreading Block

Spreading block is a separate signal processing block created as explained in section 3.5. Spreading block multiplies each incoming data with 64 spreading ZCZ code. The each input is spreaded across 64 code length. Based on the user type of code is also changed. User 1 and user 2 will have different codes. The spreading block is inherited from the interpolator block. The interpolator provides the functionality of 1:N. Where for every one input there will be N output. The function interpolator is explained in detail in section 3.4.4. For the spreading block for every 1 input there will be 64 outputs since the code length is 64. The input and output ratio will be 1:64. The code for the spreading block is given in Appendix A. The flow chart for the code is given in the figure 7.5.

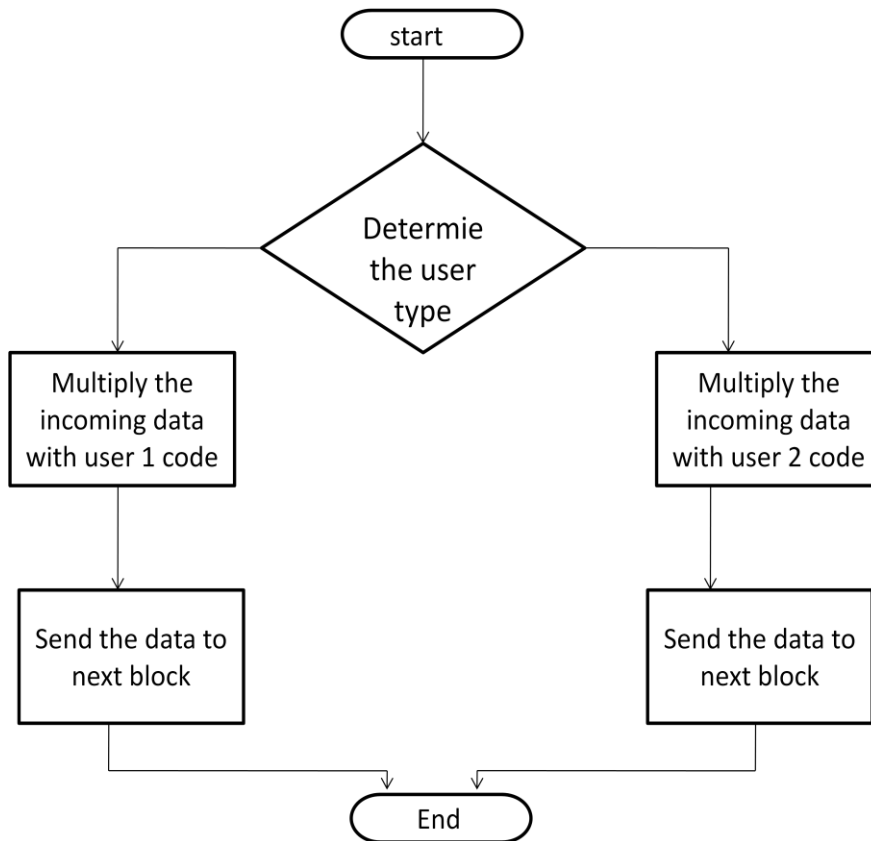


Fig 7.5: Flow chart for the spreading block

7.5 System design at receiver with despreading

As mentioned in section 7.3.1 all the components for the receiver remains the same except demodulator for despreading. In the demodulator a despreading block is added at the appropriate position. The time recovery and frequency recovery module present in the demodulator without spreading is removed. The reason behind removing frequency recovery is that it provides the functionality of FLL (Frequency Locked Loop). The usage FLL before despreading will give undesirable results. The time recovery is replaced with interp_fir filter since interp_fir filter we can control the decimation factor in its first argument as the decimation is done at despreading module. The flow chart for demodulator with despreading is given in fig 7.6.

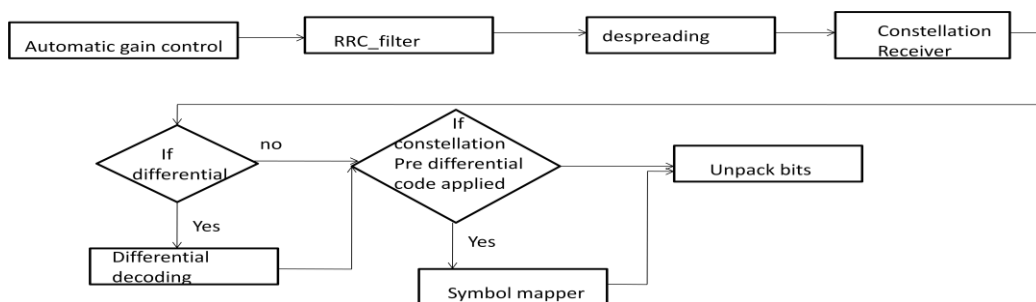


Fig 7.6: Flow chart for code acquisition

7.5.1 CODE Aquisition

Identifying the start of the code is the most important work in CDMA Systems. Improper code Aquisition can make the despreading of code not possible. To identify the start of the code we use the code's auto-correlation property. In section 5.5 the theoretical and mathematical explanation of code aquisition is given. Code aquisition has to be done for the system with tracking or without tracking. Since the start position of the symbol has to be established. For the implementation of the code aquisition a buffer is maintained to collect data. The code aquisition position for the first fully available symbol at the buffer is determined by this algorithm. The C++ code for code synchronization is given in Appendix A. The flow chart for code aquisition is given if fig 7.7. Fig 7.8 gives the graph for code aquisition position which is marked by unique peak for real time data is given. The aquisition position is 29.

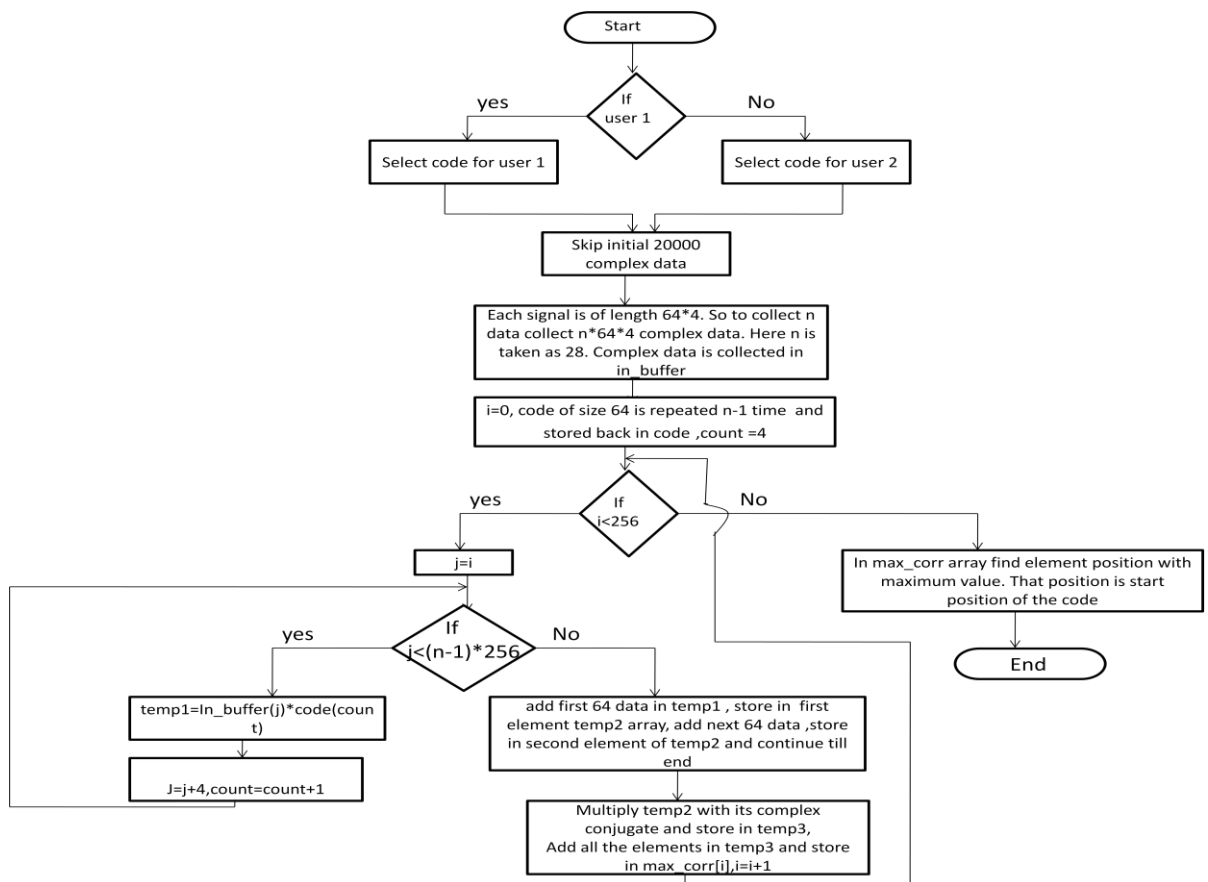


Fig 7.7: Flow chart for code synchronization

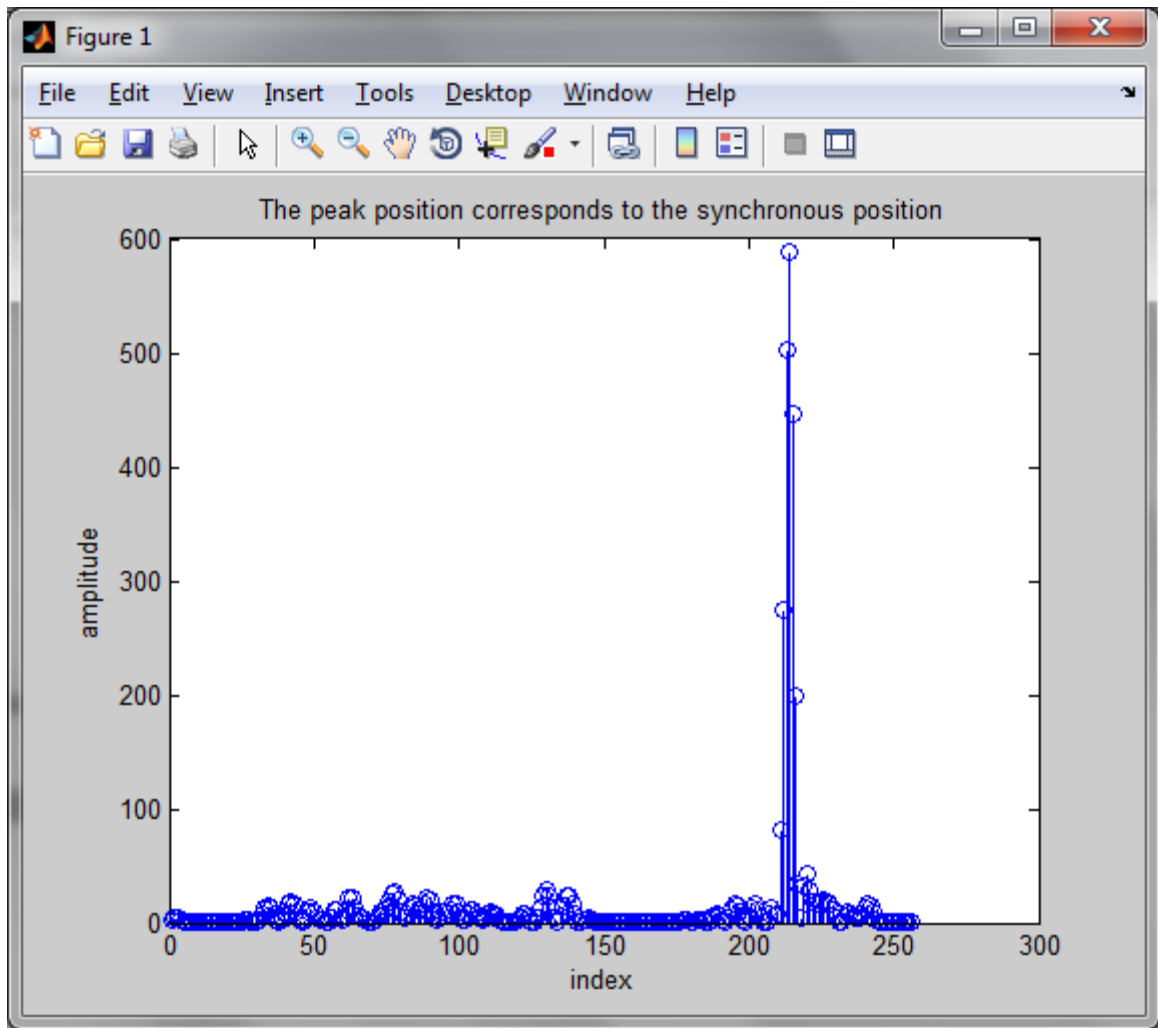


Fig 7.8: Code aquisition for real time data

7.5.2 Code despreading without code Tracking

Despreading block is a separate signal processing block created as explained in section 3.5. The code aquisition function is also present in the same signal processing block as code despreading. At the transmitter, oversampling is done at the RRC filter. Since the code length is 64 and the oversampling ratio is 4, the length of the symbol is $64 \times 4 = 256$. The start position is the start of the first fully available symbol in the buffer from which subsequent start positions of forthcoming symbols can be reached, which is 256 data away from the current aquisition position. GNU Radio has the limitation: chunk of data flow out of a signal processing block is limited in the same manner as chunk of data flow into a signal processing block. By applying the code aquisition algorithm explained in section 7.5.1, start position of the symbol in the buffer is obtained. The buffer size is $n \times 64 \times 4$ where n is the number of symbols. All the data in buffer cannot be processed and some data in the buffer has to be skipped, the start of the next symbol is 256 positions away from the current position. From the current start position, we skip 256 data in the buffer and go to next start position and see whether the end of buffer has reached. The end does not mean the complete end of the buffer, but the end means buffer has less than 256 data, which is the start position of last symbol in the buffer. From the new start position all the data in the buffer are transferred to the sig buffer. The sig data buffer stores the next incoming input data along with the old data. The despreading is done in this sig buffer which holds the data from the start of the symbol. The data in the sig buffer is multiplied with the ZCZ code for despreading, while multiplying a data sample with the code next 4 data samples are skipped to

counter the oversampling at RRC filter. The data multiplied with 64 ZCZ codes are integrated and sent to the next signal processing block. The despreading block inherits from the decimator block, so for every 256 input samples there is only one output. Details about the decimator is explained in 3.4.3. The input to the output ratio (N: 1) for our scenario is 256:1.

From the old data which mark the start position of the symbol, data is despreading and sent to the next block. The number of data sent depends on the particular iteration. Some data from the new iteration will not be sent to the next block due to the backlog of data from previous iteration. For example if initially the sig buffer has 30 data. The first data in the sig buffer marks the starting of the symbol. The next incoming data is a continuation of the symbol. If the allowed number of output data to the next signal processing block is 3, due to decimation the number of input will be $256 \times 3 = 768$. Then prepend the old data to the incoming data. Now start the despreading from the prepended data. Due to the restriction only 768 data can be processed. Due the old 30 data only the first 738 data will be processed and the remaining 30 data coming at the end will be again stored in the sig buffer and used in the next iteration. This process will continue till the end. This method does not change even if the length of chunk of data changes for each incoming chunk data. In this way problem of handling variable number of input data as discussed in section 6.2 can be handled. The C++ code for code despreading is given in appendix A. The flow chart for code despreading without code tracking is given in figure 7.9

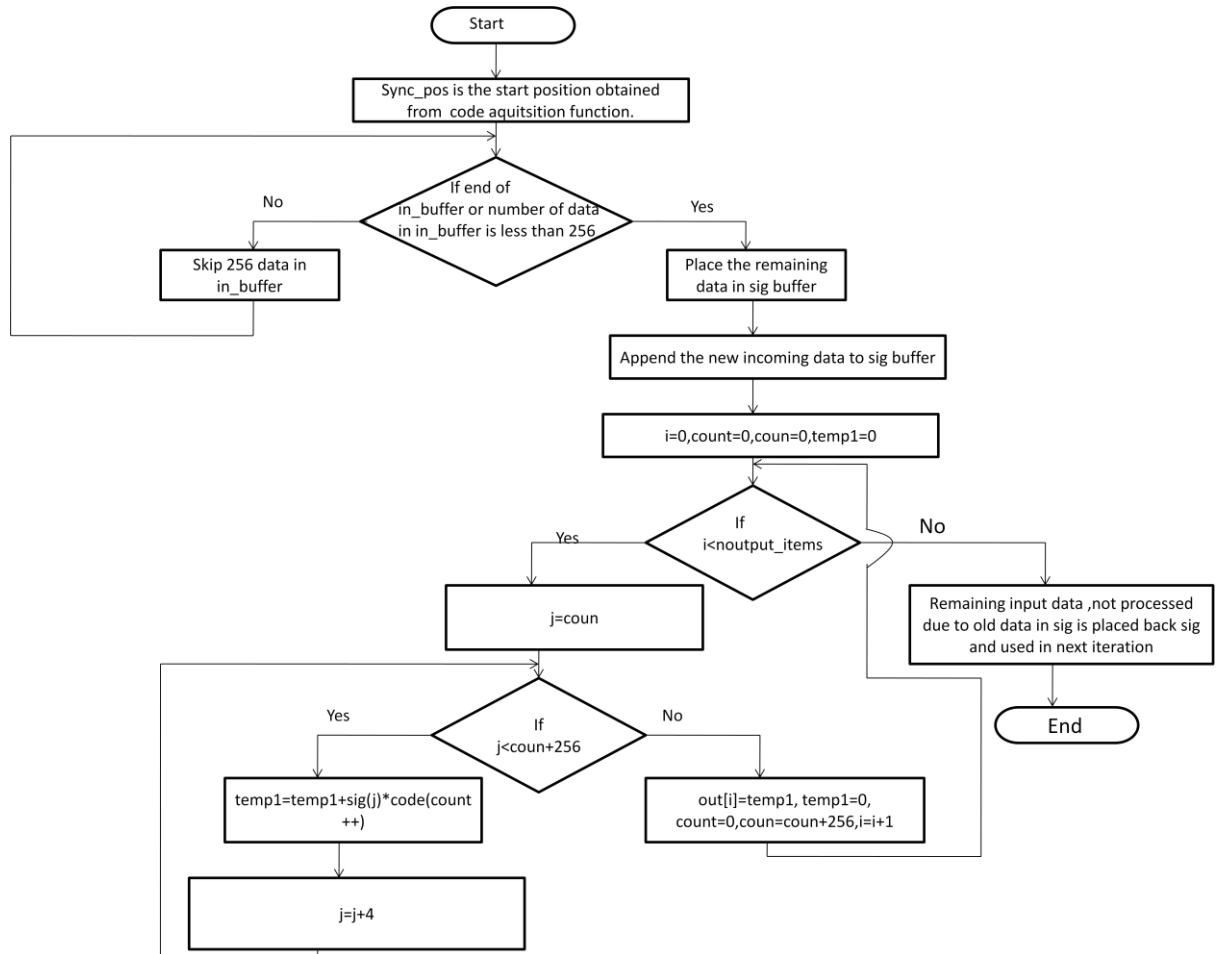


Fig 7.9: Flow chart for code despreading without code tracking.

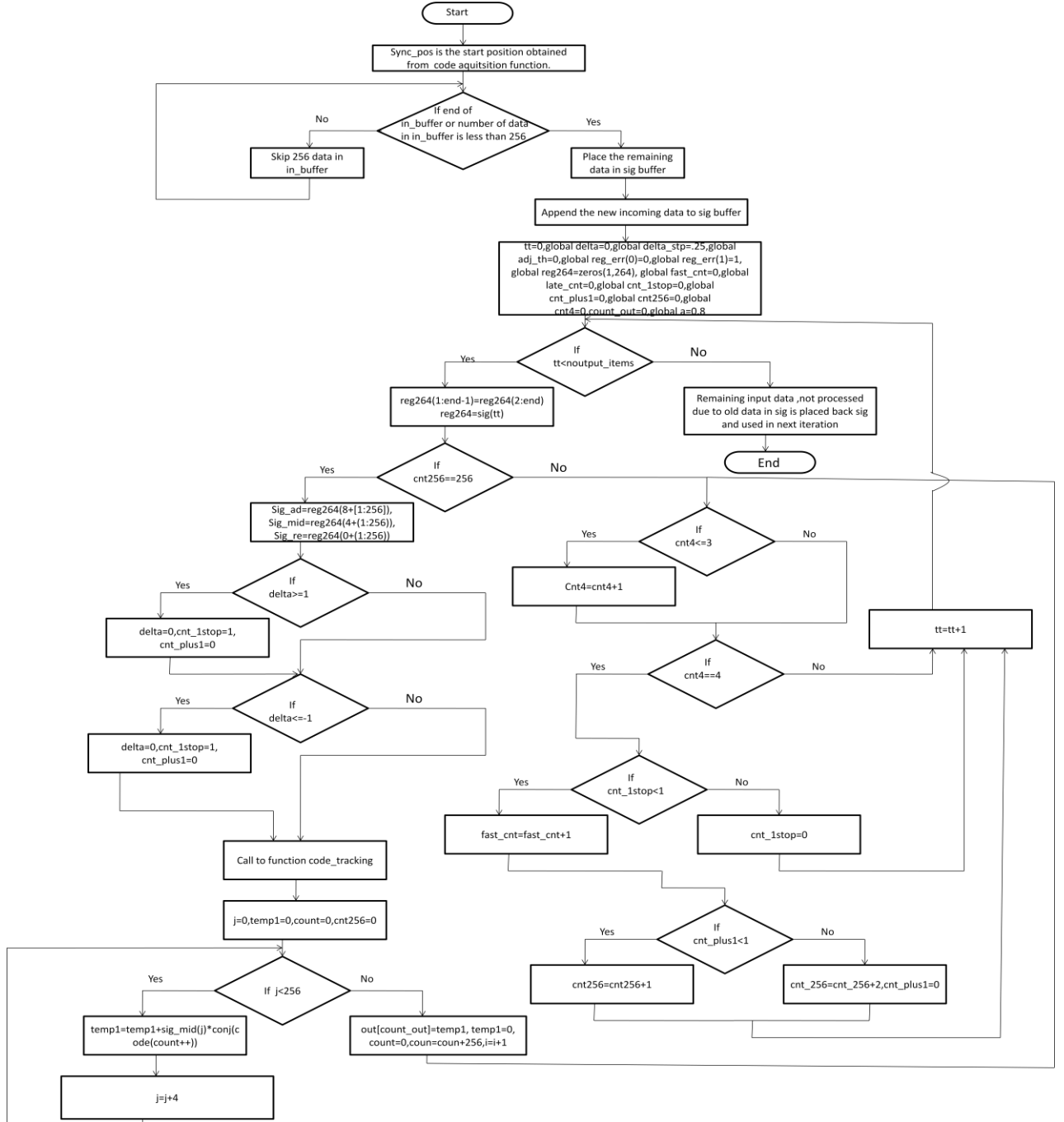


Fig 7.10: Flow chart for code despreading with code-tracking

7.5.3 Decoding with CODE Tracking

The code acquisition, buffer operations are all done similar to section 7.5.2. The start position of the code established should be verified as each time and if there is an error it should be corrected. This is done by code tracking. In despreading with code tracking we maintain three buffer sig_mid, sig_ad, sig_re where sig_mid is the correct data branch, sig_ad is advance branch and sig_re is the late branch which are similar to $R_1(\tau)$ and $R_2(\tau)$ explained in section 5.6. The late branch and early branch is used in code tracking algorithm. We maintain a counter known as cnt256 which starts from zero when this counter becomes 256 we do the despreading and code tracking. This counter places a very important role in this code tracking implementation. We have a variable known as delta which has start value of

0. This delta is incremented and decremented in the code tracking algorithm. This delta value place a crucial role in incrementing or decrementing the cnt256. If the cnt256 is not incremented then the code aquisition position is moved forward by 1 position. If the cnt256 is incremented by 2 then we move the code aquisition backward by 1 position. The code aquisition position is moved backward and forward based on the code aquisition algorithm. If delta is greater than or equal to 1 then cnt256 is not incremented and the code aquisition is moved forward by 1 position. If delta is less than or equal to -1 we increment the cnt256 by 2 and the code aquisition is moved backward by 1 position. The delta is similar to VCO explained in section 5.6. Fig 7.10 and 7.11 gives flow chart for the decoding with code tracking. Appendix A has the code for decoding with code tracking.

7.5.4 Code Tracking

The code tracking plays an important role in moving the code aquisition position back and forth. In code tracking the advance branch and late branch are multiplied with code and added. This sum from late branch and early branch is subtracted and stored in err_in variable. The reg_err(0) takes reg_err(1) value and reg_err(1) will take the value $a \cdot \text{reg_err}(0) + (1-a) \cdot \text{err_in}$, the value of a is 0.8. This is similar to loop filtering as explained in section 5.6. The adj_th is a variable which has to be determined for a particular system, this value changes depending on the device, environment. By changing values we can find the best fit for the particular device set up and environment. If reg_err(1) is greater than or equal to 1 fast_cnt is increased by 1 otherwise late_cnt is increased by 1. This fast_cnt and late_cnt are variables initialised to 0. If fast_cnt goes beyond the adj_th then current code aquisition point is going backward but the actual code aquisition point is slipping forward so we should move the aquisition point forward so the delta is increased. Similarly late_cnt goes beyond the adj_th then code current aquisition point is going forward but the actual code aquisition point is slipping backward so we should move the aquisition point backward so the delta is decreased. The despreading and code tracking has to be placed in same signal processing block since code tracking determines the code aquisition position with which code despreading has to be done. The reg_err(1) is similar to $e(\tau)$ explained in section 5.6.

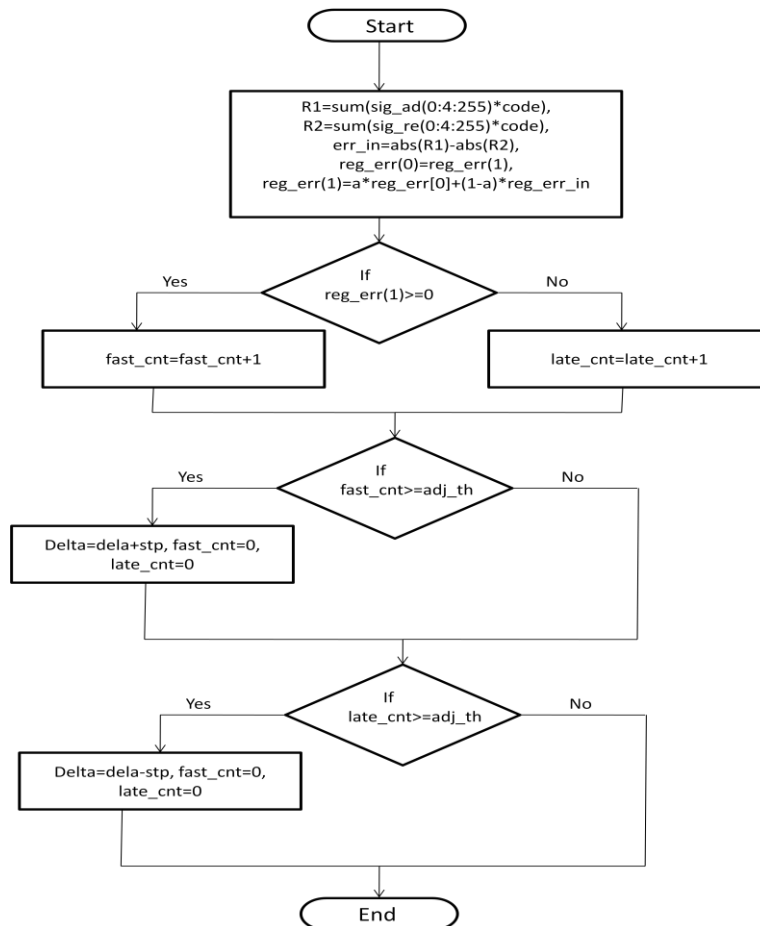


Fig 7.11: Flow chart for code tracking

7.5.5 Code Tracking and despreading in single module

It could be seen that code tracking implementation is long and we can think of splitting the block. However there is a disadvantage in GNU Radio that it is difficult to pass variables between different blocks. The code tracking block will have counter operations and it will make things a lot difficult in passing the counters between the blocks. It would be better to have a single block to do all the operation rather than passing a variable between them. This will solve the problem discussed in section 6.3

7.6 UP Link

In uplink transmission there will be more than one transmitter and one receiver. The transmitter can join the system at anytime but still the receiver should able to decode the message sent by the transmitter. In chapter 5 we saw the two users should start within the ZCZ to have the zero interference. The zone for the two codes is 16. If the transmission rate is 500e3 then the two users should start the transmission within 0.00032s.

The code which is of length 64 will repeat itself. If the transmission of the user 2 starts at the start of next ZCZ then the two users will have zero interference and can transmit the signal without any interference. The Fig 7.12 shows the graph for cross correlation for code 1 and code 2 when they are repeated thrice. The figure 7.12 clearly explains when the code repeats itself there will be new

ZCZ. When the user 2 waits for the start of next code and start the transmission due to the ZCZ property they will have zero interference.

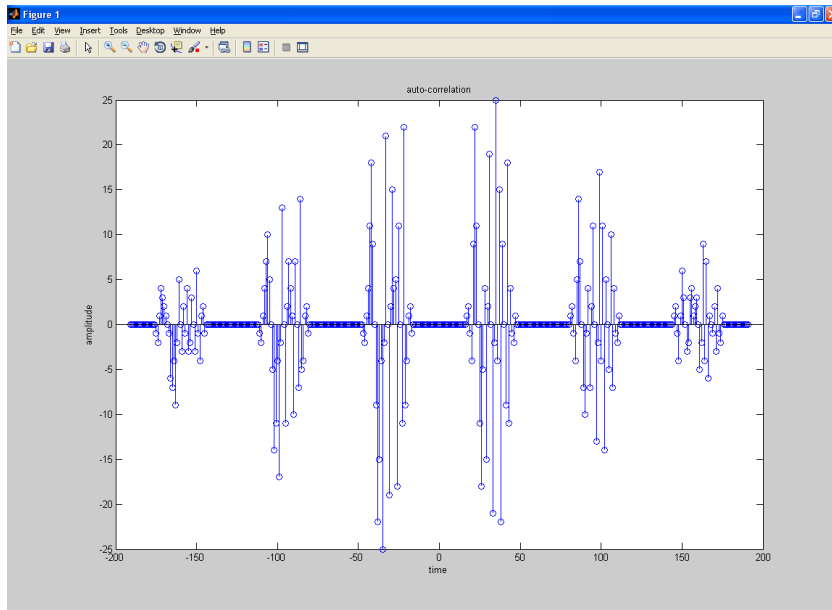


Fig 7.12: The graph for cross correlation for code 1 and code 2 when they are repeated thrice

7.6.1 Linux Server Time:

In the experiment we use GNU Radio to control the start time of two computers. The two computers should have exact time for the whole idea to work. Even a shift in milliseconds will have undesirable results. The two computers should get its time from a common clock. The common clock is the NTP (Network Time Protocol) server timing. The NTP server timing is obtained over the Internet.

The server generally gets its time mostly from GPS or any other atomic time source. Not all server use atomic time. There will be centralized atomic clock call it as stratum 0. Servers are connected in hierarchy. Server which is connected immediately next to stratum 0 is stratum 1 server. Stratum 2 server is connected to stratum 1 server and so on [28].

7.6.2 Division of Time into Slots:

As explained in section 7.6 the code will repeat itself after a particular time interval. If the second transmitter can wait till the start of next code the two users can transmit without any interference. There is another problem in this approach. The user 2 should know the start time of the user 1 to start the transmission. However it is not possible for the second user to know the start time of the first user.

To solve this problem entire time is divided into slots. Whenever a user starts transmission even if it is the first transmitter in the system it has to wait for the one of the next timeslots. If the time is divided into slots and whenever transmission starts, the transmitter will wait for the one of the next timeslots to start the transmission then it is possible to achieve zero cross correlation because of ZCZ property and it is possible to achieve zero interference. To do so we can take a time say 01-01-2012 00:00:00 as the start time of the system and divide the time after that into timeslots. This will solve the

problem we discussed in section 6.6. The Python code to control the start time of the system is shown below

```

ticks=(time.time())

ref_time='2012-1-1 00:00:00:00'

pattern='%Y-%m-%d %H:%M:%S:%f'

epoch=float(time.mktime(time.strptime(ref_time,pattern)))

offset=ticks-epoch

rem=offset%(64/500e3)

add=(ticks-rem)+(10*(64/500e3))

ticks_new=(time.time())

time.sleep(add-ticks_new)

```

The code first converts reference time which is 2012-1-1 00:00:00:00 to epoch time. All the time after that reference time is divided into slots which is multiple of $64/500e3$ second. Where 64 is the code length and 500e3 is the bit rate. This division of time slot from the reference time is shown in fig 7.13.

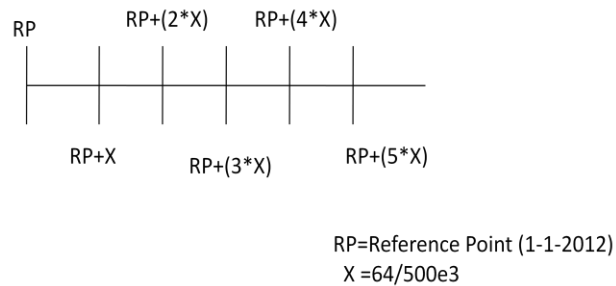


Fig 7.13: Time division of time into slots from the reference time.

The program calculates the distance from the current time to the previous time slot. From previous time slot it will wait for 10 more time slot to before start the transmission. Due to the some processing time taken by the program the next few timeslots could be consumed in this processing and it is better to skip few slots and start the transmission.

Chapter 8 Lab set Up and results

8.1 One transmitter and one receiver setup and results:

The lab set up for one transmitter and receiver is shown in figure 8.1. USRPN210 acts as receiver and USRPN200 act as transmitter. The transmitter uses ZCZ code for spreading. The signal received by the receiver is despread by the same ZCZ code. The constellation diagram for the real time data after despreading module in section 7.5.2 is shown in the fig 8.2. The constellation diagram in fig 8.2 is not as ideal as fig 5.4, but it is still a good graph and symbol can be uniquely identified with this constellation. Fig 7.8 shows the code acquisition position for the single transmitter system. It could be noted single unique peak is obtained as expected for the idle system. A simple text “**Hai INFINITUS. How are You. WELCOME!**” is transmitted from the transmitter and received properly by the receiver. For code tracking we obtain a constellation graph as shown fig 8.3. The graph in fig 8.3 is different from fig 8.2. This is because the code tracking algorithm initially adjusts the code acquisition position before settling in to a correct value. The initial adjustment leads to slight difference in the constellation diagram.

The ZCZ code has same performance for code tracking algorithm as with the normal algorithm without code tracking, so the test bed does not have problem of code acquisition position getting slipped.

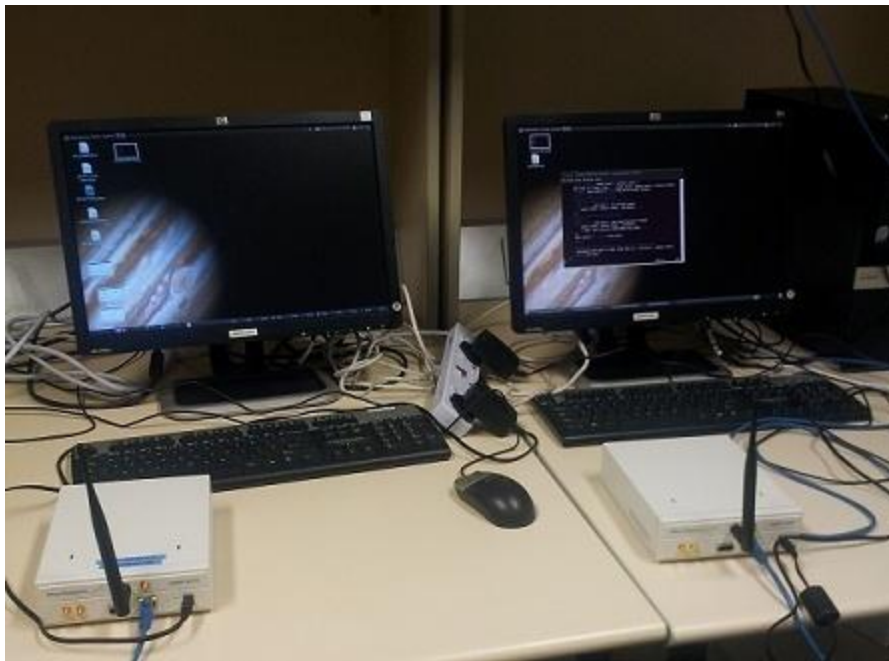


Fig 8.1: Lab set up with one transmitter and one receiver

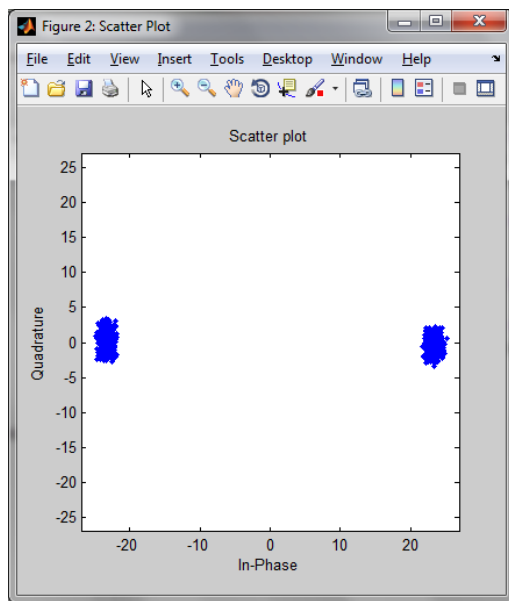


Fig 8.2: Constellation diagram after despreading without code tracking in real-time scenario

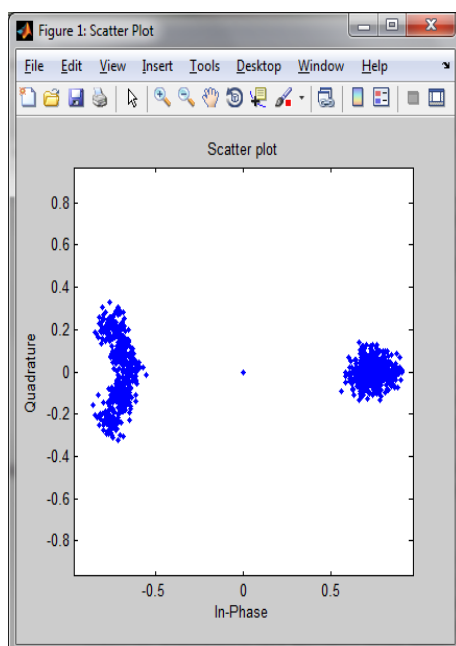


Fig 8.3: Constellation diagram for code tracking algorithm

8.2 Uplink (Two transmitters and one receiver):

In the uplink there are two transmitters and one receiver. Two USRPN200 act as transmitter and one USRPN210 acts as receiver. This is shown in fig 8.4. Here the transmitters can start the transmission at any time they want. The receiver should distinguish the each transmitter and decode the signal based on the code used by each user. The transmitters and receivers will have a common clock from GPS. The transmitter 1 will transmit plain text “**Hai INFINITUS. How are You. WELCOME!**”. The transmitter 2 will transmit a numbers “**012345678998765432156785432**”. This is to distinguish transmitter 1 and transmitter 2. The constellation diagram shown in fig 8.6 is plotted by GNU Radio for the data after despreading module on the run when the receiver actively receives the data. The constellation diagram in fig 8.6 also has time domain representation of signal before despreading. This time domain representation of the signal is to distinguish single transmitter and two transmitter uplink system. Fig 8.6 represent single transmitter and fig 8.7 represents two transmitter system. The time domain graph in fig 8.7 has higher magnitude compared to the one in fig 8.6, this indicate the constellation and time domain graph in fig 8.7 represent two transmitter system.

If the Transmitter 1 and transmitter 2 start transmitting randomly without waiting for any time slots then the two users will interfere with each other and get code aquisition graph as seen in fig 6.5 and the data received will be similar "0? IN@)NITUS.How '#a%&%/(=?OMC " instead of “**Hai INFINITUS. How are You. WELCOME!**” . If the two transmitters wait for the time slot as explained in section 7.6, we get code aquisition graph as shown in fig 8.5, which is very much similar to the single user transmitter graph in fig 7.8. This is because if the codes fall within ZCZ the codes will have ideal zero correlation. The constellation diagram in fig 8.7 for two transmitter system is not as ideal as fig 5.4, but it is still a good graph and symbol can be uniquely identified with this constellation.

First transmitter 1 transmit and the receiver receives and despread it. The constellation diagram for this scenario is shown in fig 8.6. Now the transmitter 2 waits for the time slot and start the transmission, we get constellation graph as shown in fig 8.7. The constellation still remains the same this shows two users are transmitting without interference.

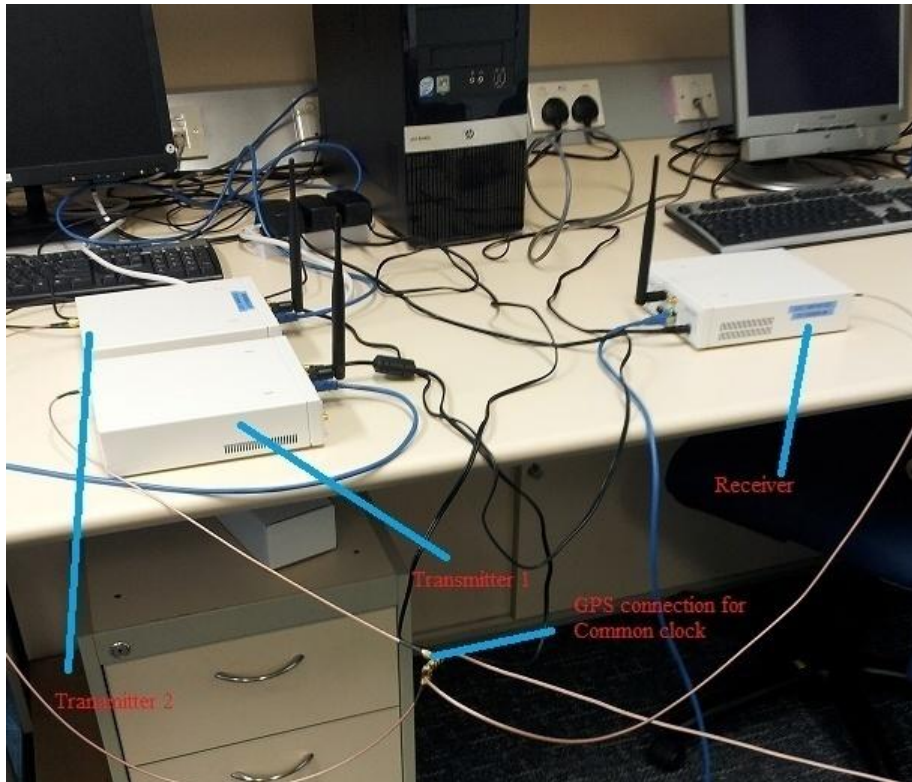


Fig 8.4: Lab set up for the Uplink two transmitters and one receiver

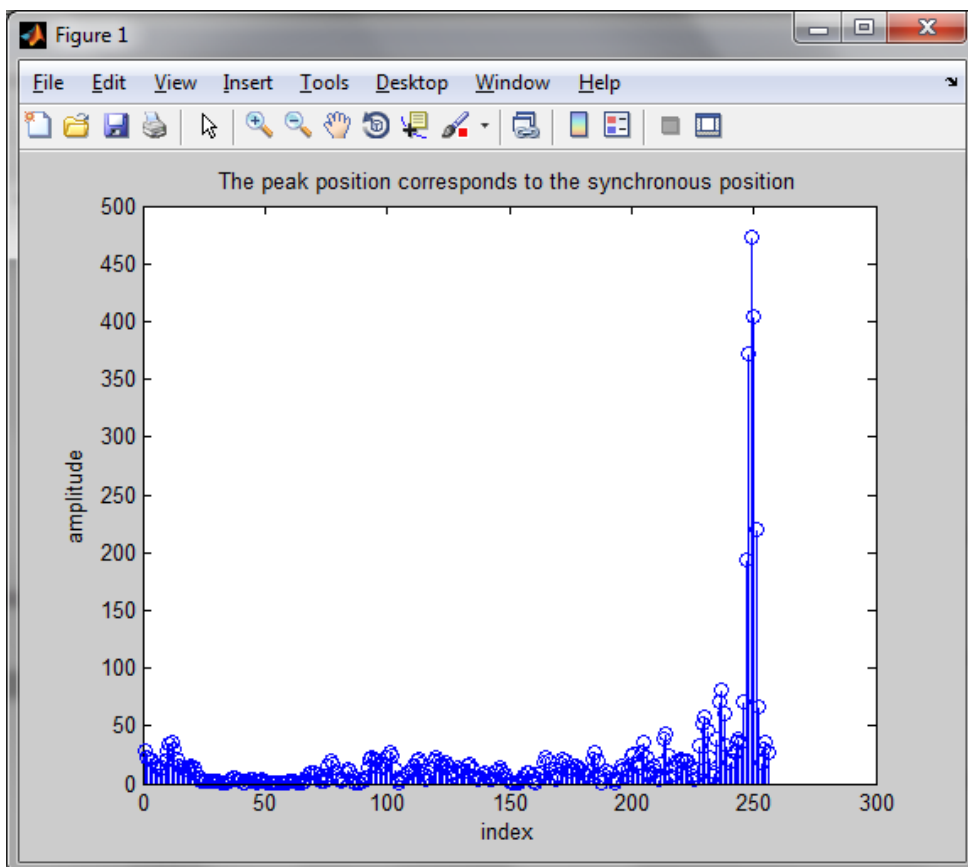


Fig 8.5: Code Acquisition diagram after two user start at time slots

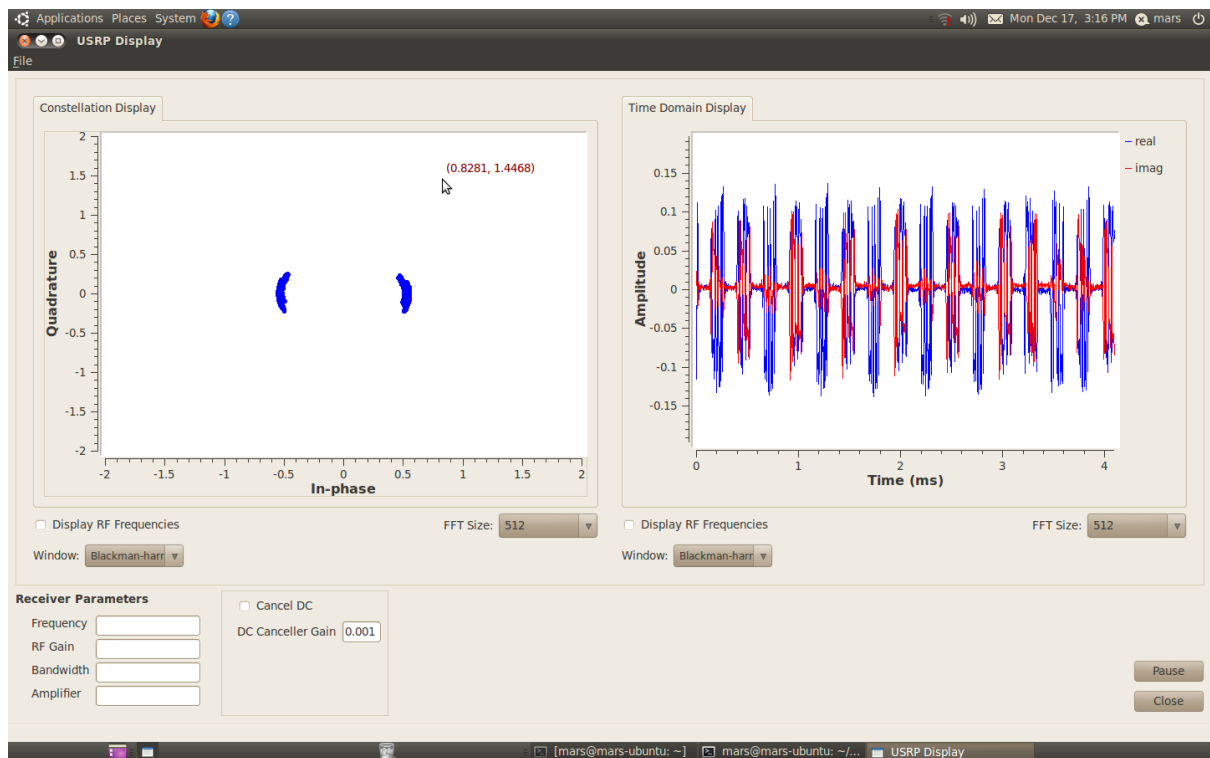


Fig 8.6: GNU Radio plot of constellation diagram and time-domain display for single user

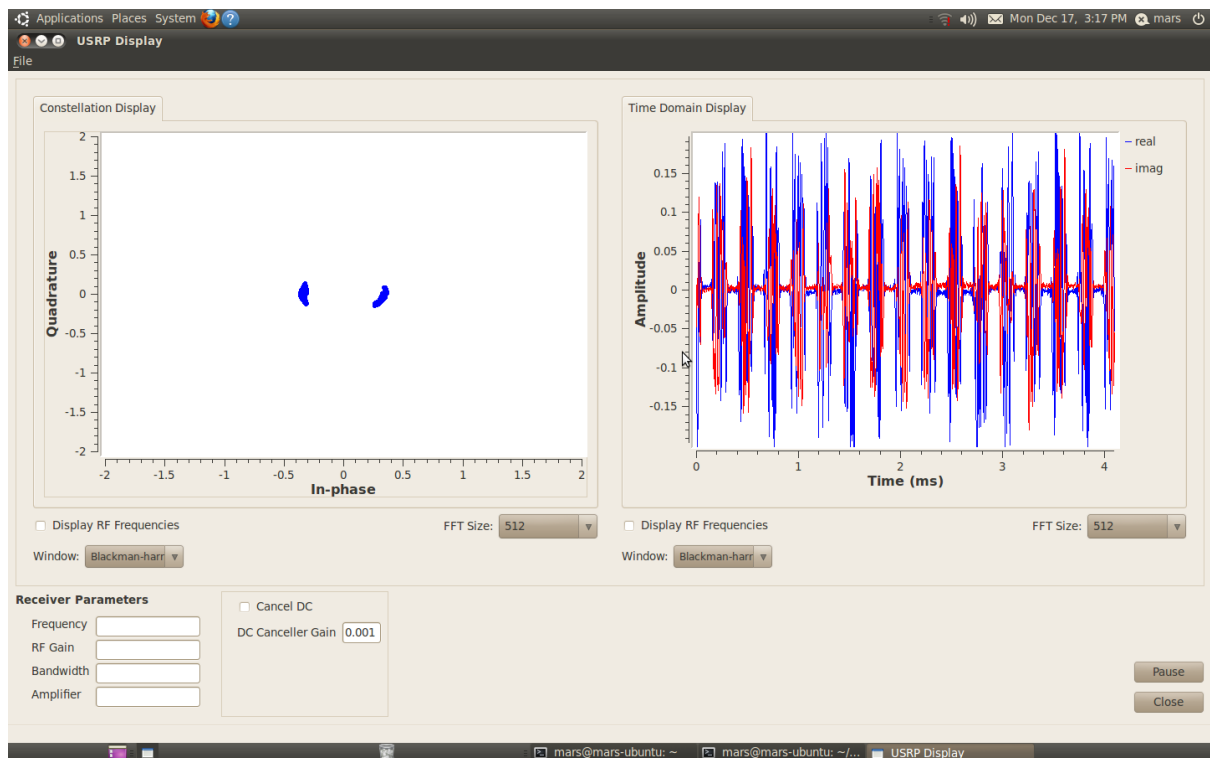


Fig 8.7: GNU Radio plot of constellation diagram and time-domain display for multi user uplink

Conclusions

Performance of the ZCZ code for real time implementation is good for a one transmitter and one receiver system, where the despreading is done with or without code tracking. For the multi user system with two transmitters and one receiver, if the transmitter starts randomly without waiting for the timeslot they will interfere with each other and the reception will be bad. If they wait for the one of the next timeslots, the two codes will fall within in ZCZ and two transmitter will have zero interference.

Future Work

Currently only two transmitters are used in this tested. In the future more than two transmitters with different codes can be tested. In the current system only a plain text is transmitted, real time audio and video can be tested. This USRP and GNU Radio have the capabilities to handle audio and video data. In the present system multiple accesses is accomplished through division of code. In the future multiple accesses are accomplished in two dimensions with respect to code and frequency.

Appendix A

Dial_tone.py

```
1. from gnuradio import gr
2. from gnuradio import audio
3.
4. class my_top_block(gr.top_block):
5.     def __init__(self):
6.         gr.top_block.__init__(self)
7.
8.         sample_rate = 32000
9.         ampl = 0.1
10.
11.         src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
12.         src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
13.         dst = audio.sink (sample_rate, "")
14.         self.connect (src0, (dst, 0))
15.         self.connect (src1, (dst, 1))
16.
17. if __name__ == '__main__':
18.     try:
19.         my_top_block().run()
20.     except KeyboardInterrupt:
21.         pass
```

Spreading.cc

```
1. /* -*- C++ -*- */
2. /*
3.  * Copyright 2012 NTU Communications Systems department NTU. Author: M.G.Praveen Kumar
4.  *
5.  * This is free software; you can redistribute it and/or modify
6.  * it under the terms of the GNU General Public License as published by
7.  * the Free Software Foundation; either version 3, or (at your option)
8.  * any later version.
9.  *
10. * This software is distributed in the hope that it will be useful,
11. * but WITHOUT ANY WARRANTY; without even the implied warranty of
12. * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13. * GNU General Public License for more details.
14. *
15. * You should have received a copy of the GNU General Public License
16. * along with this software; see the file COPYING. If not, write to
17. * the Free Software Foundation, Inc., 51 Franklin Street,
18. * Boston, MA 02110-1301, USA.
19. */
```

```

20.
21. #ifdef HAVE_CONFIG_H
22. #include "config.h"
23. #endif
24.
25. #include <gr_io_signature.h>
26. #include <zcz_spreading.h>
27.
28.
29. zcz_spreading_sptr
30. zcz_make_spreading (int arg,int user )
31. {
32.     return zcz_spreading_sptr (new zcz_spreading (arg,user));
33. }
34.
35.
36. zcz_spreading::zcz_spreading (int arg,int user)
37.     : gr_sync_interpolator ("spreading",
38.         gr_make_io_signature (1,1, sizeof (gr_complex)),
39.         gr_make_io_signature (1, 1, sizeof (gr_complex)),arg)
40. {
41.     d_inter=arg;
42.     d_user=user;
43. }
44.
45.
46. zcz_spreading::~zcz_spreading ()
47. {
48. }
49.
50.
51. int
52. zcz_spreading::work (int noutput_items,
53.                     gr_vector_const_void_star &input_items,
54.                     gr_vector_void_star &output_items)
55. {
56.     gr_complex *in = ( gr_complex *) input_items[0];
57.     gr_complex *out = (gr_complex *) output_items[0];
58.
59.     int j=0;
60.
61.     int size=d_inter;
62.     //user 1 code
63.     float code1[64]={1, 1, -1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1, -1,
64.         -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
65.         0, 0, 0, 0, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1,
66.         -1, 1, -1, 1, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0,
67.         0, 0, 0, 0, 0, 0, 0, 0 };
68.     //user 2 code
69.
70.     float code2[64]={ -1.0, 1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0,
71.         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
72.         -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0,

```



```

73.         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0 };
74.     float code[64];
75.     if(d_user==1)
76.     {
77.         for(int i=0;i<64;i++)           //determine the user type i.e whether user 1 or user 2
78.             code[i]=code1[i];
79.     }
80.     else
81.     {
82.         for(int i=0;i<64;i++)
83.             code[i]=code2[i];
84.     }
85.     spreading(code,out,in,noutput_items);
86.
87.     return noutput_items;
88. }
89. void
90. zcz_spreading::spreading(float code1[],gr_complex out[],gr_complex in[],int noutput_items)
91. {
92.     int no=0;
93.
94.
95.     for(int i=0;i<noutput_items/d_inter;i++)
96.     {
97.         for(int j=0;j<d_inter;j++)
98.         {
99.             out[j+no]=in[i] * code1[j];           //Input data is multiplied with
the code
100.        }
101.        no+=d_inter;
102.    }
103.    no=no-d_inter;
104.
105. }

```

Despreading.cc

```

1.  /* -*- C++ -*- */
2.  /*
3.   * Copyright 2012 <+YOU OR YOUR COMPANY+>.
4.   *
5.   * This is free software; you can redistribute it and/or modify
6.   * it under the terms of the GNU General Public License as published by
7.   * the Free Software Foundation; either version 3, or (at your option)
8.   * any later version.
9.   *
10.  * This software is distributed in the hope that it will be useful,
11.  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12.  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13.  * GNU General Public License for more details.
14.  *

```

```

15.  * You should have received a copy of the GNU General Public License
16.  * along with this software; see the file COPYING.  If not, write to
17.  * the Free Software Foundation, Inc., 51 Franklin Street,
18.  * Boston, MA 02110-1301, USA.
19.  */
20.
21. #ifndef HAVE_CONFIG_H
22. #include "config.h"
23. #endif
24. #include<iostream>
25. #include <gr_io_signature.h>
26. #include <zcz_despreading.h>
27. #include <fstream>
28. int check1=0;
29. int intial_no_items=0;
30. int no_items=0;
31. int new_sync_pos;
32. int check=0;
33. int track=0;
34. float code[64];
35. //code for user 1
36. float code1[64]={1, 1, -1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1, -1,
37.                  -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
38.                  0, 0, 0, 0, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1,
39.                  -1, 1, -1, 1, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0,
40.                  0, 0, 0, 0, 0, 0, 0, 0 };
41.
42. //code for user 2
43. float code2[64]={ -1, 1, 1, -1, 1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1,
44.                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
45.                   0,
46.                   -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1, 1, 1, -1, -1,
47.                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
48.                   0 };
49.
50. zcz_despreading_sptr
51. zcz_make_despreading (int arg,int user)
52. {
53.     return zcz_despreading_sptr (new zcz_despreading (arg,user));
54. }
55.
56.
57. zcz_despreading::zcz_despreading (int arg,int user)
58.     : gr_sync_decimator ("despreading", //call to decimation block
59.         gr_make_io_signature (1, 1, sizeof (gr_complex)),
60.         gr_make_io_signature (1, 1, sizeof (gr_complex)), arg)
61. {
62.     d_decim=arg;
63.     code_len=arg/4;
64.     in_buffer=new gr_complex[12000]; //constructor for despreading
65.     sig=new gr_complex[10000];

```

```

66.         d_user=user;
67.
68. }
69.
70.
71. zcz_despreading::~zcz_despreading ()
72. {
73. }
74.
75.
76. int
77. zcz_despreading::work (int noutput_items,
78.                        gr_vector_const_void_star &input_items,
79.                        gr_vector_void_star &output_items)
80. {
81.     gr_complex *in = ( gr_complex *) input_items[0];
82.     gr_complex *out = (gr_complex *) output_items[0];
83.     int co=0;
84.     int offset;
85.
86.     if(((offset=skip_data(noutput_items*d_decim))==0) && check1==0)
87.     {
88.                                     //skip some initial data
89.
90.     }
91.     //collect data in in_buffer
92.     else if(no_items<8000+240)
93.     {
94.         if(check1==1)
95.         {
96.             check1++;
97.
98.             for(int j=(noutput_items*d_decim)-offset;j<noutput_items*d_decim;j++)
99.                 in_buffer[no_items++]=in[j];
100.        }
101.        else
102.        {
103.            for(int i=no_items;i<(noutput_items*d_decim)+no_items;i++)
104.            {
105.                in_buffer[i]=in[co++];
106.
107.            }
108.            no_items+=noutput_items*d_decim;
109.
110.        }
111.    }
112.    else
113.    {
114.        if(check==0)
115.        {
116.            int OverSampRatio=4;
117.            int L=code_len;

```

```

118.         int datalen_4_sync=30;
119.         gr_complex sig_4sync[(code_len)*4*(datalen_4_sync-2)];
120.         int tempc=0;
121.         for(int i=240;i<(code_len)*4*(datalen_4_sync-2)+240;i++)
122.             sig_4sync[tempc++]=in_buffer[i];
123.
124.         int sync_pos=32;
125.         int d_len=code_len*4*(datalen_4_sync-2);
126.         //determine the user type
127.
128.         if(d_user==1)
129.         {
130.             for(int i=0;i<64;i++)
131.                 code[i]=code1[i];
132.         }
133.         else if(d_user==2)
134.         {
135.             for(int i=0;i<64;i++)
136.                 code[i]=code2[i];
137.
138.         }
139.         //call to synchronization function to achieve aquisition
140.         sync_pos=synchronization(sig_4sync,code,d_len);
141.
142.         std::cout<<"The synchronization position is : "<<sync_pos;
143.         check++;
144.         new_sync_pos=sync_pos+240;
145.         while(1)
146.         {
147.             new_sync_pos+=d_decim;
148.             if(new_sync_pos>=no_items)
149.             {
150.                 new_sync_pos-=d_decim;
151.                 //move to last synchronization position in the in_buffer and make it as new sync position
152.                 break;
153.             }
154.         }
155.         new_sync_pos=new_sync_pos-1;
156.         track=no_items-new_sync_pos;
157.         //move remaining data in in_buffer from new sync position to sig
158.         memcpy(sig,in_buffer+(new_sync_pos),(no_items-new_sync_pos)*sizeof(gr_complex));
159.
160.     }
161.     else
162.     {
163.         int coun=0;
164.         int count=0;
165.         gr_complex temp1=0;
166.         //append the incoming data to sig buffer
167.         memcpy(sig+track,in,(noutput_items*d_decim)*sizeof(gr_complex));
168.
169.         for(int i=0;i<noutput_items;i++)
170.         {

```

```

171.         for(int j=coun;j<coun+d_decim;j+=4)
172.         {
173.
174.             temp1+=(sig[j])*code[count++];    //despread the data
175.         }
176.         int te=real(temp1*conj(temp1));
177.
178.         out[i]=temp1;
179.         temp1=0;
180.         count=0;
181.         coun+=d_decim;
182.     }
183.     //place back the remaing data due to backlog to the start of sig buffer
184.
185.     for(int i=0;i<track;i++)
186.     {
187.         sig[i]=sig[(noutput_items*d_decim)+i];
188.
189.     }
190. }
191.
192. return noutput_items;
193. }
194. int
195. zcz_despreading:: skip_data(int noutput_items)
196. {
197.     intial_no_items+=noutput_items;
198.     if(intial_no_items<20000)
199.     {
200.
201.         return 0;
202.     }
203.     //function that perform skipping of data
204.     else
205.     {
206.         check1++;
207.         return (intial_no_items-20000);
208.     }
209. }
210. int
211. zcz_despreading:: synchronization(gr_complex sig[],float code_waveform[],int d_len)
212. {
213.
214.     int code_size=code_len;
215.     int n=floor(d_len/d_decim);
216.
217.     std::cout<<"n:"<<n;
218.     gr_complex CODE2[code_size*(n-1)];
219.     int no=0;
220.
221.     for(int i=0;i<n-1;i++)
222.     {
223.         for(int j=0;j<code_size;j++)

```

```

224.         {
225.             CODE2[j+no]=code_waveform[j];
226.         } //function that does the synchronization
227.         no+=code_size;
228.     }
229.
230.     float max_corr[d_decim];
231.     gr_complex temp1[(n-1)*d_decim];
232.     gr_complex temp2[(n-1)];
233.     float temp3[n-1];
234.     float max_c_t[d_decim][n-1];
235.     for(int i=0;i<d_decim;i++)
236.     {
237.         int count=0;
238.         for(int j=i;j<(((n-1)*d_decim)+i);j+=4)
239.         {
240.             temp1[count]=sig[j]*(CODE2[count]);
241.             count++;
242.         }
243.
244.
245.         gr_complex sum=0;
246.         int l=0;
247.         for(int j=0;j<n-1;j++)
248.         {
249.             for(int k=0;k<d_decim;k++)
250.             {
251.                 sum+=temp1[k+l];
252.
253.             }
254.             temp2[j]=sum;
255.             l+=d_decim;
256.             sum=0;
257.         }
258.
259.         for(int j=0;j<n-1;j++)
260.         {
261.             temp3[j]=real(temp2[j]*conj(temp2[j]));
262.             max_c_t[i][j]=temp3[j];
263.
264.         }
265.         float sum1=0;
266.         for(int j=0;j<n-1;j++)
267.         {
268.             sum1+=temp3[j];
269.         }
270.         max_corr[i]=sum1;
271.     }
272.
273.     for(int i=0;i<d_decim;i++)
274.     {
275.         std::cout<<"\npos:"<<i<<":"<<max_corr[i];
276.     }

```

```

277.
278.
279.
280.
281.     float temp=max_corr[0];
282.     int pos=0;
283.
284.     for(int i=1;i<d_decim;i++)
285.     {
286.         if(max_corr[i]>temp)
287.         {
288.             temp=max_corr[i];
289.             pos=i;
290.         }
291.     }
292.
293.     for(int j=0;j<n-1;j++)
294.         std::cout<<" "<<max_c_t[pos][j]<<" ";
295.
296.     return pos;
297.
298.
299. }

```

Code tracking Despreading.cc

```

1.  /* -*- C++ -*- */
2.  /*
3.   * Copyright 2012 NTU Communication department. Author: M.G.Praveen Kumar
4.   *
5.   * This is free software; you can redistribute it and/or modify
6.   * it under the terms of the GNU General Public License as published by
7.   * the Free Software Foundation; either version 3, or (at your option)
8.   * any later version.
9.   *
10.  * This software is distributed in the hope that it will be useful,
11.  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12.  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13.  * GNU General Public License for more details.
14.  *
15.  * You should have received a copy of the GNU General Public License
16.  * along with this software; see the file COPYING. If not, write to
17.  * the Free Software Foundation, Inc., 51 Franklin Street,
18.  * Boston, MA 02110-1301, USA.
19.  */
20.
21. #ifdef HAVE_CONFIG_H
22. #include "config.h"
23. #endif
24.

```

```

25. #include <gr_io_signature.h>
26. #include <zcz_codesync_despreading.h>
27.
28. #include <math.h>
29. #include <fstream>
30. #include <iostream>
31. #include "stdio.h"
32. #include <stdio.h>
33. #include <string.h>
34. #include<stdlib.h>
35. #include <vector>
36. // #include "mat.h"
37. int check=0;
38. std::fstream myfile1;
39. int c=0;
40. int track;
41. int check1=0;
42. float reg_err[2]={0,0};
43. int fast_cnt=0;
44. int late_cnt=0;
45. float delta=0.0;
46. gr_complex sig_ad[256],sig_mid[256],sig_re[256];
47. float delta_stp=0.25;
48. int adj_th=100;
49. float a=0.8;
50. int flag1=0;
51. int new_sync_pos;
52. int intial_no_items=0;
53. int no_items=0;
54. int cnt_1stop=0;
55. int cnt_plus1=0;
56. int cnt4=0;
57. float code[64];
58. //code for user 1
59. float code1[64]={1, 1, -1, -1, -1, -1, -1, -1, -1, 1, -1, 1, 1, -1,
60.                 -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
61.                 0, 0, 0, 0, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1,
62.                 -1, 1, -1, 1, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
63.                 0, 0, 0, 0, 0, 0, 0, 0 };
64. //code for user 2
65. float code2[64]={ -1, 1, 1, -1, 1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1,
66.                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
67.                  -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, -1,
68.                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
69.                  0 };
70.
71.
72.
73. zcz_codesync_despreading_sptr
74. zcz_codesync_make_despreading (int arg,int user)
75. {

```



```

76.         return zcz_codesync_despreading_sptr (new zcz_codesync_despreading (arg,user));
77.     }
78.
79.
80. zcz_codesync_despreading::zcz_codesync_despreading (int arg,int user)
81.     : gr_sync_decimator ("despreading",                                     //call to decimation block
82.         gr_make_io_signature (1, 1, sizeof (gr_complex)),
83.         gr_make_io_signature (1, 1, sizeof (gr_complex)), arg)
84. {
85.     d_decim=arg;
86.     d_user=user
87.     in_buffer=new gr_complex[12000];
88.     sig=new gr_complex[5000];                                             //constructor
89.     data_buffer=new gr_complex[12000];
90. }
91.
92.
93. zcz_codesync_despreading::~zcz_codesync_despreading ()
94. {
95. }
96.
97.
98. int
99. zcz_codesync_despreading::work (int noutput_items,
100.                                gr_vector_const_void_star &input_items,
101.                                gr_vector_void_star &output_items)
102. {
103.     gr_complex *in = (gr_complex *) input_items[0];
104.     gr_complex *out = (gr_complex *) output_items[0];
105.
106.     gr_complex sum1=0;
107.
108.     int co=0;
109.     int offset;
110.     if(((offset=skip_data(noutput_items*d_decim))==0) && check1==0){
111.
112.         //skip initial data
113.     }
114.     //collect data in in_buffer
115.     else if(no_items<8000+240)
116.     {
117.         if(check1==1)
118.         {
119.             std::cout<<"offset:"<<offset;
120.             check1++;
121.
122.             for(int j=(noutput_items*d_decim)-offset;j<noutput_items*d_decim;j++)
123.                 in_buffer[no_items++]=in[j];
124.         }
125.         else
126.         {
127.             for(int i=no_items;i<(noutput_items*d_decim)+no_items;i++)
128.                 in_buffer[i]=in[co++];
129.             no_items+=noutput_items*d_decim;

```

```

128.     }
129. }
130. else
131. {
132.
133.
134.     if(check==0)
135.     {
136.         int OverSampRatio=4;
137.         int L=64;
138.         int datalen_4_sync=32;
139.         gr_complex sig_4sync[64*4*(32-2)];
140.                                     //determine the user type
141.         if(d_user==1)
142.         {
143.             for(int i=0;i<64;i++)
144.                 code[i]=code1[i];
145.
146.         else if(d_user==2)
147.         {
148.             for(int i=0;i<64;i++)
149.                 code[i]=code2[i];
150.
151.         }
152.         int tempc=0;
153.         for(int i=240;i<64*4*30+240;i++)
154.             sig_4sync[tempc++]=in_buffer[i];
155.
156.         int sync_pos=32;
157.         int d_len=64*4*(32-2);
158.
159.
160.                                     //call to synchronization function to achieve aquisition
161.         sync_pos=synchronization(sig_4sync,code,d_len);
162.
163.
164.         check++;
165.         new_sync_pos=sync_pos+240;
166.
167.         while(1)
168.         {
169.             new_sync_pos+=256;
170.             if(new_sync_pos>=no_items)
171.             {
172.                 new_sync_pos-=256; //move to last synchronization position in the
in_buffer and make it as new sync position
173.                 break;
174.             }
175.         }
176.         new_sync_pos=new_sync_pos-1;
177.         track=no_items-new_sync_pos;
178.         memcpy(sig,in_buffer+(new_sync_pos),(no_items-new_sync_pos)*sizeof(gr_complex));

```

```

179.
180.
181.     }
182.     else
183.     {
184.
185.         memcpy(sig+track,in,(noutput_items*d_decim)*sizeof(gr_complex));
186.         gr_complex temp_out[5000];
187.
188.         int coun=0;
189.         int count=0;
190.         gr_complex temp1=0;
191.
192.         int count1n=0;
193.         gr_complex reg264[264];
194.         int cnt256=1;
195.         int Plen=512;
196.
197.         gr_complex din;
198.         gr_complex s_cur=0,s_pre=0;
199.         gr_complex s_cur_pre=0;
200.         gr_complex rdata_NoCFO[1000];
201.         int * rdata_NoCFO_4debug;
202.         int cnt4PSYNC=0;
203.
204.         gr_complex cfo_est;
205.
206.
207.
208.         int count_out=0;
209.         int cou=0;
210.
211.         for(int tt=0;tt<noutput_items*d_decim;tt++)
212.         {
213.             din=sig[tt];
214.
215.
216.             for(int k=0;k<263;k++)
217.             {
218.                 reg264[k]=reg264[k+1];
219.             }
220.             reg264[263]=din;           //new data at the end
221.
222.
223.             if(cnt256==256)
224.             {
225.
226.                 int co=0;
227.                 //early branch
228.                 for(int j=8;j<264;j++)
229.                     sig_ad[co++]=reg264[j];
230.                 co=0;
231.                 //correct data buffer which will be sent to despreading

```

```

232.         for(int k=4;k<260;k++)
233.             sig_mid[co++]=reg264[k];
234.         //late branch
235.         co=0;
236.         for(int j=0;j<256;j++)
237.             sig_re[co++]=reg264[j];
238.         //delta controls the movement of cnt256
239.         if(delta>=1)
240.         {
241.             delta=0;
242.             cnt_1stop=1;
243.             cnt_plus1=0;
244.         }
245.         if(delta<=-1)
246.         {
247.             delta=0;
248.             cnt_plus1=1;
249.             cnt_1stop=0;
250.
251.         }
252.
253.         code_tracking(sig_ad,sig_re,code);
254.
255.         cnt256=0;
256.         for(int j=0;j<d_decim;j+=4)
257.         {
258.             //despreading carried out at sig_mid
259.
260.             temp1+=(sig_mid[j])*(code[count++]);
261.
262.
263.             out[count_out++]=temp1;
264.             count+=d_decim;
265.             count=0;
266.             temp1=0;
267.         }
268.         //counter operations to which move the code aquisation position
269.         if(cnt4<=3)
270.         {
271.             cnt4=cnt4+1;
272.         }
273.
274.         if(cnt4==4)
275.         {
276.             if(cnt_1stop<1)
277.             {
278.                 if(cnt_plus1<1){
279.
280.                     cnt256=cnt256+1;
281.
282.                 }
283.                 else
284.                 {

```

```

284.
285.             cnt256=cnt256+2;
286.             cnt_plus1=0;
287.         }
288.     }
289.     else
290.         cnt_1stop=0;
291.
292. }
293.
294.
295.
296.
297. }
298.
299. //move remaining data in in_buffer from new sync position to sig
300.
301.     for(int i=0;i<track;i++)
302.     {
303.         sig[i]=sig[(noutput_items*d_decim)+i];
304.     }
305.
306. }
307. }
308.
309.
310.     return noutput_items;
311. }
312.
313. int
314. zcz_codesync_despreading:: synchronization(gr_complex sig[],float code_waveform[],int d_len)
315. {
316.
317.
318.
319.     int code_size=64;
320.
321.     int n=floor(d_len/256);
322.     std::cout<<"d_len:"<<d_len;
323.     std::cout<<"n:"<<n;
324.     gr_complex CODE2[code_size*(n-1)];
325.     int no=0;
326.
327.     for(int i=0;i<n-1;i++)
328.     {
329.         for(int j=0;j<code_size;j++)
330.         {
331.             CODE2[j+no]=code_waveform[j];
332.         }
333.         no+=code_size;
334.     }
335.
336.     float max_corr[256];

```

```

337.     gr_complex temp1[(n-1)*256];
338.     gr_complex temp2[(n-1)];
339.     float temp3[n-1];
340.     float max_c_t[256][n-1];
341.     for(int i=0;i<256;i++)
342.     {
343.         int count=0;
344.         for(int j=i;j<(((n-1)*256)+i);j+=4)
345.         {
346.             temp1[count]=sig[j]*(CODE2[count]);    //function that does the synchronization
347.             count++;
348.         }
349.
350.
351.         gr_complex sum=0;
352.         int l=0;
353.         for(int j=0;j<n-1;j++)
354.         {
355.             for(int k=0;k<256;k++)
356.             {
357.                 sum+=temp1[k+1];
358.
359.             }
360.             temp2[j]=sum;
361.             l+=256;
362.             sum=0;
363.         }
364.
365.         for(int j=0;j<n-1;j++)
366.         {
367.             temp3[j]=real(temp2[j]*conj(temp2[j]));
368.             max_c_t[i][j]=temp3[j];
369.
370.         }
371.         float sum1=0;
372.         for(int j=0;j<n-1;j++)
373.         {
374.             sum1+=temp3[j];
375.         }
376.         max_corr[i]=sum1;
377.     }
378.
379.     for(int i=0;i<256;i++)
380.     {
381.         std::cout<<"\npos:"<<i<<":"<<max_corr[i];
382.     }
383.
384.
385.
386.
387.     float temp=max_corr[0];
388.     int pos=0;
389.

```

```

390.
391.
392.
393.     for(int i=1;i<256;i++)
394.     {
395.         if(max_corr[i]>temp)
396.         {
397.             temp=max_corr[i];
398.             pos=i;
399.         }
400.     }
401.
402.     for(int j=0;j<n-1;j++)
403.         std::cout<<" "<<max_c_t[pos][j]<<" ";
404.
405.     return pos;
406.
407.
408.
409.
410. }
411.
412. int
413. zcz_codesync_despreading:: skip_data(int noutput_items)
414. {
415.     intial_no_items+=noutput_items;
416.     if(intial_no_items<20000)
417.     {
418.
419.         return 0; //function that perform skipping of data
420.     }
421.     else
422.     {
423.         check1++;
424.         return (intial_no_items-20000);
425.     }
426.
427. }
428.
429.
430. void zcz_codesync_despreading::code_tracking(gr_complex sig_ad[],gr_complex sig_re[],float code[])
431. {
432.     gr_complex R1=0,R2=0;
433.     int count=0;
434.     for(int i=0;i<256;i+=4)
435.     {
436.         R1+=(sig_ad[i]*(code[count++]));
437.         R2+=(sig_re[i]*(code[count++]));
438.     }
439.     float err_in=abs(R1)-abs(R2);
440.     //IIR loop filtering
441.     reg_err[0]=reg_err[1];
442.     reg_err[1]=a*reg_err[0]+(1-a)*err_in;

```

```

443.
444.     if(reg_err[1]>=0)// collecting the dc component of error signal
445.     {
446.         fast_cnt=fast_cnt+1;    //early branch win
447.
448.     }
449.     else
450.     {
451.         late_cnt=late_cnt+1;    //late branch wins
452.     }
453.     if (fast_cnt>=adj_th)
454.     {
455.         delta=delta+delta_stp;
456.         fast_cnt=0;              //makes incoming signal faster
457.         late_cnt=0;
458.
459.     }
460.     if(late_cnt>=adj_th)
461.     {
462.         delta=delta-delta_stp;
463.         fast_cnt=0;              //make incoming signal slower
464.         late_cnt=0;
465.     }
466.
467. }

```


References

- [1] Bernard Sklar (2001). Digital Communications Fundamentals and Applications. 2nd ed. Prentice-Hall Inc., New Jersey
- [2] Jhong Sam Lee and Leonard E Miller (1988) . CDMA System Engineering Hand book. 1st ed. Artech House Inc., 1998, Norwood, MA, USA
- [3] GNU Radio Project [Online] Available: <http://gnuradio.org/redmine/projects/gnuradio/wiki> Last Accessed (27 March 2013)
- [4] GNU Radio Project [Online] Available:<http://gnuradio.org/redmine/projects/gnuradio/wiki/USRP> Last Accessed (27 March 2013)
- [5] ETTUS Research, "USRP Networked Series" [Online]
Available:https://www.ettus.com/content/files/2987_Ettus_N200-210_DS_FINAL_1.27.12_1.pdf
Last Accessed (27 March 2013)
- [6] GNU Radio Using of Included tools and Utility Program [Online]
Available:<http://gnuradio.org/redmine/projects/gnuradio/wiki/HowToUse#Using-the-included-tools-and-utility-programs> Last Accessed (27 March 2013)
- [7] GNU Radio Project [online] Available: <http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html#block-diagram-fig> Last Accessed (27 March 2013)
- [8] Martin Braun. Comprehensive GNU Radio Archive Network [Online]
Available: <https://cgran.org/wiki/devtools> Last Accessed(27 March 2013)
- [9] ETTUS RESEARCH, "USRP E100" [online] <https://www.ettus.com/product/details/UE100-KIT>
Last Accessed (27 March 2013)
- [10] Radio Ware, "The USRP Board" [Online]
Available:<http://radioware.nd.edu/documentation/hardware/the-usrp-board> Last Accessed (27 March 2013)
- [11] William Stalling (2005). wireless communications and networks. 2nd ed. Prentice-Hall Inc., New Jersey
- [12]Wireless innovation Forum, "Software Defined Radio" [Online]
Available: http://www.wirelessinnovation.org/Introduction_to_SDR Last Accessed (27 March 2013)
- [13] Base Classes for GR Blocks [Online]
Available:http://gnuradio.org/doc/doxygen/classgr__sync__interpolator.html#details Last Accesses(27 March 2013)
- [14] Python Software Foundation, "The Python Tutorial" [Online] Available:
<http://docs.python.org/tutorial/> Last Accessed(27 March 2013)

- [15] Tutorial Points, "Python Tutorial" [Online] Available:
<http://www.tutorialspoint.com/python/index.htm> Last Accessed (27 March 2013)
- [16] GNU Radio understanding Flow Graphs [Online] Available:
<http://gnuradio.org/redmine/projects/gnuradio/wiki/TutorialsWritePythonApplications> Last Accessed (27 March 2013)
- [17] Base Classes for GR Block [Online] Available:
http://gnuradio.org/doc/doxygen/classgr__block.html Last Accessed (27 March 2013)
- [18] Base Class for GR Block [Online]
 Available:http://gnuradio.org/doc/doxygen/classgr__sync__block.html Last Accessed (27 March 2013)
- [19] Writing Signal Processing Block [Online] Available:<http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html> Last Accessed (27 March 2013)
- [20] ETTUS Research, "USRP2 Datasheet" [Online] Available:
http://www.olifantasia.com/gnuradio/usrp/files/datasheets/ds_usrp2.pdf Last Accessed (27 March 2013)
- [21] The Math Work Inc., "Kron Function" [Online] Available:
<http://www.mathworks.com/help/techdoc/ref/kron.html> Last Accessed (27 March 2013)
- [22] Gerald Youngblood, "A Software Defined Radio For Masses" [Online] Available:
<http://www.flex-radio.com/Data/Doc/qex1.pdf> Last Accessed (27 March 2013)
- [23] Adrio Communication Ltd., "Software Defined Radio, SDR, Tutorial" [Online] Available:
<http://www.radio-electronics.com/info/rf-technology-design/sdr/software-defined-radios-tutorial.php> Last Accessed (27 March 2013)
- [24] One Stop Gate.com, "An Overview Of Software Defined Radio" [Online]
 Available:<http://www.onestopgate.com/gate-study-material/electronics/radio-receiver/overview-software.asp> Last Accessed (27 March 2013)
- [25] ETTUS Research, "RFX2400" [Online]
 Available:<https://www.ettus.com/product/details/RFX2400> Last Accessed (27 March 2013)
- [26] Naveen Manicka (2007), "GNU Radio Testbed"
 Available:http://www.eecis.udel.edu/~manicka/Research/NaveenManicka_Thesis.pdf Last Accessed (27 March 2013)
- [27] Fan, P.Z "Class of binary sequence with zero correlation Zone" Electronics Letter, May 1999, Vol. 39, issue: 10, page 777-779
- [28] The Network Timed Protocol [Online] Available: <http://www.ntp.org/> Last Accessed (27 March 2013)
- [29] H. H. Chen, J. F. Yeh, and N. Suehiro, "A multicarrier CDMA architecture based on orthogonal complementary codes for new generations of wideband wireless communications," *IEEE Commun. Mag.*, vol. 39, no.10, pp.126-135, 2001

- [30] A. Pezeshki, A. R. Calderbank, W. Moran, and S. D. Howard, "Doppler Resilient Golay Complementary Waveforms," *IEEE Trans. Inf. Theory*, vol. 54, no. 9, pp. 4254-4266, 2008.
- [31] Z. L. Liu, Y. L. Guan, "Meeting the Levenshtein Bound with Equality by Weighted-Correlation Complementary Set," accepted by Proc. 2012 IEEE Int. Symposium on Information Technology (ISIT'2012), MIT, Boston, US, Jul. 2012.
- [32] Z. L. Liu, Y. L. Guan, and W. H. Mow, "Improved Lower Bound for Quasi-Complementary Sequence Set," Proc. 2011 IEEE Int. Symposium on Information Technology (ISIT'2011), St-Petersburg, Russia, Aug. 2011.
- [33] Z. L. Liu, Y. L. Guan, and B. C. Ng, "A New Class of Quadriphase Even-shift Orthogonal Sequences," in The Fourth IEEE International Workshop on Signal Design and its Applications in Communications (IWSDA'09), Fukuoka, Japan, Oct. 2009, pp. 12-15
- [34] Davis, J.A, " Peak-to-Mean Power Control in OFDM, Golay Complementary Sequences, and Reed-Muller Codes" *IEEE Trans. Inf. Theory* vol. 45, no. 7, pp. 2397-2417, 1999
- [35] Slimane.S.B, GLASS.A, " Multi-Carrier CDMA Systems Using Bridge Functions", IEEE Vehicular Technology Conference proceedings Vol. 3, pp. 1928-1932, 2000

TRITA-ICT-EX-2013:75