DEVELOPMENT AND TESTING OF THE RHINO HOST STREAMED DATA ACQUISITION FRAMEWORK

A thesis submitted to the Department of Electrical Engineering,
UNIVERSITY OF CAPE TOWN, in fulfilment of the requirements for the degree of

Master of Science
MSc(Eng) by Dissertation in Electrical Engineering

*a*t the

Faculty of Engineering and the Built Environment
University of Cape Town

*b*y

Mpati Boleme

Supervised by :
DR. SIMON WINBERG
AND
MR. LERATO MOHAPI

June 18, 2017

Declaration

I know the meaning of plagiarism and declare that all the work in this dissertation, save for that which is properly acknowledged and referenced, is my own. It is being submitted for the degree of **Master of Science** in Electrical Engineering at the University of Cape Town. This work has not been submitted before for any other degree or examination in any other university.

Signature of Author: Signed by candidate Signature removed
University of Cape Town

Cape Town
June 18, 2017

ABSTRACT

This project focuses on developing a supporting framework for integrating the Reconfigurable Hardware INterface for computing and radiO (RHINO) with a Personal Computer (PC) host in order to facilitate the development of Software Defined Radio (SDR) applications built using a hybrid RHINO/multicore PC system. The supporting framework that is the focus of this dissertation is designed around two main parts: a) resources for integrating the GNU Radio framework with the RHINO platform to allow data streams to be sent from RHINO to be processed by GNU Radio, and b) a concise and highly efficient C code module with accompanying Application Program Interface (API) that will receive streamed data from RHINO and provide data marshalling facilities to gather and dispatch blocks of data for further processing using C/C++ routines.

The methodology followed in this research project involves investigating real-time streaming techniques using User Datagram Protocol (UDP) packets, furthermore, investigating how GNU Radio high-level SDR development framework can be integrated into the real-time data acquisition systems such as in the case of this project with RHINO. The literature for real-time processing requirements for the streamer framework was reviewed. The guidelines to implement a high performance, low latency and maximum throughput for streaming will consequently be presented and the proposed design motivated.

The results achieved demonstrate an efficient data streaming system. The objectives of implementing RHINO data acquisition system through integration with standard C/C++ code and GNU Radio were satisfactorily met. The system was tested with real-time Radio Frequency (RF) demodulation. The system captures a pair of an In-phase/Quadrature signal (I/Q) sample at a time, which is one packet. The results show that data can be streamed from the RHINO board to GNU Radio over GbE with a minimum capturing latency of 10.2µs for 2^0 packet size and an average data capturing throughput of 0.54 Mega Bytes per second (MBps). The captur-

ing latency, in this case, is the time taken from the time of the request to receiving the data. The FM receiver case study successfully demonstrated results of a demodulated FM signal of a 94.5 Mega Hetz (MHz) radio station.

Further recommendations include making use of the 10GbE port on RHINO for data streaming purposes. 10GbE port on RHINO can be used together with GNU Radio to improve the speed of the RHINO streamer.

ACKNOWLEDGEMENTS

I would like to gladly express my gratitude to the following people who assisted towards successful completion of this research project:

- **Dr. Simon Winberg**, my UCT supervisor, for his guidance, and encouragement during the entire research project.
- **Mr. Lerato Mohapi**, my UCT co-supervisor, for his consistent encouragement and assistance with the research project.
- **Prof. Michael Inggs**, for his suggestions concerning the project.
- **Square Kilometer Array - South Africa**, for financial assistance throughout this research project.
- **UCT colleagues** in the RRSG and SDRG groups for their help and suggestions with regards to challenges I came across in the research project.
- **My family** for the support they have given me, the encouragements when I wanted to give up.

CONTENTS

Abstract

Acknowledgements **iii**

Contents **iv**

List of Figures **viii**

List of Tables **x**

List of Abbreviations **xi**

Nomenclature **xiv**

1 Introduction **1**

1.1 Background 1

1.2 Problem Description 3

1.2.1 Significance of the Study 3

1.3 Focus 4

1.4 Objectives 4

1.5 Scope and Limitations 5

1.6 Methodology Overview 5

1.7 Thesis Overview 7

2 Literature Review **10**

2.1 Software Defined Radio 10

2.1.1 Advantages and Disadvantages of SDR 13

2.1.2 Signal Processing Architectures in SDR 14

2.1.3 Digital Baseband Signal Processing 17
2.2 Overview of the RHINO platform 21
2.2.1 Hardware Architecture 21
2.2.2 DSP Firmware for RHINO 22
2.3 Data Transmission Protocols 23
2.4 Buffering and Data Access Patterns 26
2.4.1 Circular Buffer 26
2.4.2 Single Buffer 26
2.4.3 Double Buffer 27
2.5 Comparison of SDR based Software Tools 27
2.6 Overview of GNU Radio 29
2.6.1 Out of Tree Modules 31
2.6.2 GNU Radio with RTL-SDR 33
2.7 Summary 34

3 Methodology **36**
3.1 Overview 36
3.2 User Requirements and Constraints 37
3.2.1 Establishing User Requirements and Constraints 37
3.2.2 Synthesising System Requirements from Data Collected 38
3.2.3 User Requirements 39
3.2.4 System Constraints 40
3.3 System Design Specifications 40
3.4 Hardware Design Specifications 41
3.4.1 Hardware Specifications 41
3.4.2 Experimental Set-up 42
3.5 Software Design Process 43
3.5.1 Rhino Application Deployment and Activation Process 43
3.5.2 C/C++ Software Design Process 44
3.5.3 GNU Radio UDP Block Software Design Process 45
3.6 System Evaluation Process 45
3.6.1 System Connectivity and Capturing 47
3.6.2 Latency and Throughput 48
3.6.3 Reliability 48

3.7 Summary 48

4 System Design **50**

4.1 Architecture of the RHINO Streamer 50

4.2 Implementation of the C/C++ Program 51

4.3 Implementation of RHINO UDP Block 52

4.3.1 UDP Source Block 53

4.4 Configuring the RHINO Firmware 57

4.4.1 Instantiation of I/O Peripherals 57

4.4.2 The NCO and DDC Cores 58

4.5 Experiments Design 59

4.5.1 FPGA NCO Experiment 59

4.5.2 RHINO GNU Radio UDP Source Block Test Design 59

4.6 Case Study Design 60

4.7 Summary 61

5 Results and Discussion **62**

5.1 Evaluation Methodology 62

5.2 System Connectivity and Data Capturing Tests 63

5.2.1 Preliminary experiments with GNU Radio 63

5.2.2 FPGA NCO with C++ program Test 63

5.2.3 RHINO UDP Source Block Test 64

5.3 Latency and Throughput Tests 65

5.4 Reliability Tests 69

5.5 Case Study: SDR FM Receiver 70

5.6 Summary 73

6 Conclusions and Further Work **74**

6.1 Conclusions 74

6.1.1 Systems Requirements 74

6.2 Recommendations For Further Work 76

A RHINO Streamer User Requirements Questionnaire **83**

B QT C++ Code **89**

C Chirp Signal **99**

C.1 chirp.cpp 99

C.2 chirp.h 102

D RHINO UDP source block for GNU Radio **104**

D.1 RHINO-Source.cpp 104

D.2 RHINO-Source.h 109

E Creating OOT Modules **113**

E.1 Creating a new module 113

E.2 Creating cmake 114

E.3 Creating XML File 114

E.4 Installing The Block in GRC 114

E.5 Deleting a Block From Own Tree 114

LIST OF FIGURES

1.1 Block Diagram of an RF receiver [5] 2
1.2 Spiral Methodology 6

2.1 An ideal SDR architecture 12
2.2 A realistic SDR architecture 12
2.3 FPGA generic architecture with I/O Blocks and Logic Blocks 16
2.4 Generic SDR architecture with baseband signal 17
2.5 DDC translating IF signal to baseband signal [25] 18
2.6 CIC Filter - (a) Differentiator (b) Integrator [25] 19
2.7 RHINO high-level block diagram [40] 21
2.8 UDP packet 25
2.9 TCP packet 25
2.10 Circular buffer data structure [47] 26
2.11 Single buffer data structure 26
2.12 Double buffer data structure 27
2.13 Colour codes for data output/input types on GRC 30
2.14 Chirp Signal Source test on the sending side. 32
2.15 Chirp Signal Source on the receiving side. 33
2.16 Flow Diagram for a GNU Radio FM Receiver using RTL-SDR Dongle 34
2.17 FM Receiver Plot Results for the FM Receiver using GNU Radio and RTL-SDR dongle 35

3.1 Architecture of the system 41
3.2 Refined Architecture of the system 42
3.3 Logging into the RHINO interface 44
3.4 RhinoUDPserver and Mainwindow class design 45

3.5 GNU Radio UDP block for RHINO development stages 46

4.1 RHINO Streamer Architecture 51
4.2 Flow diagram showing the relationship between rhinoudpserver and Mainwindow class 52
4.3 RHINO Source Block For GNU Radio 54
4.4 RHINO Source Block Flow Graph 54
4.5 Receiving Data from RHINO 55
4.6 Source Block Data Access and Buffering Technique 56
4.7 FPGA NCO connecting with C++ program on PC 59
4.8 RHINO GNU Radio Source Block tested with GRC 60
4.9 FM signal receiver chain test on GNU Radio 60

5.1 RHINO NCO Plot on QT C++ Program 64
5.2 Flow Diagram For Testing RHINO GNU Radio UDP Source Block 65
5.3 GNURadio Real time plot from NCO data 66
5.4 GNURadio Real Time Frequency plot of sampled data at 9.108MHz 66
5.5 Latency Plot of Samples Captured on PC 68
5.6 Reliability Tests for Data Capturing on GNU Radio 69
5.7 Reliability Tests for Data Capturing on GNU Radio 70
5.8 Block Diagram chain for FM receiver on GNU Radio 71
5.9 FM signals directly from the RTL-SDR dongle and RHINO on GNU Radio. . . 71
5.10 FM demodulated signals for the RTL-SDR dongle and RHINO FM receiver tuned to 94.5MHz radio station. 72

LIST OF TABLES

2.1 OSI seven-layer model 11
2.2 Comparison of UDP to TCP 24
2.3 Comparison of SDR based software tools 28
2.4 Chirp Source Block Parameters 32

3.1 Functional Requirements 39
3.2 Non-functional Requirements 39
3.3 Constraints of the RHINO Streamer System 40

5.1 Parameters set on the RHINO UDP Source block 64
5.2 The Relation Between Number of Samples Captured and Time taken 67
5.3 The Average Time Taken to Capture 2^N packets 67
5.4 The Relation Between Size of Samples Captured and Time taken 68
5.5 Comparison between RHINO FM receiver and RTL-SDR dongle 72

LIST OF ABBREVIATIONS

- **ADC** – Analogue to Digital Converter
- **API** – Application Program Interface
- **ASIC** – Application Specific Integrated Circuit
- **BORPH** – Berkeley Operating system for Re-Programmable Hardware
- **CLP** – Configurable Logic Blocks
- **CPLD** – Complex Programmable Logic Device
- **DAC** – Digital to Analogue Converter
- **DC** – Direct Current
- **DDC** – Digital Down Converter
- **DFT** – Discrete Fourier Transform
- **DSP** – Digital Signal Processing
- **DSPs** – Digital Signal Processors
- **DUC** – Digital Up Converter
- **FFT** – Fast Fourier Transform
- **FM** – Frequency Modulation
- **FMC** – FPGA Mezzanice Card
- **FPGA** – Field Programmable Gate Array
- **FTW** – Frequency Tuning Word

- **GBE** – Gigabit Ethernet
- **GPP** – General Purpose Processor
- **GPMC** – General Purpose Memory Controller
- **GPU** – Graphics Processor Unit
- **GRC** – GNU Radio Companion
- **IC** – Integrated Circuit
- **IEEE** – Institute of Electrical and Electronics Engineers
- **I/O** – Input/Output
- **IP** – Intellectual Property
- **I/Q** – In-phase/Quadrature signal components
- **LAN** – Local Area Network
- **LSFR** – Linear Feedback Shift Register
- **LVDS** – Low Voltage Differential Signalling
- **MAC** –Media Access Control
- **NCO** – Numeric Controlled Oscillator
- **OFDM** – Orthogonal Frequency-Division Multiplexing
- **OOT** – Out Of Tree
- **OS** – Operating System
- **OSI** – Open System Interconnection
- **PC** – Personal Computer
- **PCI** – Peripheral Component Interconnect
- **RF** – Radio Frequency
- **RFFE** – Radio Frequency Front End
- **RHINO** – Reconfigurable Hardware INterface for computing and radiO

- **ROACH** – Reconfigurable Open Architecture COmputing Hardware
- **RSIGR** – RHINO Streaming Interface for GNU Radio
- **Rx** – Receive
- **SDR** – Software Defined Radio
- **SDRAM** – Synchronous Dynamic Random-Access Memory
- **SKA** – Square Kilometre Array
- **TCP** – Transmission Control Protocol
- **Tx** – Transmit
- **UDP** – User Datagram Protocol
- **UHD** – Universal Hardware Driver
- **USB** – Universal Serial Bus
- **USRP** – Universal Software Radio Peripheral
- **VCO** – Voltage Controlled Oscillator
- **VoIP** – Voice over Internet Protocol
- **VHSCI** – Very High Speed Integrated Circuit
- **XML** – eXtensive Markup Language

NOMENCLATURE

- **Analogue to digital Converter (ADC):** an electronic device that converts data from its analogue format to its digital form.
- **Berkeley Operating system for Re-Programmable Hardware (BORPH):** An Operating System for FPGA-based reconfigurable computers.
- **Digital to analogue Converter (DAC):** an electronic device that converts data from its digital format to its analogue form.
- **FPGA Mezzanice Card (FMC):** is an ANSI?VITA standard that defines the input/output mezzanine module connecting to an FPGA or a device with re-configurable input/output capability.
- **Field Programmable Gate Array (FPGA):** an integrated circuits with programmable logic blocks, interconnection network and a set of I/O cells which can programmed for a specific fuction.
- **Low Voltage Differential Signalling (LVDS):** a standard that specifies digital data characteristics.
- **Out Of Tree(OOT MOdule):** GNU Radio component that allows a user to build blocks that are not within the GNU Radio source tree. The module extends the functionality of the software.
- **Reconfigurable Hardware INterface for computing and radiO(RHINO):** an FPGA based board that provides an affordable SDR platform easy to use.
- **Reconfigurable Open Architecture COmputing Hardware(ROACH):** an FPGA based board with similar features with that of the RHINO board.
- **Universal Hardware Driver (UHD):** The USRP hardware driver provided by Ettus Research to be used with the USRP product family.

- **Universal Software Radio Peripheral (USRP):** hardware platform designed and sold by Ettus Research for purposes of software defined radios implementations.
- **Very High Speed Integrated Circuits Hardware Description Language (VHDL):** a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as FPGA.

CHAPTER 1

INTRODUCTION

This project is about the implementation of a signal processing software framework based on the RHINO. The project presents a tool-based solution for doing SDR Digital Signal Processing (DSP) on a PC where RHINO is used as a signal acquisition and data streaming peripheral in real-time. One of the critical elements of real-time data streaming is to keep the data moving consistently with minimum latency and maximum throughput [48]. This project aims to achieve this, moreover, the purpose of this project is to be able to perform radio signal processing on an SDR software framework. The approach of performing radio signal processing on software is in contrast to the classical approach of processing signals in hardware. An SDR approach provides a number of benefits including reducing development time by reusing existing SDR tool-based software for quickly developing system prototypes.

1.1 BACKGROUND

The concept of SDR was first introduced in the late 1970s [22], it was only known to be applicabe to the defence communication until the 1900s. Moreover, the high cost and low computing power in FPGA and Digital Signal Processors (DSPs) contributed to the restriction in the commercial implementation of SDR technology. However, over the years, SDR has proved to hold great promise in the field of wireless communication [23], with the introduction of Graphics Processor Units (GPU) in personal computers which can perform digital signal processing operations needed for the SDR-based system.

RHINO is a signal processing platform that has been used at University of Cape Town with a number of collaborators over the past years as an educational experimental platform in SDR applications, as well as to prototype radio and radar systems. Many projects in the development and support of RHINO have been successfully completed in the previous years [40] [6] [30] [7]. The study is supported by the MeerKat Digital Signal Processing (DSP) program at UCT since

2011.

One of the recent and relevant projects to the project at hand is titled: RHINO Software Defined Radio processing blocks [53]. The project was aimed at developing RHINO FPGA Intellectual Property (IP) cores and some of the cores developed in that project are the Input/Output (I/O) cores which will be used extensively in this project. The I/O cores enable communication between the RHINO FPGA and an external peripheral through the Gigabit Ethernet. In addition to integrating RHINO with standard C/C++ code to support application development, GNU Radio is used in this project as an SDR software tool for signal processing.

Figure 1.1 shows a block diagram of a typical RF receiver from the RF stage to the baseband stage. In the RF stage, the RF filter selects the frequency bandwidth for the signal attenuating a range of frequencies outside the bandwidth. The Low Noise Attenuator (LNA) limits the noise bandwidth to prevent the aliasing of the wideband noise into the signal band [42]. Before the first mixer in the architecture, an image rejection filter is needed to eliminate the image frequency that will result in the same output IF with the RF input. The mixer down-converts the RF signal to IF. More details on RF receiver will be covered in Chapter 2.

In the IF stage of Figure 1.1, the first component is the IF filter which is used for channel selection and then the signal is amplified before it is sent to the ADC. The ADC converts the analogue signal to a digital signal. The next stage is the digital baseband stage, and the stage is meant for the Digital Down Conversion (DDC) of the IF signal to a baseband signal. DDC is performed by multiplying the input signal with amplitude values of sine and cosine from a look-up table [26].

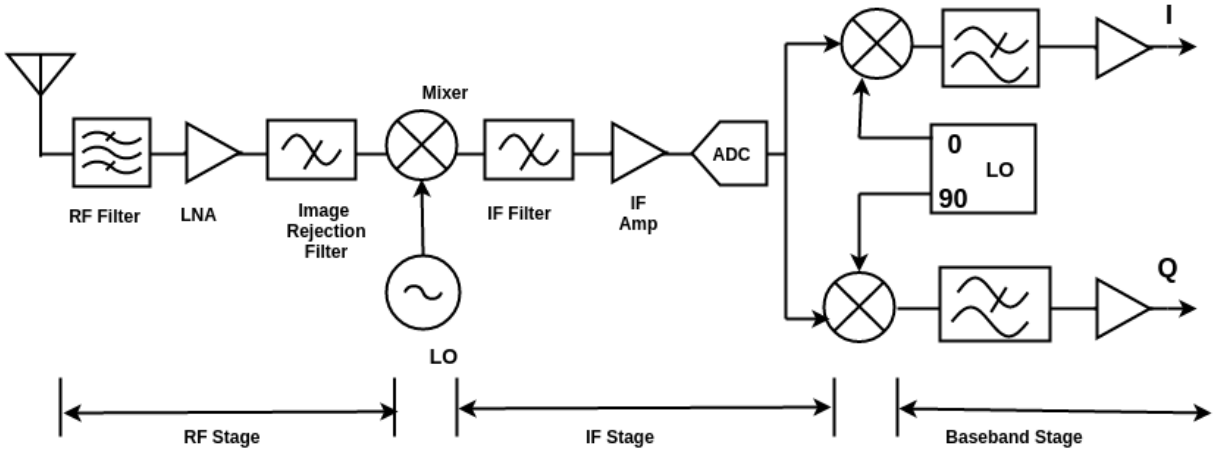


Figure 1.1: Block Diagram of an RF receiver [5]

Baseband stage as shown in the RF receiver architecture in Figure 1.1 produces a baseband signal which can later be processed. The mixers at this stage translate the signal coming into a

baseband signal. Signal processing architectures include GPPs, FPGAs, and DSPs, moreover, there exist SDR based software tools, such as GNU Radio, and a relatively new platform OpenSDR [31], which can implement baseband signal processing.

1.2 PROBLEM DESCRIPTION

The project that this dissertation reports on aims to respond to the challenges in developing a supporting framework for integrating the RHINO platform with a PC host in order to facilitate the development of SDR applications built using a hybrid RHINO/multicore PC system. The main problem with the use of RHINO is that RHINO is difficult to use. It takes users long to figure out how to get data from it. At the moment, there are no useful tools that it integrates with easily. Users keep going through the same lengthy learning and code development processes, even though they may just want to use it as a source of sampled radio data.

GNU Radio is one of the popular software development toolkits which provides signal processing blocks for the implementation of software radios. It can be used both as a simulation tool and together with an external RF hardware such as RHINO. This will be elaborated on in Section 2.6. Unlike Labview or MATLAB Simulink, GNU Radio is open source, free of cost to use by anyone with little to no programming knowledge using pre-written blocks [13]. Section 2.5 will go into details with regards to comparing different software tools for SDR implementations. GNU Radio can be used with hardware peripherals such as Universal Software Radio Peripheral (URSP) modules, a low-cost platform for SDRs [28]. However, there has not been such support for RHINO prior to this project. User defined GNU Radio blocks can be implemented using the Out Of Tree (OOT) module feature that GNU Radio offers. The use of this feature makes it possible for GNU Radio to be integrated with other peripherals, such as RHINO.

This project aims to provide not only a solution for integrating RHINO with GNU Radio framework for data streaming but also developing a C/C++ interface code that can be extended for use with RHINO in other applications. The project involves streaming data from RHINO over 1Gb Ethernet onto the host PC for signal processing. The data will be streamed and captured using GNU Radio and an independent C/C++ program at high speed, minimum latency, minimum data loss and processed in real time.

1.2.1 Significance of the Study

UCT has invested a large capital in RHINO, sampling hardware and RF front-end components which are used in the UCT SDRG for teaching purposes and research. As mentioned in Section 1.2, without proper tools, RHINO is very challenging to use. The significance of this project

is to enable a more effective use of RHINO for SDR applications prototyping. Furthermore, the study aims to assist members of the SDRG to make better use of RHINO and investment that was put in. The study is important in the improvement of the usability of the platform and previously developed supporting resource, such as FPGA-based cores that have been provided that can provide various pre-processed data streams for inspection and processing on the PC. The project fits in with practices commonly used in the group that involve use of OCTAVE/-MATLAB, C/C++ programming and GNU Radio.

1.3 FOCUS

The focus of this project is on the integration and testing of RHINO with a host PC. GNU Radio will be used together with RHINO as a software framework to build SDR applications. The integration of RHINO with GNU Radio will be achieved by implementing a GNU Radio block which will act as a data acquisition tool from RHINO to the host PC over Gigabit Ethernet. The block is expected to communicate well with other existing blocks on GNU Radio for purposes of signal processing. Furthermore, it is expected to be parametrizable and user-friendly. The project also focuses on developing a C/C++ based code that interfaces with RHINO to stream data. This code can be further developed and extended to interface RHINO with other C based applications.

The tests will focus on the streaming of data from RHINO to the host PC in real-time, which involves streaming data to a C/C++ program and GNU Radio independently. The connectivity and inter-communication of the GNU Radio blocks with RHINO will also be tested. Furthermore, the focus will be on testing the system based on the following factors: latency, data loss, and throughput. A case study will focus on the real world SDR application which will thoroughly test the streaming of data and the connectivity of RHINO with GNU Radio.

1.4 OBJECTIVES

The main objective of this project is to implement a software tool based solution for SDR-DSP on a PC with RHINO as a real-time data acquisition hardware. The main objective will be achieved by achieving the following sub-objectives:

- Build upon existing firmware for the RHINO to provide a source of sampled signal data.
- Develop a C/C++ program that can interact with RHINO, which can also be extended such that RHINO communicates with other C based applications.
- Design and implement a C/C++ based GNU Radio block for interfacing with RHINO.

- Test the C/C++ program with RHINO by streaming data in real-time.
- Test the SDR framework (GNU Radio) for efficiency with regards to connectivity with RHINO and inter-communication within the software.
- Evaluate the system's data streaming efficiency by testing the following factors: latency, throughput and data loss.
- Structure test cases to evaluate the efficiency of the RHINO GNU Radio block on the GNU Radio framework.

1.5 SCOPE AND LIMITATIONS

The scope of the project is limited to the integration of RHINO with a host PC for data streaming and signal processing. It should be noted at this point that the data streaming is only limited to the transmission chain, that is, streaming data from RHINO to the PC and not from the PC to RHINO. The project will involve the development and testing of a RHINO data acquisition block on GNU Radio. Performance tests will be structured around this area of study, furthermore, the tests will be structured around the ability of RHINO to communicate with the host PC over Gigabit Ethernet using a suitable network protocol. The RHINO board has both the 1GbE and 10GbE I/O ports, however, the scope of this project is limited to streaming data over the 1GbE port.

The case study for this project will be focused on thoroughly testing the integration using an SDR real world application. Communication of the RHINO board with other peripherals through the 1GbE port is possible because of the IP cores implemented on RHINO FPGA as part of an MSc project here at UCT [53]. One of the IP cores that are essential to the project at hand is the I/O cores namely: Gigabit Ethernet core and the ADC-DAC core interface core. These cores will facilitate the RHINO data streaming ability which is the main objective of this project.

1.6 METHODOLOGY OVERVIEW

The methodology of this project has generally followed the spiral model [4]. Each spiral of the circle as shown in Figure 1.2 begins at the user requirements then design specification, design of hardware and software and finally the evaluation process phase respectfully.

The system is required to be a real-time SDR transmission platform and this will be achieved with the use of a firmware ready 1GbE FPGA data acquisition sub-system. The ADC IP core

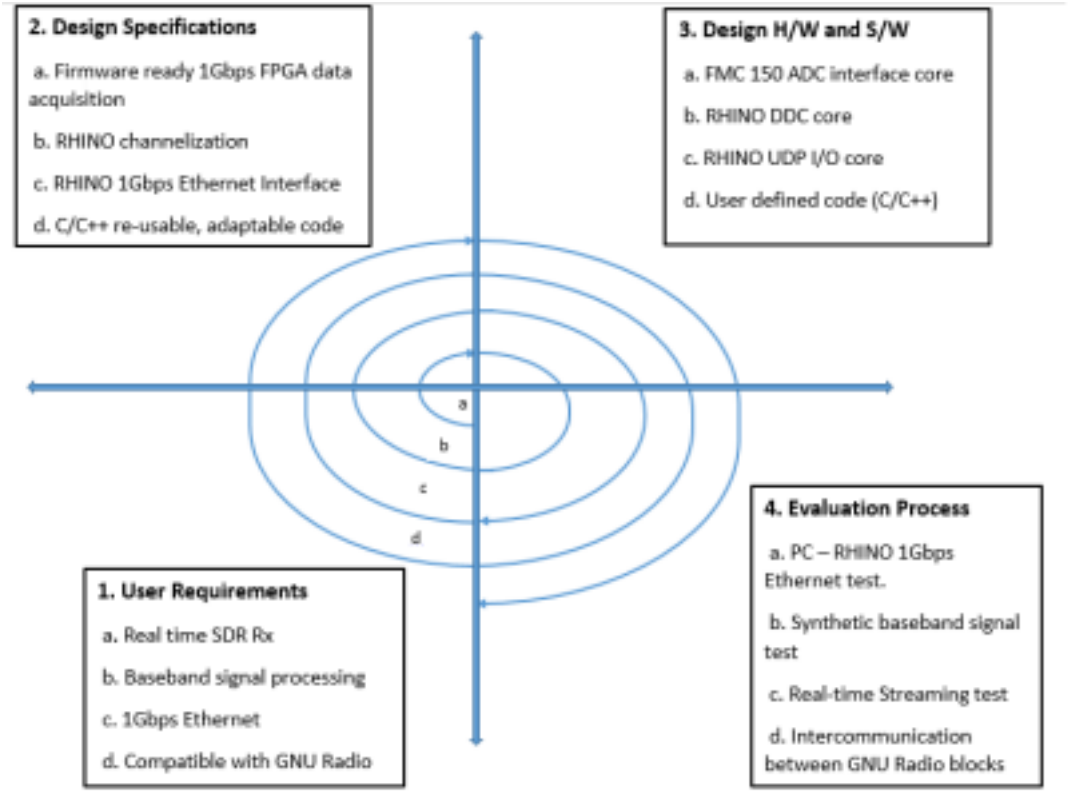


Figure 1.2: Spiral Methodology

interfaces with the FMC 150 ADC hardware to link the RF section of the system with the RHINO FPGA (IF section) which in turn links with the PC through a 1GbE connectivity. The evaluation for this user requirement is a connectivity test between the PC and RHINO over a 1GbE User Datagram Protocol (UDP) connection.

Baseband signal processing which is implemented on the PC is possible due to the channelization firmware on RHINO. The implementation of DDC core on the FPGA translate the IF signal into a baseband signal and transmitted to the receiving PC. The evaluation process was conducted by using synthesized baseband signal from the RHINO FPGA to test the usability of the data and capturing ability of the system on the receiving side in alignment with the objectives of the systems.

The user requirement for the RHINO streaming interface for GNU Radio is to use a 1GbE connectivity. RHINO has a 1GbE and 10GbE port which can both be used to connect to an external peripheral. The system is required to use the 1GbE, and this is achieved with the design and implementation of a UDP 1GbE core which was tested with the real time stream test from the RHINO to the connecting PC.

The final system requirement is compatibility with an SDR tool, GNU Radio. A user-defined

C/C++ reusable and adaptable UDP server/client class was used to build into a GNU Radio OOT module. This module was used to capture synthesized FPGA data from RHINO on the PC, evaluation process was based on the intercommunication between the RHINO GNU Radio block and the source tree GNU Radio blocks. Furthermore, the evaluation process tests the latency, throughput and reliability of the communication as iterated in Section 1.4 .

1.7 THESIS OVERVIEW

This section highlights a brief overview of each chapter in this write-up; the remainder of this dissertation comprises six chapters.

Chapter 2 highlights the extensive literature reviewed concerning SDR systems and the RHINO platform. The chapter starts off with an introduction to the concept of SDR with emphasis on the advantages outweighing the disadvantages. Signal processing architectures in relation to SDR are highlighted, with mentions of GPPs, DSPs, and FPGAs. These architectures are compared with one another and this section also includes an overview of their relevance in SDR systems. Digital baseband signals processing is an important part in SDR, signals are converted from RF to baseband signals in an SDR system, for this reason, this chapter also highlights the two main processing techniques important in this project, namely digital down conversion and frequency modulation. This section gives an elaborate overview of these processing techniques.

RHINO is then introduced as the data acquisition platform in this project, the hardware, and software that make up the board. RHINO is used in this project as a firmware ready platform due to the readily available IP cores implemented as part of one of the projects here at UCT. The chapter, therefore, provides literature of these IP cores, however, those that are only relevant to this project, that is, DDC core, ADC/DAC core and UDP/IP core. Thereafter, data transmission protocols, UDP and TCP are discussed. The two are compared in Table 2.2 to establish the best protocol to data transmission between RHINO and PC. The different techniques of buffering and accessing data are discussed as an introduction to the techniques used for the RHINO UDP source. A comparison among SDR software tools that can be used with RHINO follows leading to a choice of the best option. An overview on GNU Radio introduces the concept of out of tree modules to extend the functionality of the software by creating own blocks with own functionality. This section further continues to build on OOT modules by highlighting source and sink blocks on GNU Radio which is the type of blocks that will be used to facilitate the data transmission from RHINO to the PC. The chapter is finalised with a tutorial on using GNU Radio with an RTL-SDR dongle.

Chapter 3 presents the methodology for this project. The chapter provided guidelines that are

used for the design process by highlighting the following: user requirements and constraints, design specification, hardware and software design process and finally evaluation process. The user requirements are divided into two sections; functional and non-functional requirements. The design specification provides an overview of the architecture of the system. This chapter further goes into details of the software design process highlighting the reasons behind the choice of data transmission protocol used for this project. Furthermore, a figure which depicts the development of the GNU Radio block for RHINO follows.

This chapter thereafter provides an overview of the hardware used in this project namely, a PC and the RHINO board, the specifications of the devices are highlighted. The evaluation process evaluates both the hardware and software. Hardware evaluation highlights the hardware experimental set up while the software evaluation demonstrates the software experimental set-up evaluating GNU Radio. The evaluation is based on the data transfer capabilities of GNU Radio from a file source to the other connected PC. The experiment set-up is of the client/server orientation. The data used was captured from RHINO Numeric Controlled Oscillator (NCO) using Wireshark. The software evaluation also tests the efficiency of an independent C/C++ server/client classes which the RHINO GNU Radio will be built from. This is in terms of latency and maximum throughput that can be achieved.

Chapter 4 describes the design of the RHINO streaming interface for GNU Radio. The RHINO streamer as depicted in Figure 2.7 shows all the different components of the system highlighting the essential IP cores in the FPGA, the structure of the data from the RHINO to GNU Radio installed on the PC. The figure shows the two blocks that facilitate the communication between RHINO and GNU Radio over GbE. The chapter then goes into details about the implementation of the RHINO UDP module, source block. This section highlights the structure of the block which includes the parameters that are set by the user. The process flow of the functionalities of the block is also highlighted in this section demonstrating the different actions taken in receiving data from RHINO.

Figure 4.4 demonstrates the data flow in a RHINO source block showing extraction of data, buffering and transfer to other GNU Radio blocks. The configurations of the RHINO firmware will be discussed highlighting instantiation of the I/O peripherals, NCO and DDC cores, finally configuring the channelization firmware parameters. The chapter is finalised with the tests design for RHINO UDP source block, explaining the experiments carried out to test it and the case studies designed to test the system as a whole. The code implemented can be viewed in the Appendix section of the report.

Chapter 5 presents the results based on the implementation of the design tests in chapter 4.

The evaluation will be based on the following factors: data capturing and connectivity, latency and throughput,and finally the reliability of the system. The RHINO UDP source block will be tested based on these factors and the success of the experiments determined. The results will further be interpreted and any discrepancies explained. Conclusion on these test will be highlighted in the following chapter.

After the interpretation of the results **Chapter 6** provides conclusions and recommendations for future work.

CHAPTER 2

LITERATURE REVIEW

This chapter will focus on the relevant literature reviewed to support the undertaking of this project. It will begin by introducing the concept of SDR in Section 2.1, the history involved and thus the evolution of the concept to this day. SDR hardware which includes modern SDR architectures and SDR software will also be covered, specifically the RHINO architecture and firmware. Furthermore, applications of SDR will also be underlined in this chapter. An overview of RHINO platform will follow in Section 2.2 which will highlight the hardware architecture and DSP firmware for RHINO. The literature on the data transmission protocols considered will follow in Section 2.3, a comparison will be made so as to come to a conclusion of the best data transmission protocol for this project. The chapter will thereafter present a comparison of the commonly used SDR software tool at UCT and conclude on the best option for this project. Finally, the chapter will introduce GNU Radio in Section 2.6, highlighting how the SDR based software has been used with other hardware. This section will then introduce the concept of out of tree modules (OOT) for GNU Radio. Then a summary of the chapter will follow in Section 2.7.

2.1 SOFTWARE DEFINED RADIO

With the ever increasing need for means of people to communicate effectively in a variety of ways such as data communication, broadcast messaging, voice and video communication, the modification of radio devices to suit these needs does not come as a surprise. Historically, radio devices were designed to only support a specific waveform, that is, they had a specific function. However, over the years this has come to a change [17]. Radio devices are now multi-purpose devices which can accommodate different waveforms; this is due to the flexibility that SDR bring into the picture.

The Wireless Innovation Forum working in collaboration with the Institute of Electrical Engi-

neering (IEEE) P1900.1 group defines SDR as *Radio in which some or all of the physical layer functions are software defined* [12] [36]. To elaborate on each term individually:

- Radio is any device with the ability to transmit or receive signals in the RF part of the electromagnetic spectrum wirelessly in order to transfer information from point A to B. Every day devices that contain radio receiver/transmitter include cell phones, television sets, personal computers, garage door openers and many more.
- Physical layer is one of the seven layers defined by the Open System Interconnection (OSI) model. The model defines the networking framework that implements protocols in these layers. Table 2.1 [17] shows the seven layers elements.

	Data Unit	#	Name	Function
Host layers	Data	7	Application	Network process to application
		6	Presentation	Data representation and encryption
		5	Session	Interhost communication
	Segment	4	Transport	End-to-end connections and reliability
Media layers	Packet	3	Network	Path determination, logic addressing
	Frame	2	Data Link	Physical addressing
	Bit	1	Physical	Media, signal and binary transmission

Table 2.1: OSI seven-layer model

An ideal SDR architecture as shown in Figure 2.1 consists of a signal entering the receiver at the antenna, it is then sampled, digitized and processed in the digital signal processing stage. The ideal architecture is not feasible and poses the following challenges that can not easily be overcome:

- In this architecture, the antenna would have to be manually tuned dynamically to the specific band of frequencies required. Since antennas are mechanical structures, tuning them would prove to be very difficult. This also applies to the filters required to be implemented in the antenna.

- The entire band captured by the antenna would have to be digitized by the ADC. This would be frequencies ranging from 1 MHz to 60 GHz, which is the highest frequency for terrestrial communication [17]. According to Nyquist's criterion, sampling frequency has to be twice that of the highest frequency. At the moment, there is no ADC that can sample at 120GHz.

For the reasons mentioned above, this architecture would not be the best one for an SDR architecture.

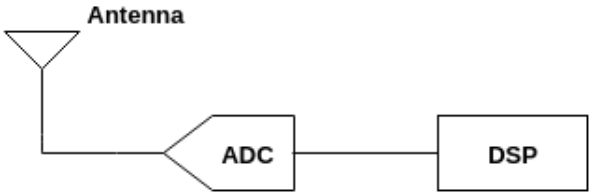


Figure 2.1: An ideal SDR architecture

A more realistic architecture is shown in Figure 2.2. This architecture uses components such as a bandpass filter for selection of a range of frequencies required for the application, a down converter, and a low pass filter before the addition of the ADC. All these components constitute as RF Front End (RFFE). This structure provides a more stable, reliable and robust approach.

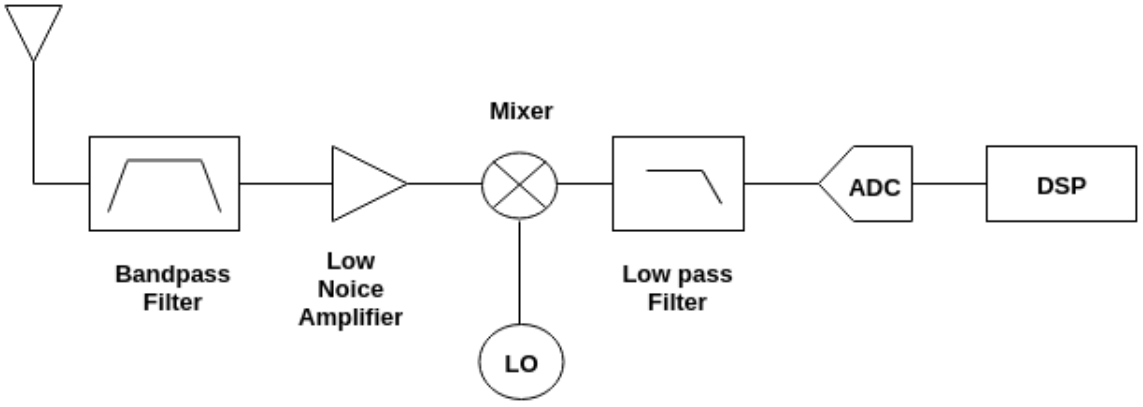


Figure 2.2: A realistic SDR architecture

RFFE makes it possible to work in IF, the added components in Figure 2.2 architecture contributes to the down conversion of high-frequency signals into baseband signals. The digital baseband processing, therefore, becomes feasible.

2.1.1 Advantages and Disadvantages of SDR

As mentioned earlier in this chapter, the reason the communication world is moving rapidly into SDR is because of the many benefits it brings to the field. One of the major issues is the expenses involved in the designing and production of transceivers limited to a specific functionality. The following points iterate the advantages of SDR in communications systems.

Advantages of SDR

- Less analogue parts in SDR systems.
- A large RF spectrum can be analysed.
- Provides portable devices.
- SDR promises the possibility of cognitive radio.
- Interoperability

The essence of SDR is to remove as much of analogue parts and do them in software. This provides a more flexible, adjustable and multi-purpose system, processing data such as modulation/demodulation can be done with ease. With a large spectrum analysed, quantities such a signal strength, interference patterns can easily be studied. This is very useful in areas of research.

SDR provides portable devices which do not require replacements whenever a new feature needs to be added to the system. With SDR, a simple software update is all it takes to improve on the features or add-on to the system. Various experiments can be conducted without building new circuits. This becomes especially significant when it comes to systems with hardware that is very expensive to replace, systems that are not necessarily easy to replace such as space radios. Moreover, for long life cycles radio, replacing hardware whenever a new feature is required would prove to be costly.

Cognitive radio is a radio system that is able to sense and predict its surrounding and adjust its functionality accordingly [9]. Most of the spectrum allocated for use for certain communication systems are not always in use at any given moment, sometimes the spectrum is used for just a few minutes. With cognitive radio, the spectrum could be ”borrowed” for such a particular time, and released when the need arises.

Interoperability has importance in a world where different wireless communication-based systems exist but cannot communicate with one another. One of the many attractions of cognitive

radio in SDR is the potential of many wireless communications systems to detect and establish a common communications pathway, accommodating incompatible radios [33].

Disadvantages of SDR

Even though there are many benefits of SDR, there is no technology without challenges. The following are some of the disadvantages of SDR in communication systems.

- Cost
- Power consumption
- Time and skills requirements.

The cost that comes with SDR is one of the major challenges. Function specific radios such as a garage door opener do not need the complexity that comes with SDR. In cases where an upgrade in a functionality of the system is not likely, there is no need to increase the cost of a system by changing it to an SDR-based system.

The power consumption associated with the DSP complexity and wide RF bandwidth in SDR is one other disadvantage. The signal processing device FPGA/GPP , which will be covered later in the chapter, uses ten times the power that Application Specific Integrated Circuit (ASIC) uses [17]. Wideband RFFE ADC/DAC devices required for SDR system uses up more power than their narrowband counterparts.

Designing SDR systems require a lot of time and energy dedication, furthermore, a special set of skills are required. Due to its complexity, testing these systems is not as straightforward as we would like it to be because a thorough testing would entail testing all the waveforms and combinations for any scenario. This would result in a lot of money and time invested, lack of thorough testing could result in a risk of failure of the system.

2.1.2 Signal Processing Architectures in SDR

As mentioned in the earlier sections of this chapter, an ideal SDR has some of its physical layer functionalities done in software. Additionally, the signal processing portion of the system is implemented on programmable devices. The following section reports on the hardware involved in the DSP part of an SDR system and some of the SDR based software.

Signal Processing Devices

DSP devices discussed in this section are as follows:

- General Purpose Processors
- Digital Signal Processors
- Field Programmable Gate Arrays

GPPs are microprocessors built for general-purpose computers such as personal computers and workstation desktop computers, some of the vendors include, Intel, ARM, and AMD. GPPs are designed to carry out a variety of operations in parallel. Although not initially designed for signal processing, they offer high-speed performances (high clock rates). However, their computational power has not been able to keep up with the rapid growth of the data [18]. GPPs' operations include fixed-point, floating-point and logic operations. As a result, this makes them ideal for some of the SDR functionalities. One of the advantages of GPP based SDR is the ease of programming functionalities and debugging, unlike in the case of an FPGA-based SDR. Moreover, SDR platform architectures based on GPPs or a combination of DSPs and GPPs are relatively cheap [52]. These are ideal for experiments in the laboratory as power consumption is not a concern.

In a GPP based SDR, the PC receives the baseband samples through either of the following interface standards: USB, Ethernet or PCI, into the buffer then onto the signal processing thread. The problem, however, is the latency challenges with regards to GPP based SDR, high performance is achieved only when the processor is working on a block of data rather than on each sample at a time [17]. Furthermore, the transmission of data from the hardware device to the host computer requires a high bus throughput communication. Also, physical layer and MAC layer protocols require real-time processing of strictly minimum time [52]. These and other factors, therefore, results in GPP based SDR prone to latency problems in terms of digital samples transmission.

Some personal computers have Graphics Processing Units (GPU) installed. GPUs can provide most of the digital signal processing operations required in an SDR system. Furthermore, GPUs can process a lot of data in parallel using its Arithmetic Logic Units (ALUs) without affecting the other applications running on the computing device [22]. These graphics units although initially designed to accelerate operations in 3 Dimension (3D), they include all the advantages of FPGAs and DSPs in signal processing.

DSPs are microprocessors optimized for digital signal processing tasks, operations such as digital filtering and Fourier analysis. DSPs can perform both fixed and floating-point operations. One of the major advantages of DSPs over GPP is the fact that they are very energy efficient,

more especially the fixed-point operations DSPs. However, DSP development environment is a lot complicated compared to that of GPPs, and so DSP developers are not as easy to find [22].

An ideal DSP SDR-based system would have to be paired with a GPP as DSPs do not implement network layer functionalities. Because of their low power consumption, DSPs are used in SDR systems that require low power for functionality.

FPGAs are Integrated Circuits (IC) that are configurable and programmable. They are made up of Configurable Logic Blocks (CLP) which are connected through programmable interconnects. These logic blocks can be configured to perform complex functions or simple logic operations such as AND, OR, etc. FPGAs are different from ASICs in the sense that they can be programmed for any functionality after manufacture, while ASICs are manufactured to perform a specific functionality and can not be changed. FPGAs have become very powerful with regards to reconfigurable hardware resources, they are typically used as accelerator devices because of their ability to process data in parallel [1]. Figure 2.3 shows a generic architecture of an FPGA.

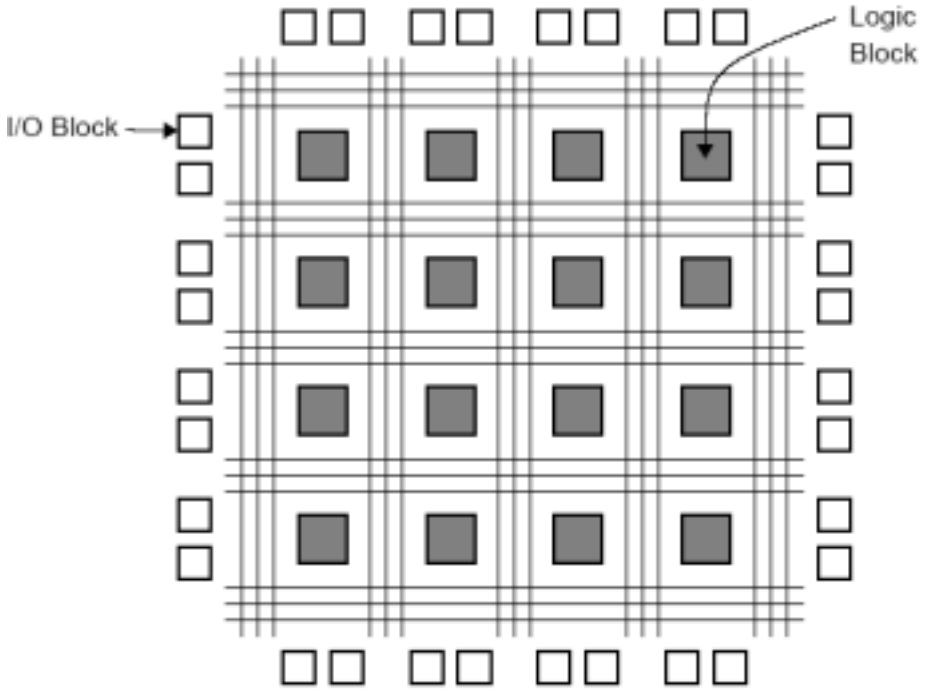


Figure 2.3: FPGA generic architecture with I/O Blocks and Logic Blocks

Because of their programmable ability, FPGAs can be used in a wide variety of applications. They are most desirable for implementing flexible radios. Furthermore, an FPGA-based SDR has adequate signal processing resources to support a wide range of waveform in one device [17]. They can be used in wireless communication systems for RF signals, baseband signals. An intellectual property core (IP) is a block of logic that is used in configuration and functionality

development of an FPGA. An implementation of an FPGA-based SDR could be a combination of FPGAs and GPPs usually on a PC. The fastest interconnect technology currently available for FPGA is PCI Express [1]. However, PCI Express comes with some setbacks. Besides being the most expensive interconnect technology, a PCI Express driver is necessary on the PC side and the FPGA board needs to be PCI Express enabled.

The transmission of data from the FPGA to the PC could be achieved with the combination of the Internet Protocol version 4 (IPv4) and UDP as a transport protocol. UDP requires less programming on the PC side (UDP programming interface readily available) and comparatively has high transmission rates. More information on UDP will be provided in the later stages of this chapter.

2.1.3 **Digital Baseband Signal Processing**

A baseband signal is defined as any signal with frequencies close to DC, a signal with bandwidth in the lower range of frequencies [45]. In an SDR architecture, the RFFE accepts analogue signal with a high frequency, the signal is passed through a bandpass filter and down-converted into an IF signal. The signal is passed through a mixer which then produces I/Q samples that pass through the low pass filter then into the ADC. The I/Q samples are what is referred to as baseband signal, and the samples are passed on to the signal processing stage. Figure 2.4 below iterates the process of getting a baseband signal from an RF signal in an SDR architecture.

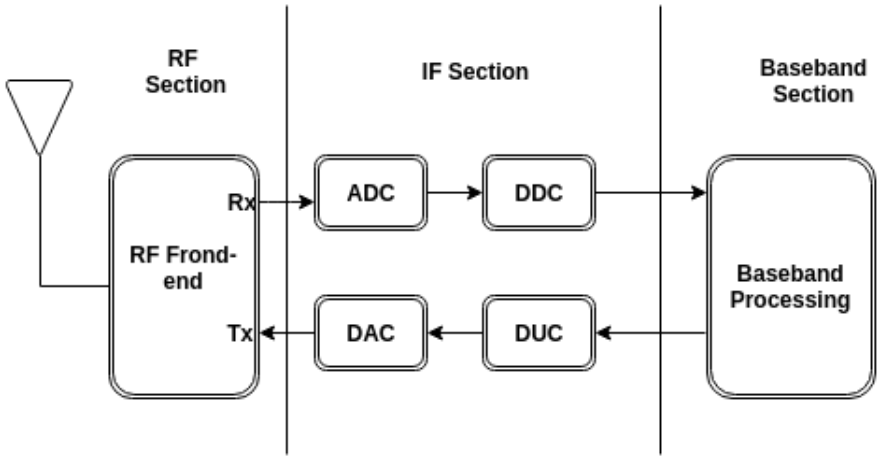


Figure 2.4: Generic SDR architecture with baseband signal

Digital baseband processing involves a lot of different algorithms, some of which can be implemented as IP cores in an FPGA. Some of the IP cores relevant to this project will be highlighted in the later stages of the report. With these FPGA IP cores, RHINO becomes a firmware ready data acquisition platform.

Digital Down Conversion

DDC shifts the digitized IF signal from its carrier down to the baseband signal, it reduces the amount of subsequent processing without loss of information carried by the IF signal. DDC can be implemented on FPGAs, ASICs or DSPs and can be classified into narrowband and wideband DDCs. There are three sections in a DDC architecture as shown in Figure 2.5, namely: NCO, digital mixer and decimating Low-Pass Filter (LPF).

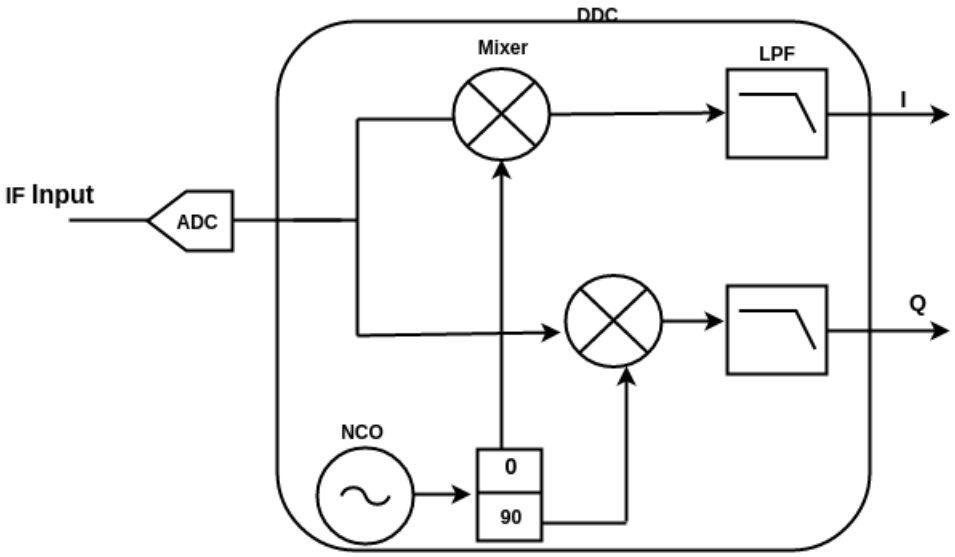


Figure 2.5: DDC translating IF signal to baseband signal [25]

- NCO - Direct Digital Frequency Synthesizer (DDS), calculates the new phase at the sampling frequency f_s with the phase advance defined by the tuning word. Tuning word is programmable with frequencies up to $f_s/2$ (Nyquist).
- Digital mixers - the output is two frequencies, the sum, and the difference frequency signals.
- Decimating LPF - often implemented as Cascaded Integrator Comb filters (CIC)

The CIC is made up a number of stages, each stage represents a pair of comb and integrator filters interconnected. A comb filter as shown in Figure 2.6(a) is a differentiator with a transfer function $H_c(z) = 1 - z^{-D}$, where D is a differential delay. This differential delay is operating at a lower sampling frequency of f_s/R, R is the down-/up-converter value. An integrator-filter as shown in Figure 2.6(b) is a single pole accumulator with a transfer function $H_I(z) = 1/1 - z^{-1}$, and operates at a higher sampling frequency of f_s.

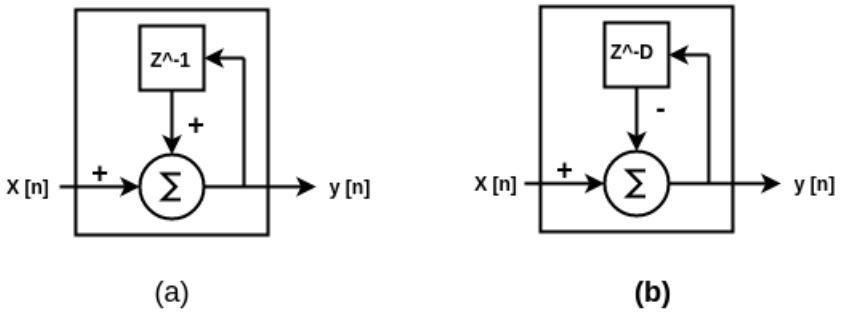


Figure 2.6: CIC Filter - (a) Differentiator (b) Integrator [25]

It can be concluded that in the case of down-conversion, an integrator starts sampling at a higher frequency, applies the transfer function then the integrator output enters the comb stage at a lower sampling frequency and applies the transfer function. Therefore, it follows that the up-conversion is the reverse.

Frequency Modulation

In wireless communications, modulation is the process of changing one or more properties of the data signal for transmission. Essentially, one of the properties of the carrier signal is altered by the information signal for transmission. To retrieve the information, the modulated signal goes through the process of demodulation.

There exist different types and techniques of modulation, each changing different properties of the carrier signal. Frequency Modulation (FM) is of much interest in wireless communication. Frequency modulation is defined as the type of modulation where the frequency of the carrier signal is varied in proportion to the amplitude of the baseband signal (message signal).

The simplest way to generate FM signal is to apply the message signal to the Voltage Controlled Oscillator (VCO) to get an output signal. The frequency of the modulated signal is expressed as follows:

$$f_i = f_c + K_{VCO} m(t) \tag{2.1}$$

Where K_{VCO} is the voltage to frequency gain of the VCO given in Hz/V, the product of the gain and the message signal $K_{VCO}m(t)$ is the instantaneous frequency deviation. It follows that the instantaneous phase of the output signal is as follows:

$$\theta_i(t) = 2\Pi f_c + 2\Pi K_{VCO} \int_0^t m(t)dt \tag{2.2}$$

assuming the initial phase is zero. Hence the output signal is given as:

$$x_{FM}(t) = A_c cos[2\Pi f_c + 2\Pi K_{VCO} \int_0^t m(t)dt] \tag{2.3}$$

It can be seen from Equation 2.3 the amplitude of the output signal A_c is constant and affected by the message signal. Also the FM signal has a non-linear relationship with the message signal. In order to estimate the bandwidth of the FM signal, a single-tone message shown in Equation 2.4 is used:

$$m(t) = A_m cos(2\Pi f_m t) \tag{2.4}$$

Substituting equation 2.4 into equation 2.3 yields the following:

$$x_{FM}(t) = A_c cos(2\Pi f_c t + \beta sin(2\Pi f_m t) \tag{2.5}$$

where β is referred to as the modulation index and is expressed as : $\beta = \frac{\Delta f}{f_m}$ and Δf is referred to as the frequency deviation expressed as : $\Delta f = K_{VCO} A_m$

The number of side-bands in the spectrum of an output signal for a single-tone message is a function of β and can be expressed in terms of the n^{th} order Bessel functions as follows:

$$x_{FM}(t) = A_c \sum_{n=-\infty}^{\infty} J_n(\beta) cos(2\Pi(f_c + n f_m)t) \tag{2.6}$$

Taking the Fourier transform, the spectrum of the FM signal has magnitude coefficients as a function β and can be shown in the following equation:

$$x_{FM}(f) = \frac{A_c}{2} \sum_{n=-\infty}^{\infty} J_n(\beta)[\delta(f - f_c - n f_m) + \delta(f + f_c + n f_m)] \tag{2.7}$$

The number of side-bands and the associated magnitude for an FM signal can be in the Bessel function tables. [8]

2.2 OVERVIEW OF THE RHINO PLATFORM

Reconfigurable Hardware Interface for computatioN and radiO (RHINO) is an FPGA based board which has similar features as its counterpart Reconfigurable Open Architecture Computing Hardware (ROACH). In the past years, there have been multiple projects based on RHINO at the University of Cape Town under the umbrella Software Defined Research Group (SDRG) . The aim of the group is to develop skills in the field of SDR systems and to pursue innovations in specialised field of engineering.

RHINO is an open source software and hardware that provides an affordable SDR platform that is easy to use and learn to a broad audience. This FPGA board can be used both in teaching and research environment. The following section will highlight the architecture of RHINO, the OS it uses, peripherals and some of the DSP firmware developed which are relevant to this project.

2.2.1 Hardware Architecture

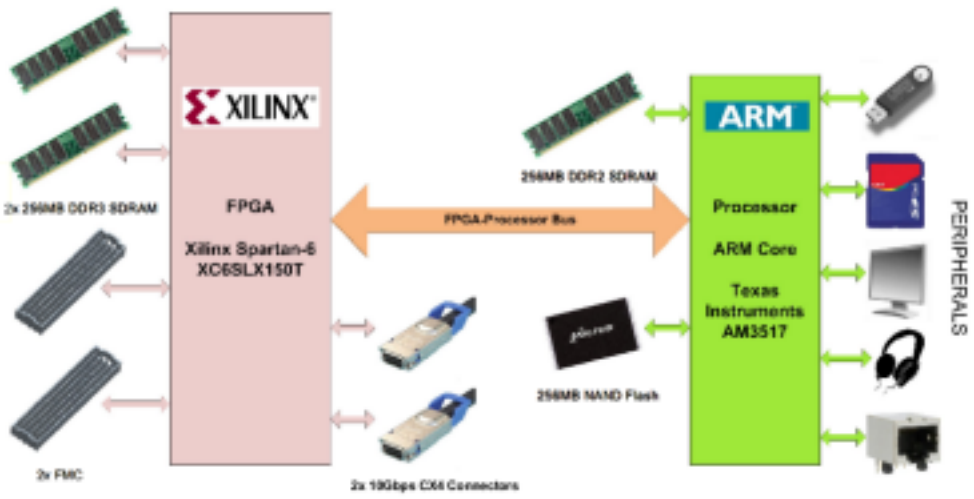


Figure 2.7: RHINO high-level block diagram [40]

A high-level block diagram of RHINO [40] is shown in Figure 2.3 indicating all the different components that make up the board. Further clarification of each component follows below:

- **FPGA** - Xilinx Spartan-6 FPGA with adequate logic resources. Supports a variety of peripherals to enable communication between the FPGA and the peripheral.
- **Processor** - AM3517 ARM Processor manufactured by Texas Instruments, supports Linux.
- **BORPH** - The processor run on Berkeley Operating system for Re-Programmable Hardware (BORPH). Allows the user to program the FPGA with a given design/configuration,

runs it as software process within Linux.

- **Ethernet** - There exists two Ethernet interfaces, 100Mbps which connects to the processor for monitoring and controlling of RHINO, also programming the FPGA. 1Gbps and 10Gbps interfaces for high-speed connection network using TCP or UDP transport layer protocols.
- **I/O Interface** - FPGA Mezzanine Card (FMC) connector for interfacing with ADCs and DACs.
- **Clock** - 100MHz FPGA clock.

2.2.2 DSP Firmware for RHINO

The demand for the design and implementation of IP cores is increasing steadily due to the need for an increase in productivity and time-to market [11]. Third party IP cores are extremely expensive and prone to errors. Furthermore, the open IP cores such as OpenCores and GRLIB are not parametrizable as one would like [27]. With FPGAs, one can implement flexible radio which supports a wide variety of waveform by taking advantage of its reconfigurability. This can be realised by the implementation of a library of parameterizable IP cores for purposes of reconfigurability.

There are a number of successfully completed research projects in the RRSG at UCT which resulted in the firmware for RHINO [53] [40], however, this research project makes use of the firmware by L. Tsouenyane [53]. The scope of this report will be limited to I/O cores as they are the most relevant to the project at hand. The following section will briefly elaborate on these cores. Detailed information can be found in the [53] thesis report.

The following IP Cores will be discussed in this section:

- Digital Down Converter Core
- ADC/DAC interface Core
- Ethernet UDP core

The DDC is a DSP core with the functionality to convert IF signals to a baseband signal, that is, a bandwidth of frequencies lower in the frequency spectrum. The sampling rate is reduced in this way, filter requirements and further signal processing become more feasible [29]. One

of the advantages of implementing a DDC on the FPGA is the speed associated with this, also the fact that once designed, the sampling frequency does not depend on outside factors such as temperature and time. Figure 2.5 shows DDC architecture.

The Numeric Controlled Oscillator (NCO) builds discrete-time cosine and sine waveforms with configurable waveform frequency, and these waveforms are synthesized from a lookup table with sine and cosine values. The Frequency Tuning Word (FTW) determines the phase step of the accumulator. In order to equally distribute the unwanted harmonics produced in the waveform, a Linear Feedback Shift Register (LSFR) is added in the last significant bits of the phase accumulator. More information on the designs can be found in the thesis report [53].

The ADC/DAC interface core designed is for a 4DSP-FMC150 FPGA Mezzanine Card (FMC) daughter card. The FMC150 is designed with TIs ADS62P49/ADS62P49 dual-channel 14-bit 250Msps ADC and TIs DAC3283 dual channel 16-bit 800Msps DAC [53]. The TIs CDCE72010 PLL is a clock device which provides a clock to drive the ADC and DAC. The FMC150 core provides Low Voltage Differential Signalling (LVDS) interface to a 4DSP FMC150 daughter card. The design is configured to operate the ADC and send digital samples to the DAC at 61.44MSps. FMC150 core was tested to have a maximum sampling rate of 163.84MSps and 245.76MSps for the ADC and DAC [53].

The UDP/IP core enables RHINO to communicate with external peripherals through the Ethernet interface. The design of the core was based on the combination of IPv4 and UDP to provide efficiency and high speed over a Gigabit Ethernet (GbE). Gigabit Ethernet is a transmission standard based on IEEE 802.3 protocol with a transmission rate of up to 1000Mbps [57]. It is faster than IEEE 1394a and IEEE1394b and can transmit up to 100 meters without requiring a hub. The core uses Open-Cores tri-mode Media Access Control (MAC) address published under GNU Lesser General Public License (LGPL), they support data rates of 10, 100,1000Mbps which is ideal for the core since it is configured for the 1000 Mbps speed. UDP is selected as the transport layer protocol because of its high speed, real-time data transmission ability. More information on UDP vs TCP will follow in the coming section of this chapter.

The UDP/IP core was tested successfully and the maximum throughput speed attained on GbE using Wireshark was 98.62MB/s [53].

2.3 DATA TRANSMISSION PROTOCOLS

This section of the chapter reviews the network protocols as GNU Radio, running on a PC, needs to connect via a network to the RHINO in order to acquire digitized samples and control

messages. The transport layer in the OSI seven layer model as seen in Table 2.1 is responsible for the host-to-host communication through LAN or in remote networks.

The data transmission protocols responsible for packet delivery in this layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Reliability and speed are basic requirements in communication for modern applications, especially with real-time systems. The two protocols are used in different communication systems depending on the factors that are essential in the system, for example, TCP/IP is preferable when time is not as essential as reliability. Table 2.2 [3] outlines the differences between these two data transmission protocols.

Table 2.2: Comparison of UDP to TCP

	UDP	**TCP**
Connection	Connectionless	Connection oriented. Requires handshake before data is sent.
Packet Entity	Datagram	Segments
Reliability	Not reliable. Packets are sent without any check as to whether they have arrived or not.	TCP is reliable. Messages managed and acknowledged upon arrival. Error checking done.
The speed of transfer	UDP is fast compared to TCP due to the lack of error check	Slower than UDP.
Ordering of data	Packets are independent, as so no ordering available.	Packets are rearranged as specified.
Header size	8 Bytes. Refer to Figure 2.8	20 Bytes. Refer to Figure 2.9
Applications	UDP is used in applications where speed is essential, Real-time streaming. Application such as gaming, Voice over Internet Protocol (VoIP), and video streaming.	Applications where high reliability is required. Applications such as emails, file transfer.

Based on the facts outlined in Table 2.2, it is a clear indication that UDP is the most desirable for the real-time data streaming applications. One of the major requirements for data transmission is minimum latency. Because UDP is connectionless with no acknowledgment, data is continuously transmitted which minimises latency and increases the transmission rate. Furthermore, UDP is faster than TCP in terms of communication due to lack of error check. Therefore, it follows that it would be ideal for real-time streaming applications such as VoIP and video streaming.

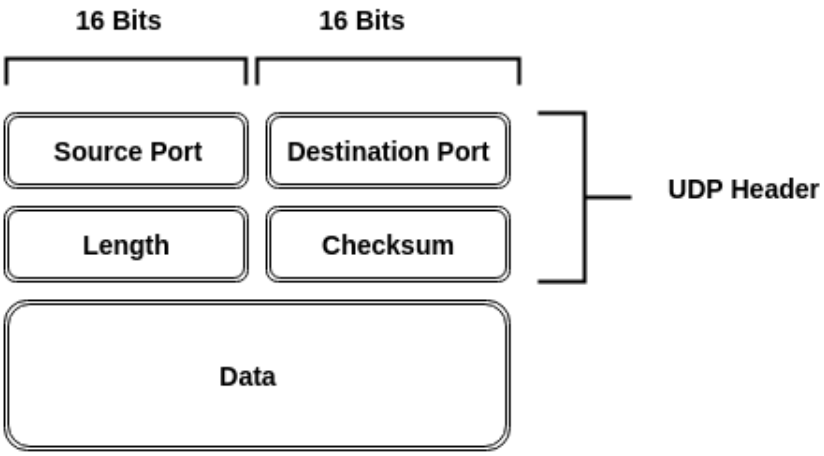


Figure 2.8: UDP packet

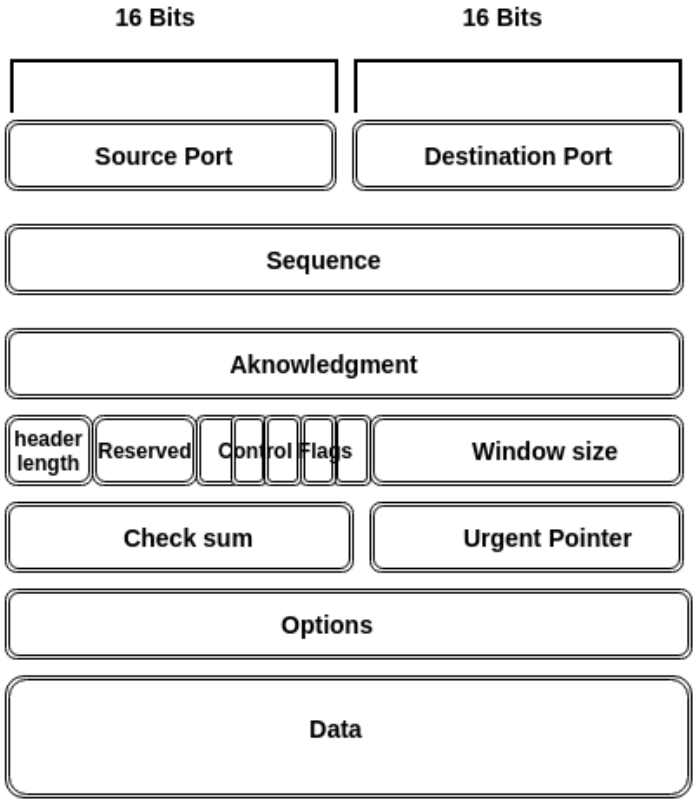


Figure 2.9: TCP packet

2.4 BUFFERING AND DATA ACCESS PATTERNS

Communication between two devices with different speeds can sometimes cause latency in the transmission, the bottleneck is the time taken for the I/O and memory access [54]. The buffering types and data accessing patterns considered in designing RHINO source block will be highlighted in this section.

2.4.1 Circular Buffer

This is a data structure that has a fixed length and uses a single buffer. The circular buffer is used to store the most recent data values of a continuously updated data source by using the first in first out (FIFO) data accessing pattern [47]. Data is accessed sequentially and updated accordingly. This type of buffer is best for queues that have fixed lengths and are updated regularly, for data of the streaming nature. Figure 2.10 demonstrate a circular buffer.

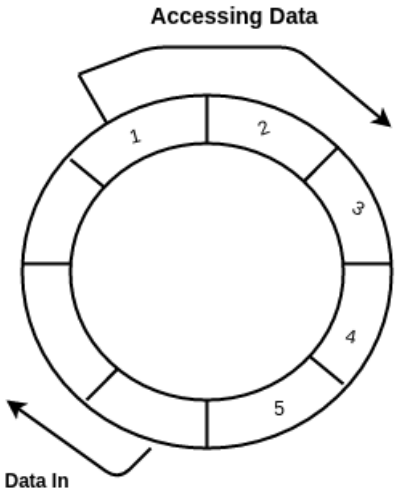


Figure 2.10: Circular buffer data structure [47]

2.4.2 Single Buffer

This is the simplest and basic of the data structures used to store data. With a single buffer, data is ”packed” in the buffer until full and then thereafter emptied, this happens continuously. Figure 2.11 shows the structure of a single buffer. Data come in from one end and accessed on the other end following the FIFO pattern.

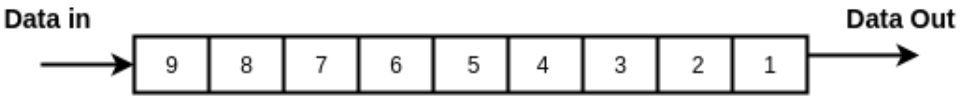


Figure 2.11: Single buffer data structure

2.4.3 **Double Buffer**

A double buffer is a buffering concept that uses two buffers instead of a single buffer as in the previously mentioned data structure. The two buffers are used simultaneously for different processes. Data is accessed from one while another one is used to receive and store data, and vice versa. Figure 2.12 shows the two stages in storing and accessing data using a double buffer.

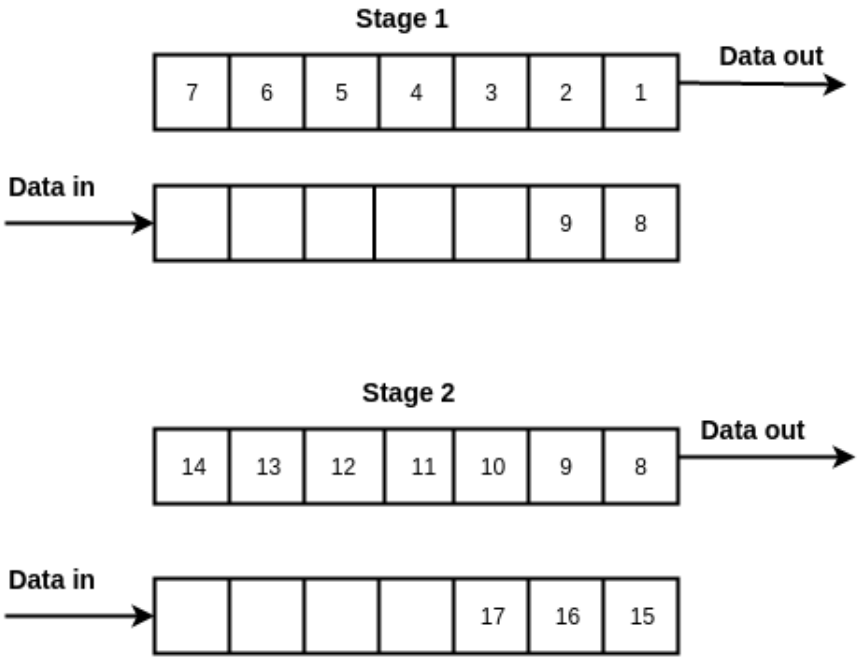


Figure 2.12: Double buffer data structure

2.5 COMPARISON OF SDR BASED SOFTWARE TOOLS

This section will report on the commonly used SDR based software tools for DSP purposes at the University of Cape Town (UCT) namely: GNU Radio, MATLAB, Python SciPy and Delite. Table 2.3 highlights the differences in these tools. Furthermore, it shows the advantages and disadvantages of one over the other. A tick (✓) in Table 2.3 indicates that the software tool satisfies the feature and a cross (x) indicates that the feature is not met.

As depicted in Table 2.3, the following five features will be considered [21] [44]: 1) Open source: indicates if the software is freely accessible for use in SDR projects under an open-source license such as GNU license. 2) Ease of use to SDR experts: this means that the individuals considered in this case are not necessarily programming languages experts but DSP and/or SDR experts who are familiar with MATLAB/Octave. An assumption is made that with these individuals, there is a lack of extensive programming background, therefore, this criterion considers the use of tools that are familiar to the individuals. 3) Graphical User Interface: the

graphical user interface refers to the graphical modelling environment where each of the DSP models can be connected using software links in an intuitive way. 4) Extensibility: in this context refers to the user being able to extend on or optimize the features already available on the software. 5) SDR support: this feature refers to the ability of either of these tools/programming languages to provide configuration interfaces for real-time data acquisition systems which capture data from antennas, digitize it, and stream it to the host PC for processing using available DSP algorithms.

Table 2.3: Comparison of SDR based software tools

Criteria	Python Scipy	GNU Radio	Matlab	Delite
Open Source	✓	✓	x	✓
Ease of use to SDR experts	x	✓	x	x
Graphical User Interface	x	✓	✓	x
Extensibility	✓	✓	✓	✓
SDR Support	✓	✓	✓	✓

Python SciPy is an open source python library used for technical and scientific computing. It offers modules for Fast Fourier Transform (FFT), signal and image processing, Ordinary Differential Equations (ODE) and many other common tasks in engineering [39]. The SciPy library's development is supported by an open community of developers. Even though the software tool provides functionalities of DSP which can be implemented in SDR, implementation requires a user to be comfortable with the python programming language. This would mean learning a new programming language which would not be ideal for an SDR expert. The software does not provide a means of a graphical interface for the user, however, one can implement new features to add for an SDR functionality required.

GNU Radio is an open source toolkit that provides DSP blocks implemented and ready to use for SDR, it supports both the python and C++ programming languages. It can support data processing and streaming simultaneously. GNU Radio Companion (GRC) provides a drag and drop environment which makes the software easy to use for non-programmers. There is a lot of support from the GNU Radio community with extensive documentation. GNU Radio functionality can be extended using the out of tree module feature.

MATLAB is a programming language developed by MathWorks for mathematical use and in other domains such as DSP, machine learning, and SDR. It is well known as the reference language for most DSP tools and/or languages such as Julia [20], Octave [15], Python SciPy [39], and Scilab [39]. MATLAB allows implementation of algorithms which includes DSP functionalities, plotting of functions and data, the creation of user interfaces. Furthermore, it interfaces with languages such as C/C++ C# and java. It also has an additional package,

Simulink, which adds a graphical multi-domain simulation environment. The software is used in industries and academia for purposes in engineering, science, and economics. The main disadvantage with using MATLAB is the expensive license cost.

Delite is a compiler framework developed in Scala where users implement their own Domain Specific Languages (DSL). The framework provides data-parallel patterns which can be executed on heterogeneous computing platforms such as hybrid CPU and GPU architectures [31]. Scala being a relatively new language requires some effort to learn and understanding, thus making Delite suitable for language design and user experts. While open-source efforts such as in [32] and [31] show some promising results on the use of Delite for DSL designs, its level of abstraction is perhaps still suitable for the text-based programming construct only.

While Python SciPy and Octave, and most of other DSP tools resemble MATLAB in both syntax and computation functionality, they do not provide as rich constructs in data flow model of computation as well as graphical representation of SDR processing flow as in GNU Radio. Therefore, it follows that GNU Radio is the ideal option for an SDR-DSP based framework. Section 2.6 will present more details on GNU Radio.

2.6 OVERVIEW OF GNU RADIO

In recent years, the data input and processing speeds of GPPs have become sufficiently fast to be able to implement most, if not all, digital signal processing needed for practical radio(understandably, they are still limitations according to frequencies and bandwidths compared to custom hardware solutions). An ideal SDR system, which takes this evolution to its furthermost point, would be an ultra high-speed ADC attached to an antenna. With all the subsequent processing done in software - this is generally not a practical solution except for lower frequencies and very strong signals at this point; but it is nevertheless useful to contemplate this abstracted model which is referred to as the ”Ideal SDR Concept” [17]. There are plenty of open-source SDR software frameworks, some of the popular ones include OSSIE, powerSDR, SDR Sharp [2] and GNU Radio. GNU Radio has proved to be ideal because of its rich resources with signal processing.

GNU Radio is a free and open-source software development toolkit that provides DSP blocks to implement software radios [16]. It can be used with low-cost external RF hardware to implement SDR, but also without the hardware in a simulation-like environment. GNU Radio is widely used in academics for research purposes mainly in wireless communications, in commercial environments for the real-world radio system. It is also used by hobbyists.

GNU Radio only handles digital data, applications can be written to send or receive data from digital streams which can then be transmitted using hardware. The software performs a variety of signal processing and these processes are encapsulated in what is referred to as ”blocks”. The blocks can be connected with one another and communicate by sending data from one block to the other. GNU Radio provides different types of blocks such as modulator blocks: AM Demod, FM Mod, GMSK Mod/Demod, filters. Networking tools: TCP Source/sink UDP Source/Sink, UHD:URSP sink/source, Async Msg source, filters blocks: Band pass filters, FFT filter, low pass filter etc. The applications are commonly written in Python programming language and the DSP blocks are written in C++ to enable implementation of real-time and high throughput radio systems.

The blocks are generally parameterizable, this means that the parameters on each block can be changed to meet the requirements of the experiment. Each block produces or accepts data based on the data output/input type set by the user. The type needs to be consistent with all the other blocks in a given flow diagram. However, type converter block (Float to Complex, Int to Float, Short to Float) exist in the GNU Radio source tree. These blocks change the data output/input from one type to the other. The data types are colour coded as shown in Figure 2.13.

Figure 2.13: Colour codes for data output/input types on GRC

GNU Radio can be used by non-programmers using the GNU Radio Companion. This is a friendly graphical user interface in GNU Radio which allows the user to implement signal processing applications with just a drag and drop. Because GNU Radio is an already well established SDR platform, there exists plenty ready to use blocks for one to build an SDR application. The exciting thing about GNU Radio is that it is extensible, that is, if there happens to be a certain functionality that does not already exist, one can create and add the block to the tree. The following subsection goes into details as to how this can be achieved.

2.6.1 Out of Tree Modules

OOT module is a GNU Radio component available to users who would like to extend the functionality of the software, the blocks created do not become part of the GNU Radio source tree but however, the blocks can be added to the user's tree to be used and maintained as needed. The module can be added to the GNU Radio source tree by submitting it to the devs for upstream integration [16]. Many OOT projects are hosted at the Comprehensive GNU Radio Archive Network (CGRAN) [41] available through the PyBOMS tool.

The following subsection of the chapter will cover a guide in creating a source OOT module. The guide of creating a source OOT module is based on an example *chirp source* signal block created as part of getting familiar with the GNU Radio OOT module feature.

Source OOT module

A source block is a type of block in GNU Radio that generates data or outsources data from a peripheral, the data can also be read from a file. A sink block, on the other hand, accepts the data either from storage in the case of file sink, or display in the case of plotting sink blocks, for example, time sink block, frequency sink plot. This section will highlight some of the basic commands in creating an OOT module.

gr_modtool command is used from the terminal to create a new module, the command should be run in the directory where the new module is to be located, this should be outside the GNU Radio source tree. Different options for *gr_modtool* can be viewed using the following command: *gr_modtool help*. Appendix E gives details on the instructions executed in creating a module and installing it into GNU Radio.

After creating a module, GNU Radio automatically generates files that can be edited based on the functionality of the particular block. All the programs written in C/C++ will be stored in *lib/* folder, the header files will be in *include/* folder if they are to be exported, otherwise in the *lib/* folder if they are only relevant during compile time. Essentially, all the C/C++ to be edited will be in the *lib/* folder. Python files go in the *python/* directory. Simplified Wrapper and Interface Generator (SWIG) creates the code that makes it possible for the python code to interface with the C++ code. It is unlikely that the *swig/* folder will be needed for a simple source/sink block unless some extraordinary functionalities are required for the block. An eXtensive Markup Language (XML) file is necessary for the block to be available in the GRC. The instructions to executing this feature are also covered in Appendix E.

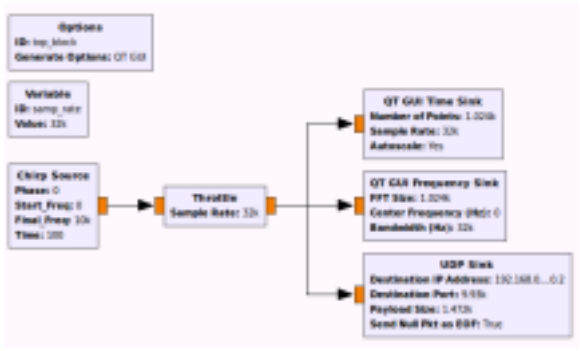
The chirp signal source block was implemented as part of the preliminary experiments to test

the implementation of OOT modules and the transmission of data over UDP. Table 2.4 shows the parameters set for the chirp source block. The parameters can be changed. However, the output data type can only be set to *float* since the chirp source block inherently produces float output data.

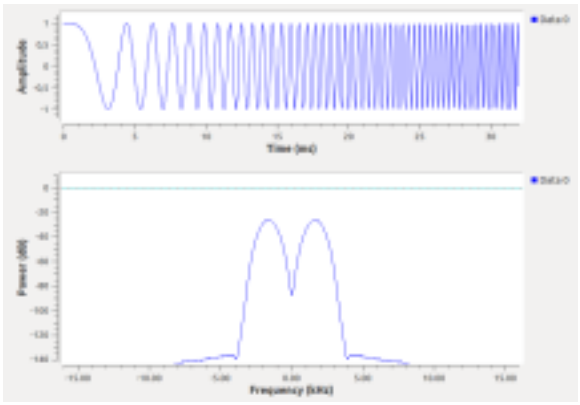
The block diagram for sending a chirp signal over UDP is shown in Figure 2.14a, and the output plot is shown in Figure 2.14b. The throttle block as seen in the figure is added in cases where a peripheral hardware is not connected, this is to prevent congestion of the CPU on the PC. The QT GUI Time sink block displays the signal in time domain while the frequency sink block in the frequency domain. The UDP sink block is responsible for linking PC1 to PC2.

Table 2.4: Chirp Source Block Parameters

Parameter	Value
Phase	0
Start Frequency	0 (Hz)
End Frequency	10000 (Hz)
Time	100 (s)



(a) Chirp Source Block Diagram



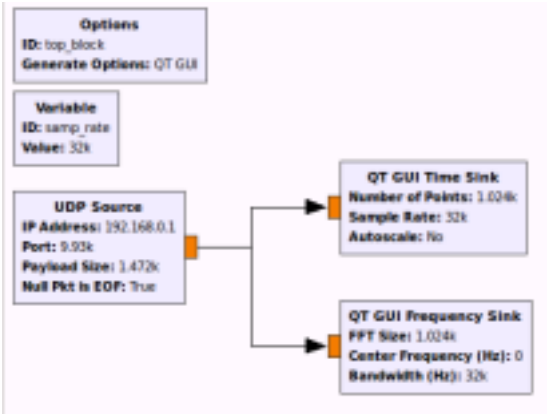
(b) Chirp Source Block output Plot

Figure 2.14: Chirp Signal Source test on the sending side.

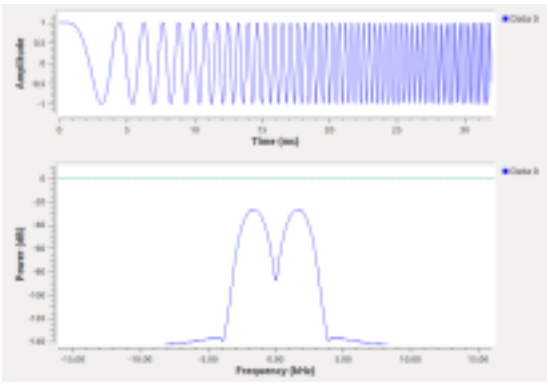
The plot shown in Figure 2.14b displays the chirp signal in time and frequency domain on the sending side. Figure 2.15a shows the block diagram on the receiving side and the plot results are shown in Figure 2.15b.

The UDP source block in Figure 2.15a connects with the UDP sink block on the sending side, Figure 2.14a (PC1) accepts data and sends it to the time and frequency sink blocks.

As mentioned in Section 2.6, GNU Radio can be used with an external RF hardware to implement SDR. Subsection 2.6.2 demonstrates the capabilities of GNU Radio with an SDR hardware. URSP modules [10] and RTL-SDR [37] dongles are some of the RF hardware that was



(a) Block Diagram on the receiving side.



(b) Chirp Source Block output Plot

Figure 2.15: Chirp Signal Source on the receiving side.

used with GNU Radio to implement an FM receiver. RTL-SDR dongle will be used in this project as a benchmark for testing signal processing platform based on RHINO. Subsection 2.6.2 outlines the implementation of an FM receiver using the RTL-SDR dongle and GNU radio.

2.6.2 GNU Radio with RTL-SDR

Using an RTL-SDR is the cheapest way of receiving and processing radio transmission using a PC with frequencies ranging from 20-2000MHz. This range of frequencies includes the following radio signals: FM radio stations, police radio transmissions, small radio devices such as car key [38]. RTL-SDR dongles are readily available for purchase. The section of the chapter that follows will demonstrate a basic experiment for building an FM receiver using GNU Radio and the RTL-SDR dongle.

FM Receiver using RTL-SDR

The tutorial is a demonstration of an FM receiver using RTL-SDR hardware. This design is based on a tutorial provided by [37]. Figure 2.16 shows the flow diagram of the experiment using RTL283U USB dongle.

The block diagram for the FM receiver as shown in Figure 2.16 is built from the following components:

- **RTL-SDR Source** - The RTL-SDR source block produces baseband samples with the sampling rate set to 2M, the center frequency can be set in this block, station frequency.

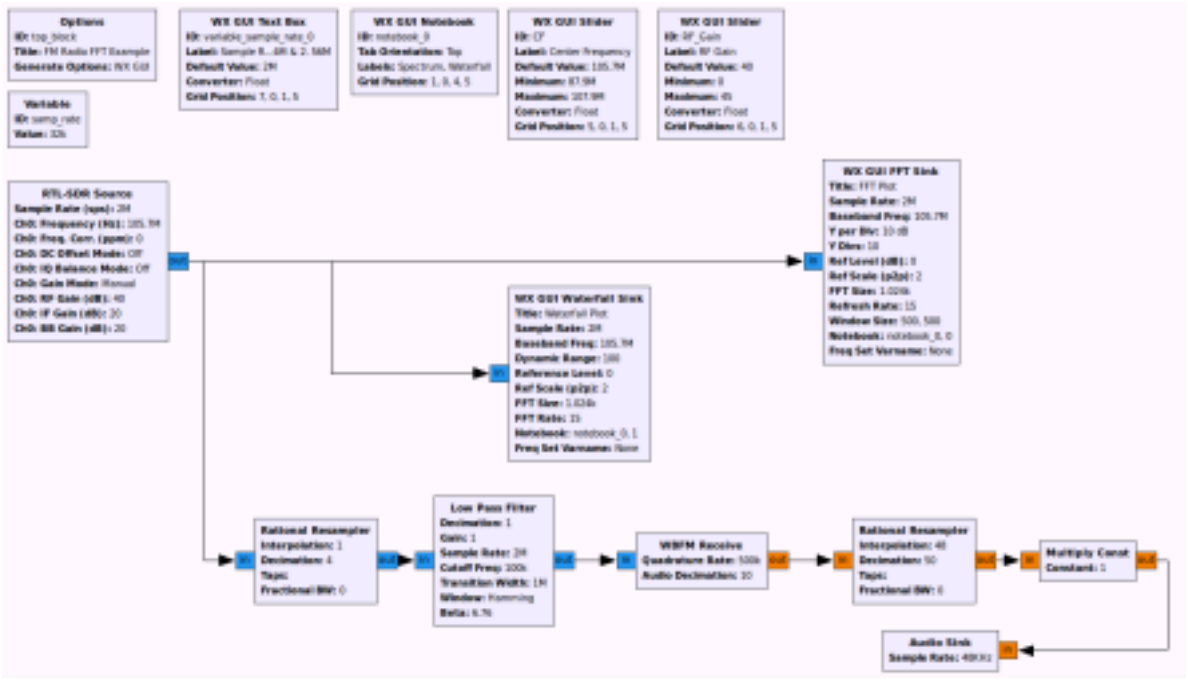


Figure 2.16: Flow Diagram for a GNU Radio FM Receiver using RTL-SDR Dongle

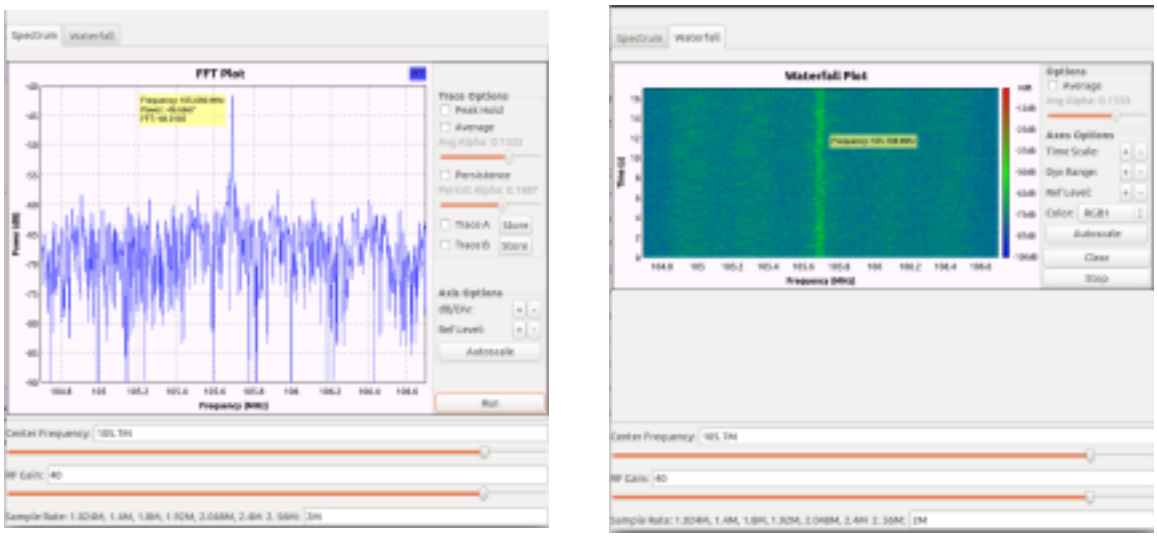
Osmocom source block can be used in place of the RTL-SDR source block.

- **Rational Resampler** - The rational resampler adapts the notional rate of the stream by decimating/interpolating.
- **Low pass filter** - The low pass filter gets rid of the frequencies other than the one set in the RTL-SDR source block.
- **WBFM Receive** -The block accepts the baseband samples and outputs demodulated audio.
- **WX GUI FFT Sink** -Displays an FFT of the source signal from the RLT-SDR source block.**WX GUI Waterfall sink** has the same function but with a different view.
- **Audio sink** - Outputs the audible demodulated audio with PC sound card.

The output results of the above experiment are shown in Figure 2.17. The FM receiver was tuned to SAfm radio station 105.7MHz.

2.7 SUMMARY

This chapter highlighted the concept of software defined radio and how it has become an important topic in wireless communications systems in Section 2.1. Every technology regardless



(a) FFT plot

(b) Waterfall Plot

Figure 2.17: FM Receiver Plot Results for the FM Receiver using GNU Radio and RTL-SDR dongle

of how revolutionary it may be has some challenges, so does software defined radio. The advantages and disadvantages of SDR were also reviewed in this chapter. It is abundantly clear that the pros outweigh the cons. Signal processing architectures play an important role in SDR. Signal processing devices such as GPPs, DSPs and FPGAs were reviewed in details in Subsection 2.1.2, highlighting the advantages of one over the other and how they can be implemented in SDR. Digital signal processing on baseband signals is then covered with emphasis on the software-based processing.

RHINO is the hardware platform that will be used in this project, Section 2.2 provided an overview on the platform, with adequate information on the hardware and software of its make up. With the help of one the projects carried out, RHINO is a firmware ready signal acquisition and data transmission platform to be used with GNU Radio. GNU Radio is a free software toolkit used in creating radios, its functionalities can be extended using the out of tree module feature, details on the software was covered in Section 2.6. To finalise the chapter, a basic example using RTL-SDR dongle and GNU Radio to create an FM receiver was reported on in Subsection 2.6.2.

CHAPTER 3

METHODOLOGY

This chapter presents the methodology for the design of the RHINO data streamer for GNU Radio. The chapter starts off with an overview of the methodology in Section 3.1 which gives a general outlook on the methodology chapter. The overview section will also highlight the different phases of the methodology and each phase will be elaborated on in the later stages of this chapter. Furthermore, the chapter will show how the project builds upon the literature that was presented in Chapter 2.

3.1 OVERVIEW

This chapter provides guidelines on the design process. It will be composed of the following phases: 1) User requirements and constraints, 2) System design specifications, 3) Hardware specifications, 4) Software design process and 5) System evaluation process.

The chapter will start off by providing the user requirements and constraints on the RHINO streaming interface. The design specifications will then follow and will cover the process followed to design the RHINO streamer. The chapter will go on to provide details of the hardware design specifications and software design process. The last section of this chapter, system evaluation process will present the evaluation process of analysing and validating data obtained in the experiments to test the system.

A summary of the methodology followed is shown in Figure 1.2. It shows all four phases mentioned above and indicates each user requirement going through all the phases in a spiral methodology format. More details of these phases will follow in the later stages of this chapter. The spiral methodology is ideal for this project because it gives an opportunity for each user requirement to go through all the design stages, and any changes required can be made in the later stages. The development of the RHINO streamer is systematic and continuous, developing

new features as the system is built.

RHINO data acquisition for a host PC is a 1Gbps real time SDR-based system which will be realised with the help of a firmware ready 1Gbps FPGA data acquisition platform. With the help of the RHINO channelization capabilities through a DDC core, the system is baseband signal based. The RHINO data acquisition framework is expected to work well with SDR software. As illustrated in Figure 1.2, systematic tests were done for each of the user requirements. The following sections will iterate more on the four phases of the methodology followed in this project.

3.2 USER REQUIREMENTS AND CONSTRAINTS

The following section highlights not only the requirements and constrains for the project, but also the process followed to gather them.

3.2.1 Establishing User Requirements and Constraints

The initial user requirements for the project were stipulated by the supervisor. These were subsequently refined through a process of literature review and study of the platform and desired applications to run on the platform. The RHINO platform was thoroughly investigated through a process of investigating several projects conducted at UCT in the RRSG group over the years [40] [6] [53] [7]. Reviewing the literature related to the platform also helped in establishing the shortcomings of the board with respect to its accessibility and ease of use to new users; these inputs fed into refinements to the system constraints and how to test the system. Section 2.2 highlights the literature on the RHINO platform. Different SDR software tools were investigated in Section 2.5 to determine the best tool for the system. Further discussions with students and research staff in the RRSG group (i.e., the potential users) also provided input to determining requirements for the system. In order to obtain more formalized requirements, a survey was conducted for which both students and research staff interested in using RHINO, or having had experience with RHINO, were asked to complete a questionnaire. The questionnaire is shown in Appendix A. The survey was structured around categories for requirements, such as speed requirements, sampling rates, latency, etc. together with questions where additional response, which did not clearly fit into any of the predetermined categories, could be placed. The decided upon requirements were synthesized from the questionnaires through a process of comparing the responses and identifying priorities, based on how particular requirements were emphasised, such as through the use of words such as ”important”, ”significant”, ”critical”. Similarly issues that the respondents indicated as being of less importance, such as ”not generally important” or ”it is irrelevant ” were provided lower priority in formulating the

requirements. The refined requirements were then reviewed in consultation with the supervisor and two senior research staff members of the group, in order to further improve the requirements and identify the priorities for the supporting framework to be constructed.

3.2.2 **Synthesising System Requirements from Data Collected**

The system requirements were based on questionnaires sent to students and research staff in the research group, from the supervisor, and from a focused group discussion that reflected on the questionnaire responses that was held with the supervisor and two senior research staff members of the research group. The requirements were thus synthesised from a variety of sources. The synthesis method comprised several stages which were performed in response to additional inputs being gathered. The inputs leading to influencing the requirements, and the sequence followed, interleaved by steps of synthesis, were as follows: 1) an initial list and types of requirements to obtain was provided by the supervisor, 2) this list of requirements provided from the supervisor were stated broadly; in order to narrow in on specific needs of the group, the initial requirements were restructured as questions in the survey, as shown in Appendix A, and used to elicit further detail to make the requirements more specific and also to identify priorities. 3) Then an analysis of past projects was done to identify potential shortcomings and these were noted in the researcher's log book. 4) The responses from the survey were then analysed by looking for terms that emphasised or de-emphasised particular requirements or operational characteristics. These were then formulated into a finalized list of requirements for the supporting framework from which the design and development of the framework proceeded.

In respect to the questionnaire responses, the questions focused on the requirements from a perspective of a RHINO user. Features such as sampling rate, latency, the type of processing that is to be done on the PC, the type of data to be processed and stored were incorporated in the questionnaire. From the responses, the users insisted on the RHINO sending raw data through the GbE for processing on the PC. One other of the requirements for the system based on the responses is the need for data to be streamed continuously at maximum packet size. Question 9 highlights the need for storing raw data for later processing, furthermore, there is need to replay data for real-time spectral analysis. Visualisation is also necessary in viewing the RHINO and processed data, for this reason, it is important for the SDR software integrated with RHINO to provide means of visualising the streamed data, all this in real-time. The main constraint for the system is the FPGA sampling rate which will affect the sampling rate for the projects planned for RHINO. The questionnaire was considered to formulate the user requirements and constraints for the system in the following sections.

3.2.3 User Requirements

There are two parts of the user requirements namely: functional and non-functional requirements. Table 3.1 gives a brief outline of the functional requirements of the system while Table 3.2 gives a summary of the non-functional requirements.

Table 3.1: Functional Requirements

	Requirements
1	1Gbps Ethernet streamer system
2	Receive data as baseband signals
3	Real time streaming data acquisition system
4	Compatibility with an SDR software tool
5	Storing streamed data for later processing

The main functionality of this project is the interfacing of RHINO board with a host PC to facilitate the development of SDR applications. The functionalities of the streamer as summarized in Table 3.1 which states that the RHINO streamer will use a 1Gbps Ethernet to stream data to the PC host in real time, thus, it is important to ensure the network card on the PC supports a 1Gbps network speed.

The PC will receive data in the form of baseband signals, I/Q samples from the RHINO FPGA in real time. The data captured can then be used to extract information at the signal processing stages. The system has to be compatible with the SDR software tool in terms of relaying data and processing. One other requirement is for the data to be stored for later processing.

Table 3.2: Non-functional Requirements

	Requirements
1	Minimum loss, minimum latency, maximum throughput
2	Configurable
3	Reliable
4	User friendly

In addition to the functional requirements mentioned, the system also has the non-functional requirements as summarized in Table 3.2. The system is expected to be an effective data acquisition mechanism with minimum loss, minimum latency, and maximum possible data throughput. Minimum loss in this case means the system has to be designed in such a way that very little data is lost during transmission. Minimum latency refers to the time taken to transmit data from the source to the host PC. Data throughput is the calculated throughput for capturing data, that is, the amount of bytes captured per unit time. The system has to be configurable on the

host PC, the user should be able to change the parameters on the SDR software tool based on their requirements.

The system is expected to be reliable, its functionality must be consistent with the already existing functionalities of the software tool used. Furthermore, the system is expected to be user-friendly and accommodating to non-programmers.

3.2.4 System Constraints

Through initial experiments, the following constraints summarized in Table 3.3 were discovered.

Table 3.3: Constraints of the RHINO Streamer System

	Constraints
1	Network constraints
2	Efficiency of code implemented
3	FPGA sampling rates

One of the challenges that can be expected when using a PC host based application is the problems associated with the network used for the data transmission. If the system is run over a network, for example, UCT network, the degree of congestion in the network will affect the efficiency of the system. The C/C++ code implemented affects the functionality of the system, the more efficient the code is in terms of minimum latency and data loss in transmitting data, the more efficient the system will be. The FPGA sampling rate affects the overall data rates for the transmission of data in the system.

3.3 System Design Specifications

This section presents the design specification of the system mentioned in Section 3.1. An explanation for each design specification relating to each user requirement will be elaborated.

Figure 3.1 demonstrates the architecture of the system. Signals are received from the RF front end antenna, the signal is then digitized with the FMC 150 ADC and the ADC core. The DDC channels the signal to a lower digital frequency signal, baseband signal. The UDP Ethernet core interfaces with the 1GbE port on the FPGA to transmit the baseband signal to the connecting PC. On the host PC where GNU radio is installed, the RHINO GNU Radio UDP block accepts the I/Q baseband samples for signal processing.

The projects scope does not include the RFFE and the firmware implemented on the RHINO FPGA as all of these have already been implemented [53]. Therefore, it follows that the focus on

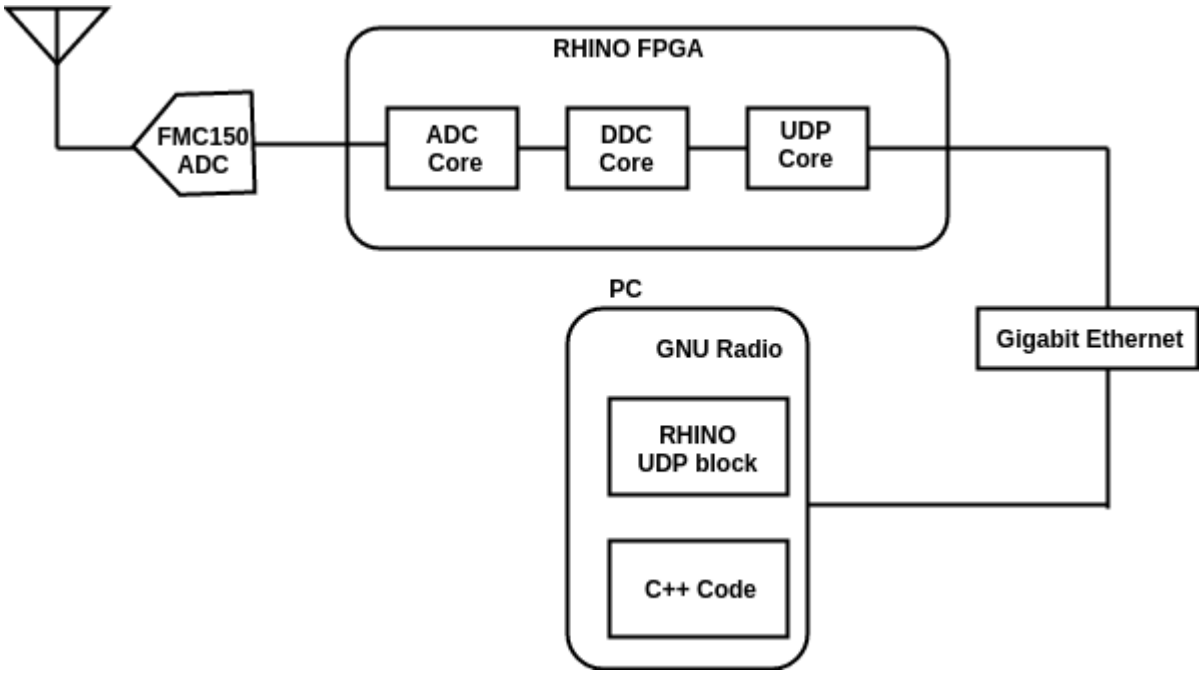


Figure 3.1: Architecture of the system

the project is mainly in the communication between the RHINO board and GNU Radio installed on the PC. Based on Section 2.5, GNU Radio is the ideal SDR-based software tool for this project. Figure 3.2 illustrates a refined architecture of the system which shows a point-to-point connection between the RHINO FPGA and the host PC. RHINO communicates with the host PC over Ethernet with the help of the firmware implemented on the FPGA, I/O cores which will be presented in details in the chapters that follow. The RHINO UDP block is implemented on GNU Radio following the procedure highlighted in Subsection 2.6.1 which then communicates with other signal processing blocks in the GNU Radio source tree.

3.4 HARDWARE DESIGN SPECIFICATIONS

This section of the report will present the hardware design specifications. It should be noted that the hardware design does not entail the development of a new hardware altogether but an elaboration on the specifications of the hardware that was used in this project. The experimental set-up will also be highlighted.

3.4.1 Hardware Specifications

The hardware design will elaborate on the specifications of the RHINO board and the PC used in this project. As mentioned in Section 1.4, the main aim of the project is to integrate the RHINO board with GNU Radio software which will run on the PC.

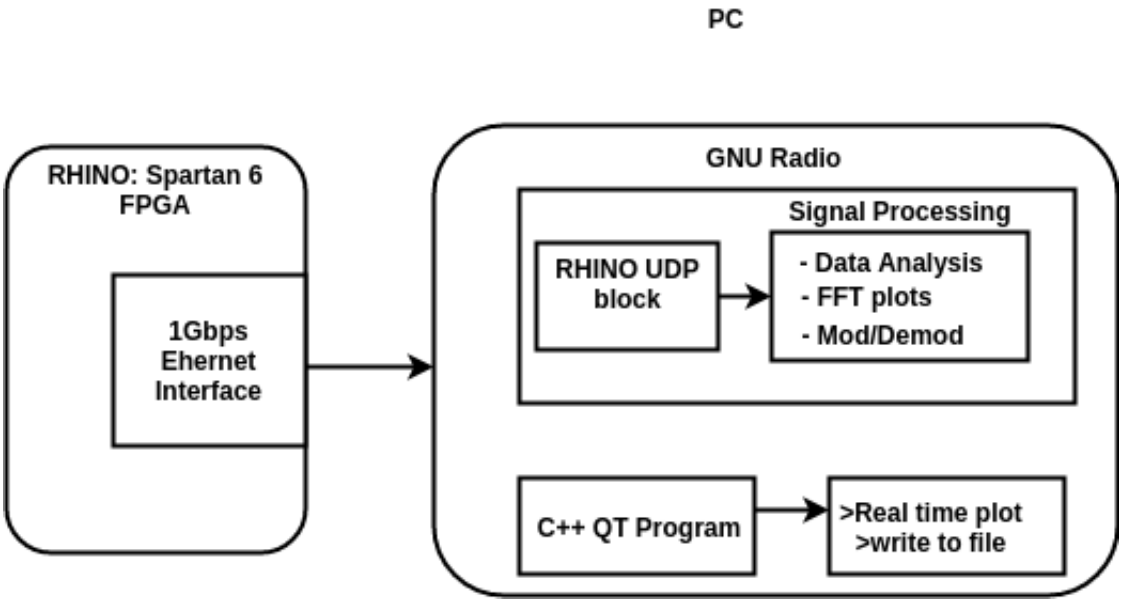


Figure 3.2: Refined Architecture of the system

- RHINO Specifications

  Section 2.2 presented the RHINO architecture in detail. It also highlighted the FPGA type on the RHINO board, the operating system, and the processor. This information highlights the RHINO specifications for this project.

- PC Specifications

  GNU Radio was installed on the host PC, below are the PC specifications:

  – Operating system: 64-bit Ubuntu 14.10
  – Processor: Intel Core Duo CPU E6550 @ 2.33GHz x 2
  – Memory: 2 GiB
  – Disks space:20GB
  – Network adapter: 1Gbps speed

3.4.2 **Experimental Set-up**

The experimental setup consists of a point-to-point connection between the RHINO board and the desktop computer through a 1Gbps Ethernet interface. GNU Radio is installed on the PC. The I/O core and the 1 Gigabit Ethernet core responsible for communication between the RHINO FPGA and peripherals are implemented on the RHINO FPGA.

Both the PC and the FPGA have to be configured with correct network parameters for the UDP communication. The PCs network card has to support a 1Gbps speed because RHINO board uses an Ethernet speed of 1Gbps. The connection is created with Cat5/e UTP cable and data is sent from the RHINO FPGA to the PC. The connectivity between the RHINO and PC was tested by running Wireshark on the host PC and monitoring the UDP packets received.

3.5 SOFTWARE DESIGN PROCESS

The software design process was based on the system requirements presented in Section 3.2. The development of the RHINO GNU Radio block was developed solely in C/C++. However, GNU Radio supports both python and C/C++. The choice was based on the fact that C/C++ offers a rich function library that can be implemented in this project. The protocol used for the data transmission is UDP and the choice was based on the following:

- Because UDP is connectionless, no acknowledgement and handshake, this makes it faster for data transmission as opposed to TCP [24]. UDP is the best protocol for efficient and fast transmission of data. As mentioned earlier, one of the requirements for the RHINO streamer is speed and efficiency.
- UDP is lightweight as compared to TCP, for this reason, it makes a quick connection and less work is done by the network card/operating system to translate the data from the packets, also there is no need to maintain the connection information since there are none [55].
- UDP is relatively easy to use in a network as opposed to TCP. Moreover, UDP offers multicast connections, which is not the case with TCP [56].

3.5.1 Rhino Application Deployment and Activation Process

Hardware Description Languages (HDL) are specialized computer languages used in electronics circuits to describe the behaviour and structure of the digital logic circuits such as FPGAs and Complex Programmable Logic Devices (CPLDs). The commonly used HDL are Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) and Verilog. VHDL is used in this project to develop the the IP cores that were used for the realisation of this project. The cores were developed on Xilinx ISE software, the bitfile is generated in the software then converted into a bof file using the following instructions: *./mkbof -o [name_of_boffile.bof] -s [bittfile] -t 5 [Directory_of_the_bitfile]*, where mkbof is an executable file for creating a bof file. The bof file is then copied to the FPGA using a File Transfer Protocol (FTP) software such as *filezilla* or Trivial File Transfer Protocol (TFTP). To connect to the

RHINO board, the users secure shell (ssh) to connect to BORPH running on the ARM processor and log in as shown in Figure 3.3.

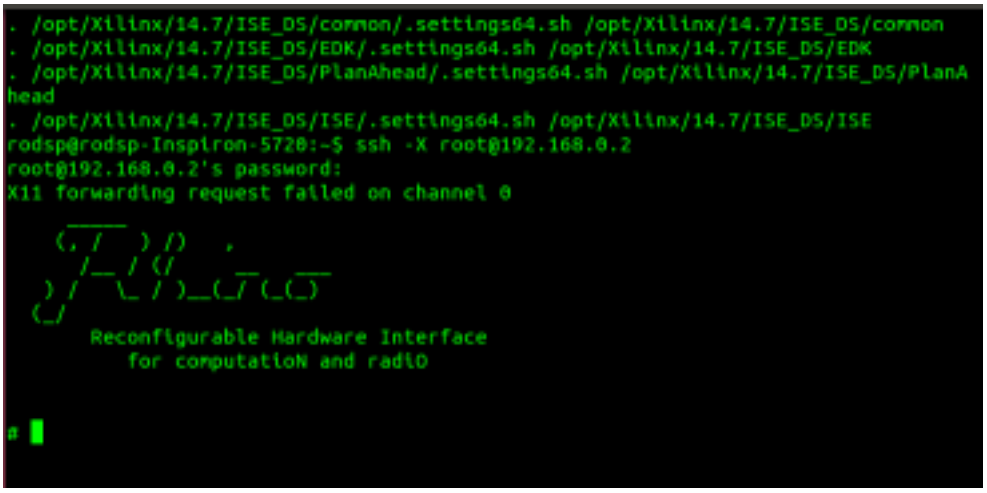


Figure 3.3: Logging into the RHINO interface

Just as software processes executing in BORPH, a hardware process executes with its own memory space and execution domain. BORPH deals with software and hardware processes the same, the processes communicate through standard Linux services and library functions [46]. The two processes communicate through defined hardware registers mapped by software drivers. The process is done by IOREG interface. The IOREG interface is found in the */proc* Linux directory. The directory has all the specifics about the hardware process. The virtual file in the IOREC folder,each with unique ID, represents hardware function specifics in the design. The unique ID of the virtual file, size and access information is stored in the bof file header. Reading or writing on the virtual files interprets to the BORPH reading or writing to physically on the hardware process respectfully.

3.5.2 C/C++ Software Design Process

As mentioned in earlier in the report, the purpose of this project is to provide a supporting framework for the integration of RHINO and a host PC to facilitate development of SDR application. This is realised by designing a C/C++ program that can be further enhanced to enable communication between the RHINO reconfigurable computing platform with other applications. Figure 3.4 demonstrates the design of a *rhinoudpserver* class which provides an extensible baseline for communicating with RHINO. This was then built upon, as a case study, into the an application that allows for plotting of captured waveforms using the class *Mainwindow*.

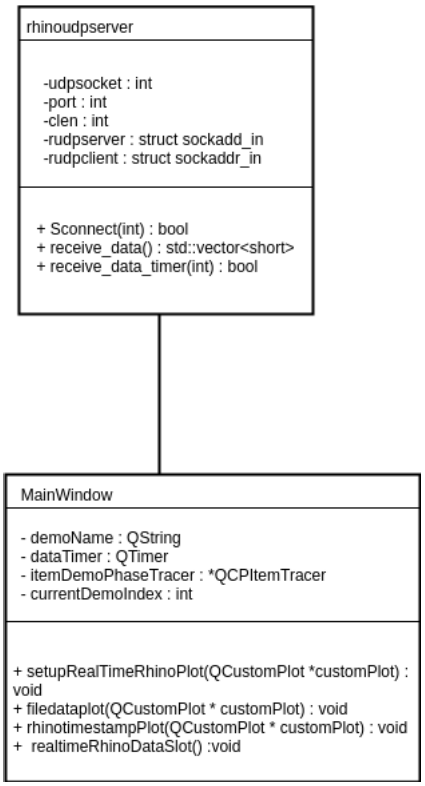


Figure 3.4: RhinoUDPserver and Mainwindow class design

3.5.3 GNU Radio UDP Block Software Design Process

Figure 3.5 below details the development process stages of the GNU Radio UDP block for RHINO.

RHINO GNU Radio block development stages as shown in Figure 3.5 demonstrates a recursive method of development, starting off with a simple implementation of a C/C++ server/client UDP connectivity between two PCs. Testing the communication between the two PCs with the server/client communication code, then testing the communication between the RHINO and the PC. The user defined code was used as the basis for the development of the RHINO UDP block and abstracted onto GRC using XML. The final stages of the project involve the testing of the whole system with the case studies designed.

3.6 System Evaluation Process

This section of the report presents the system evaluation process followed in this project. The system was evaluated with preliminary experiments conducted as a way to analyse and verify the functionality of the RHINO streamer. The experiments were conducted to evaluate the RHINO board connectivity with the host PC and the inter-communication ability of the

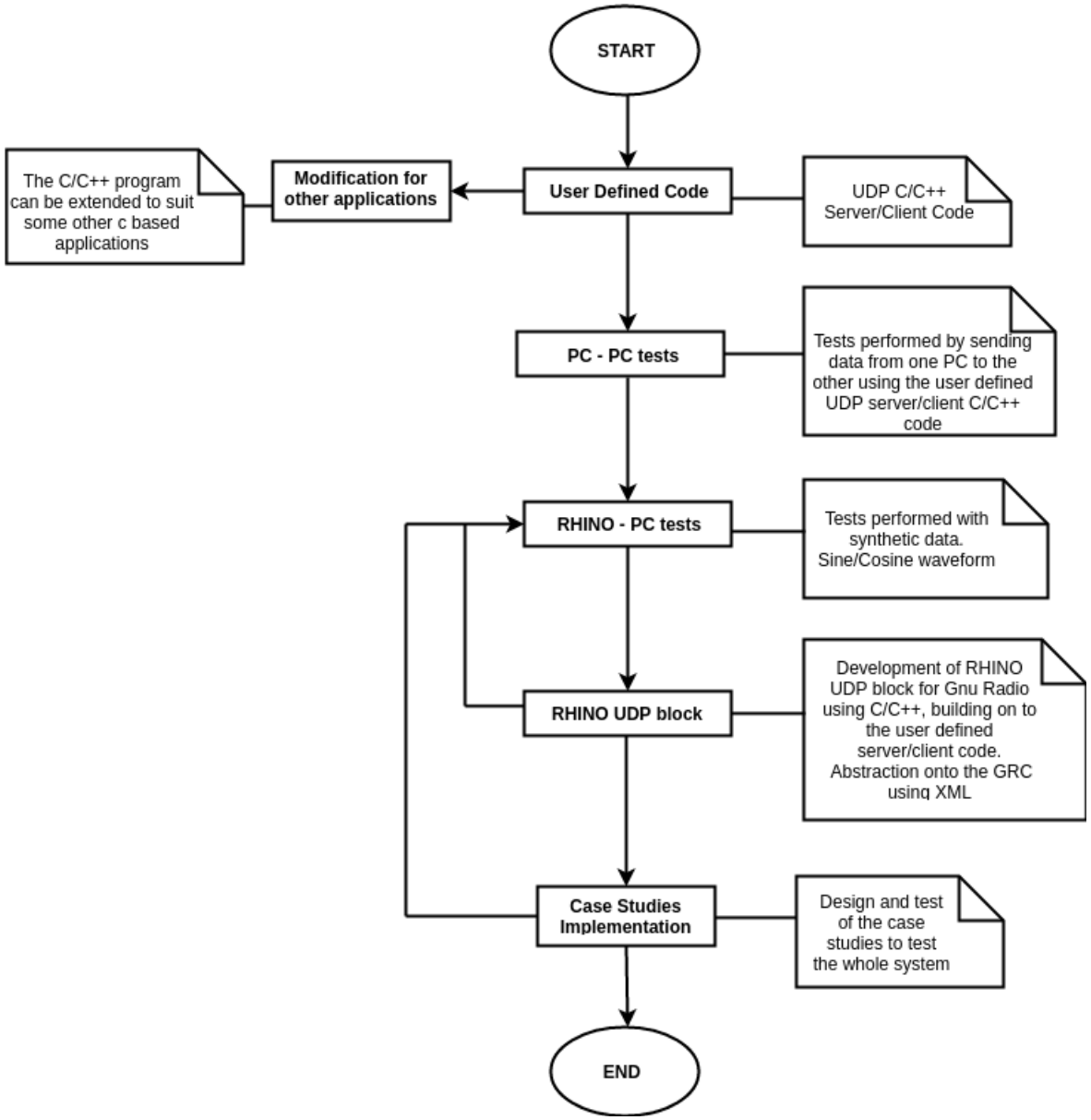
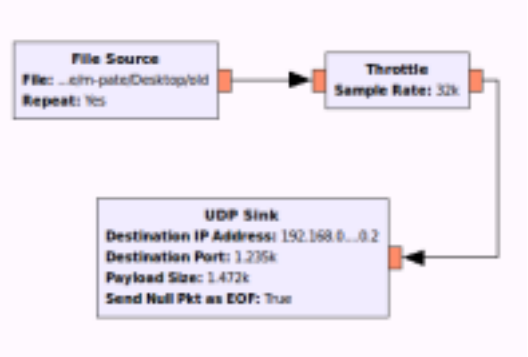


Figure 3.5: GNU Radio UDP block for RHINO development stages

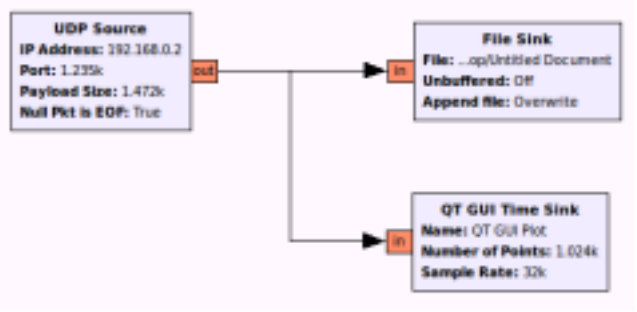
RHINO block on GNU Radio software. Furthermore, the system was also tested for latency and throughput, and finally the reliability of the system. Some of the experiments involved generating synthetic data from the RHINO NCO, streaming it over Ethernet and capturing the data with both the C/C++ program and onto GNU Radio separately. The evaluation process that tests the system as a whole was conducted with the help of a case study simulating real world applications. The following section will present the brief methodology procedure followed as an evaluation process of the system, more details in Section 5.1

3.6.1 System Connectivity and Capturing

Experiments designed as a way to test the data capturing and system connectivity include one that tests the software on its own, independent of the RHINO streamer system. This experiment was meant to evaluate GNU Radio in terms of transferring data from one PC to the next, and capturing data onto the receiving side. Figure 3.6a shows the sending block diagram and Figure 3.6b the receiving block diagram implemented on GNU Radio on two different PCs. The data used in this experiment was streamed from RHINO, captured with Wireshark and written to a binary file.



(a) UDP block diagram on the sending side



(b) UDP block on diagram on the client side

The block diagram shows the binary file read into the UDP sink block which will then communicate with the UDP source block of the receiving side. The throttle block is added to prevent CPU congestion in cases where a peripheral hardware is not used as part of the experiment setup.

On the receiving side, Figure 3.6b, UDP source block receives the data from the sending side and plots the data in time domain, the data is also written to a file. The experiment was performed for both the peer-peer connection and over the UCT network.

UDP Source block was also evaluated with real-time streaming of data, capturing data as it comes in from RHINO.

Synthetic data (I/Q samples) was streamed from FPGA NCO and first captured with an independent C++ program running on a PC. The reason for this experiment is to evaluate the code that can later be extended to not only implement the RHINO UDP block but also interface RHINO with other c based applications. The test is also run with the block to evaluate its data capturing capabilities when incorporated into the GNU Radio software.

3.6.2 Latency and Throughput

The purpose of this section of the system evaluation is to investigate the efficiency of the user-defined code that will be developed into the GNU Radio UDP block for RHINO. The performance test was structured in the following manner: *gettimeofday()* was used to note the time taken to transmit different number of samples from the RHINO FPGA. The number of samples is incremented in 2^N intervals, where N is an integer (0,1,2...,N) and time noted in seconds for different values. This is the latency between requesting the data samples and receiving the samples. Throughput is then calculated based on the number of samples and the size of each packet.

3.6.3 Reliability

The system is also evaluated based on its reliability. This is in-line with the data capturing ability of the system. The RHINO streamer is evaluated for capturing data reliably as this is one of the system requirements mentioned in Subsection 3.2.3. The methodology followed, in this case, is to compare the inbuilt UDP source block and the RHINO UDP block. Evaluating the RHINO UDP block in effect evaluates the code that was implemented in building the block. Synthetic data (I/Q samples) generated in the RHINO FPGA is streamed over Ethernet and captured first with the inbuilt UDP source block and then with the RHINO source block. In both instances, the data is plotted in time domain and results compared to each other.

Reliability, in this case, is measured in terms of packets loss during transmission which will be evident in the inconsistency of the sinusoid graph in the time domain plot. The expected result is a consistent sinusoid showing on the plot in real time. This would then imply that the RHINO streaming interface for GNU Radio system (RSIGR) is reliable and can be implemented in a real world application.

3.7 SUMMARY

This chapter covered the detailed methodology followed in this project by highlighting the design process in four phases, namely: user requirements and constraints, design specifications, hardware and software design process and finally the evaluation process. The user requirements

were highlighted in two parts, the functional and the non-functional requirements. The design specifications of the project are denoted in a form a detailed and refined architecture of the system, the detailed architecture showing more components of the system, unlike the refined architecture which shows a direct communication between the RHINO FPGA and the PC.

The chapter also highlighted the hardware and software design process, detailing the RHINO UDP block on GNU Radio development process. Furthermore, the chapter presented reasons for choosing UDP as the protocol used for data transmission. The hardware design process highlighted the RHINO and PC specifications. The systems evaluation process section evaluated the system based on system connectivity, data capturing ability, latency, throughput and reliability.

CHAPTER 4

SYSTEM DESIGN

This chapter outlines the system design of the RHINO streaming interface for GNU Radio. The chapter starts off with the actual architecture of the streamer, then goes into details of the implementation of the RHINO UDP OOT module that provides data transmission from RHINO to the PC, that is, the source module. Data accessing patterns and buffering techniques used in the RHINO source block will be highlighted. The chapter will thereafter outline the designs of the preliminary experiments conducted, and finally, a case study to test the system as a whole.

4.1 ARCHITECTURE OF THE RHINO STREAMER

The RHINO streamer consists of a firmware ready RHINO as a data acquisition and transmission platform, streaming data to the GNU Radio installed on a workstation PC over a Gigabit Ethernet connection. ”Firmware ready” in this case means that the RHINO platform used in this project has running on its FPGA IP cores which facilitate the communication between the RHINO board and peripherals. The relevant cores, in this case, are the I/O cores which will be presented in more detail in Section 4.4. Figure 4.1 presents the architecture of the system, detailing the connectivity between each component.

The RHINO firmware architecture was designed and implemented as a Masters project by one of the students at UCT [53]. This includes the design of DSP cores (DDC core, FIR core etc.) and I/O cores (ADC/DAC core and GbE core). The ADC/DAC interfaces with the FMC150 card and the GbE core interfaces with the RHINO Gigabit Ethernet port. The DSP cores perform digital signal processing functionalities such as fast Fourier transform (FFT), finite impulse response (FIR), digital down converter (DDC) etc. Details of the design and implementation of these cores can be found in [53]. The scope of this project is limited to the design of the software side of the SDR implementation, the implementation of data acquisition and signal processing on the GNU Radio software platform. For this reason, only the configuration of the

firmware will be covered in this chapter.

On the PC side, data acquisition mechanism is through the GNU Radio RHINO source block for receiving data from RHINO. The block is designed in such a way that it intercommunicate with the rest of the blocks in the GNU Radio source tree. The implementation of the block will be elaborated on further in the sections that follow.

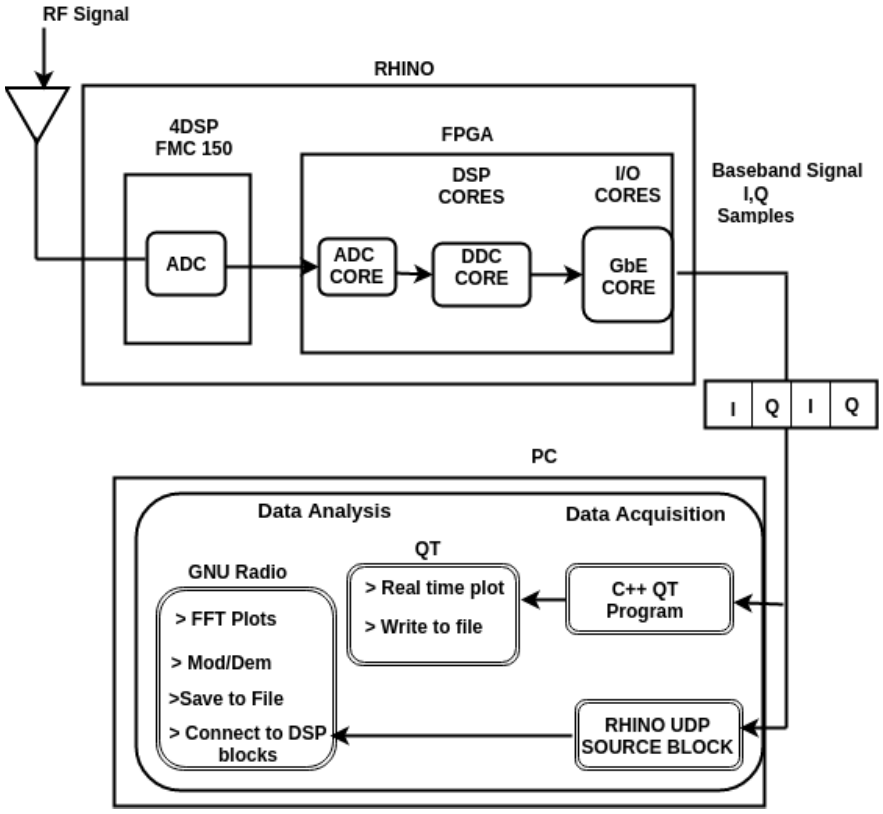


Figure 4.1: RHINO Streamer Architecture

4.2 IMPLEMENTATION OF THE C/C++ PROGRAM

The implementation of the C/C++ program provides a basis for the communication of RHINO and the host PC. The code can be extended to provide other functionalities for other applications. In this instance, the code was designed to plot the data sent from RHINO in real time using QT. Some of the functionalities include plotting data from a file and calculating the time taken to capture certain amounts of samples from the RHINO board. The main class is the *rhinoudpsever* which has functions as seen in Figure 3.4. The class functionality has been extended by implementing *Mainwindow* class which uses the *rhinoudpsever* class to connect to RHINO and plot data. Figure 4.2 shows the interaction between the two classes. The key for the three function choices are as follows:

- Option 1 - Plot data in real time
- Option 2 - Plot samples from previously recorded data.
- Option 3 - Calculate different times taken to capture different sizes of data.

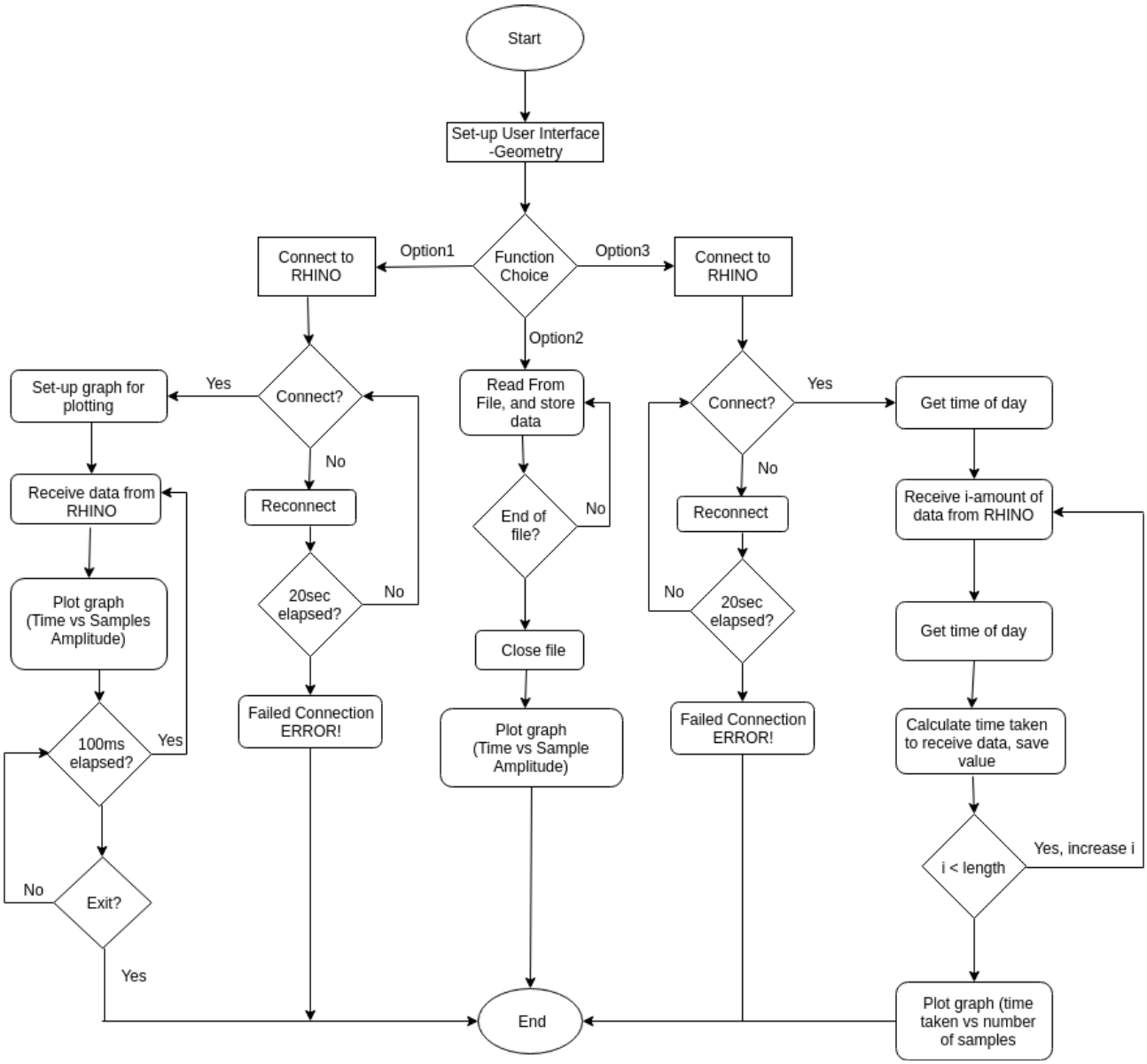


Figure 4.2: Flow diagram showing the relationship between rhinoudpserver and Mainwindow class

4.3 IMPLEMENTATION OF RHINO UDP BLOCK

GNU Radio can be extended with additional functionality by implementing new blocks using OOT. OOT module feature is used in this project to create an interface between RHINO and

GNU Radio. The following sections will present steps taken to implement the UDP source block for RHINO on GNU Radio. A flow chart of the processes within the RHINO UDP block from connection to the host, to data acquisition and data transfer to other GNU Radio blocks, will also be highlighted. The structure of the block will be outlined, showing fixed and non-fixed parameters. The RHINO UDP block was implemented using standard C++ library for socket programming.

4.3.1 **UDP Source Block**

The RHINO UDP source block facilitates the communication between the RHINO board and GNU Radio installed on a host PC. As mentioned in Subsection:3.2.3, the system has to be configurable. Therefore, it follows that the RHINO UDP source block allows the users to set certain parameters that may change based on the requirements. Parameters considered for the design of RHINO GNU Radio UDP source block are as follows:

- Port number - the UDP port number for the communication of RHINO Ethernet port and the PC's port, should not conflict with other TCP ports [14].
- Payload size - The amount of data stored in the block's buffer at a time.
- Output type - The type of output data from the block, options include; complex, float, int, short and byte.
- Sampling rate - The frequency at which the data is sampled.
- Frequency - This is the tuning frequency for the block.

The parameters decided upon can be changed according to the user's needs. Figure 4.3 shows the design structure of the source block.

GNU Radio automatically generates documentation on blocks created and stores them in the *doc/* folder which can be edited as need be, the information can be viewed under the *Doc* option as shown in Figure 4.3.

RHINO UDP source block will use the standard C++ server socket programming mode. RHINO is the client in this case providing data. Figure 4.4 demonstrates activities in the source block to facilitate its functionality. After the connection between the host PC and the RHINO board is established, the socket is created. The host PC then receives data from the RHINO for as long

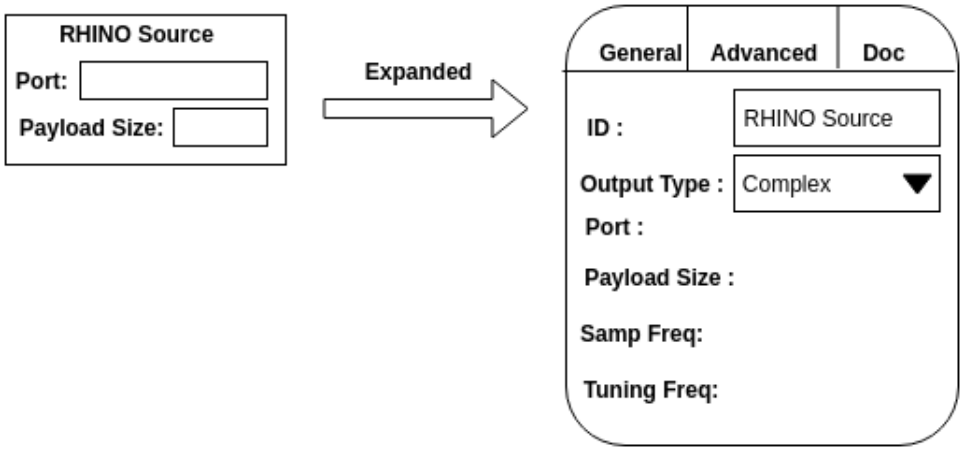


Figure 4.3: RHINO Source Block For GNU Radio

as there is a connection. In a case where a connection is not established, the system attempts to reconnect for a duration of 20 seconds and if not successful, the system comes to a halt.

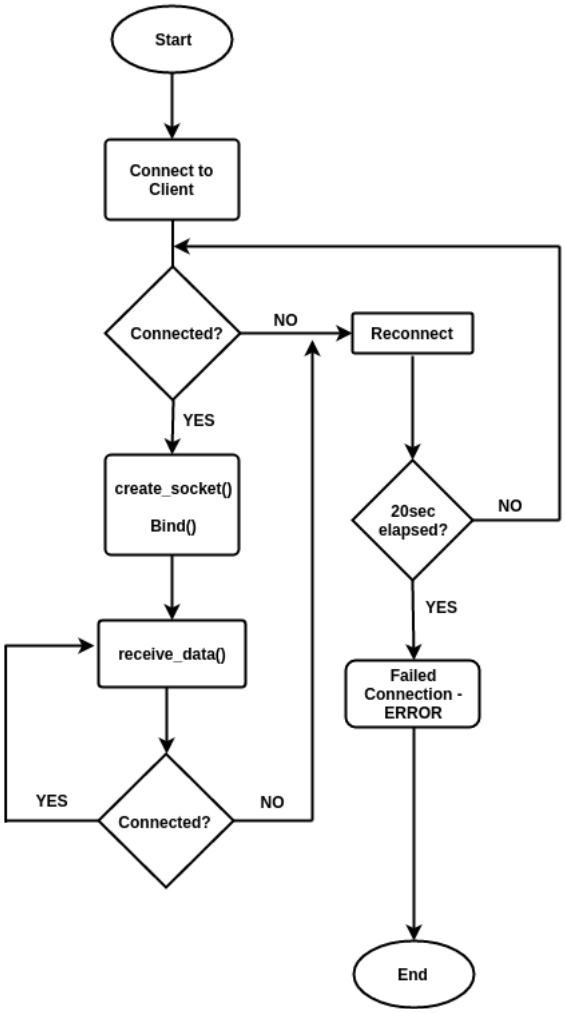


Figure 4.4: RHINO Source Block Flow Graph

When creating an OOT module in GNU Radio, the *work()* function is automatically generated. This is the function responsible for all the actions within the block. To facilitate the *receive_data ()* function, certain parameters have to be set, and buffer size defined. Figure 4.5 shows the details of the *receive_data()* function including the packet structure and how the data is received and passed on to the next block.

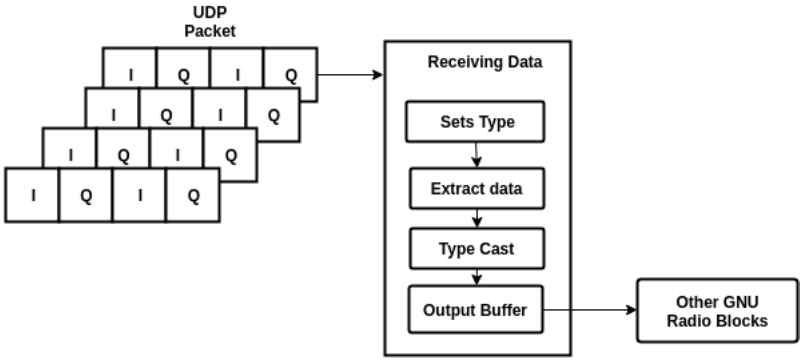


Figure 4.5: Receiving Data from RHINO

Each packet contains two I/Q pairs and the data is read one packet at a time. The size of the output buffer depends on the payload size set by the user. Output buffer then sends data to the next block in the flow graph.

RHINO UDP Source Data Accessing Technique

RHINO source block on GNU Radio uses a basic buffer for storage. Each packet is captured and stored in the output data buffer. The size of the buffer is determined by the payload size specified by the user. In the case of the source block, each packet consists of a pair of an I/Q sample which is unpacked according to the type of the output data chosen. Figure 4.6 shows two pairs of I/Q samples per packet. It should be noted at this point that the size of the packet varies according to the configurations in the RHINO FPGA firmware, details on the configurations of the firmware will be highlighted in Section 4.4.

The data sample is read as a *short* (32bits) value and in the case that output data type is *short*, it follows that there is no need to typecast. In the case of *complex* output data type, the I/Q pair produces a complex value. The output of a noncomplex data output can either produce a sine waveform (I value) or a cosine waveform (Q value). Figure 4.6 is based on the assumption of the FPGA NCO producing the I/Q samples, with two pairs per packet, I value being the sine wave samples and Q value being the cosine wave samples. The sampled data is then transferred to the connecting GNU Radio block for signal processing. As seen in Figure 4.1, some of the operations on the data includes plotting the data, performing signal processing such as modulation/demodulation or saving the information to a file.

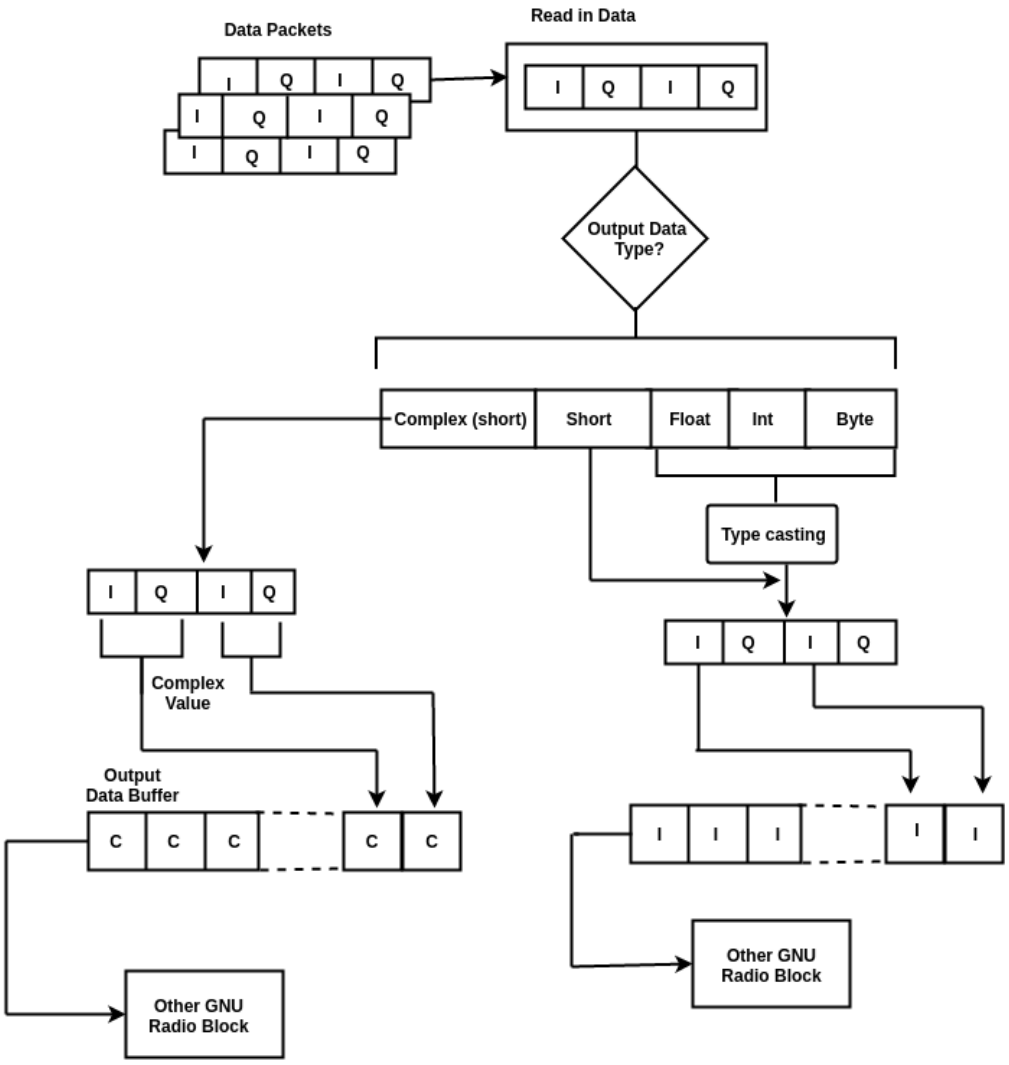


Figure 4.6: Source Block Data Access and Buffering Technique

4.4 CONFIGURING THE RHINO FIRMWARE

In this project, the RHINO IP cores which were implemented in proceeding projects such as [53] were used for real-time data acquisition and/or channelization. The real-time data acquisition using the RHINO platform requires two forms of configurations, both for the RHINO FPGA and its I/O peripherals, and the interconnection software in the MCPU system which controls and distributes data capturing and processing. In this section of the report, the RHINO FPGA programming and configuration procedure is discussed.

The focus of the project is on the I/O peripherals such as GbE and ADC cores, and the user logic such as NCO and DDC channelization firmware. The main configuration steps include; 1) Instantiation of I/O peripherals' firmware; 2) Integrating I/O instantiations with user logic, in this case, the NCO and DDC cores; and 3) Configuring the firmware parameters which include the ADC's sampling rate, the GbE transmission speed, and the NCO/DDC frequency tuning word and other relevant parameters such as number of CIC stages and rate factor.

4.4.1 Instantiation of I/O Peripherals

The FMC150 ADC/DAC board was used in the experiments as the main signal source. The inputs to the FMC150 are signals from the SIGLENT SDG1010 [43] function generator and the RF front-end presented in [51] that was used in radar applications at the University of Cape Town. The FMC150 firmware used was from [53], configured to work at the sampling frequency of 122.88 MSPS. In this firmware, the following calculations were made for use in the FM case study.

- Bandpass sampling $n = 2$ (Chosen from range of 1 to 5) based on the formula:

$$1 \leq n \leq \frac{f_H}{f_H - f_L} \tag{4.1}$$

  where $f_H = 108MHz$ and $f_L = 88MHz$, the FM bandwidth.

- Sampling frequency f_s is 122.88 MSPS based on the equation:

$$\frac{2 * f_H}{n} \leq f_s \leq \frac{2 * f_L}{n - 1} \tag{4.2}$$

- Bandpass sampling frequency range for FM: 14.88MHz to 34.88 MHz using the equation:

$$f_s modf \tag{4.3}$$

where:
f_s is the sampling frequency,
f is the frequency to be translated.

- FM Channel tuned into: 94.5 MHz (KFM Radio Station) which will translate to 28.4MHz

The data from the FMC150 ADC was passed directly to the FMC150 DAC which connects to the RIGOL MS01104 oscilloscope [35] for verification, and to the circular buffer in the FPGA as discussed in [53] as the adc-to-ethernet bridge. This buffering technique is used to eliminate timing issues related to the difference in sampling rates used in both the ADC and the GbE I/O used. The GbE has the maximum data rate of 125 MSPS for 8-bit samples, however, in order to accommodate the 16-bit I/Q samples (32-bit samples), the buffering was required in the FPGA prior to sending data to the physical GbE chip. This is necessary in order for the GbE chip to cope with the higher data rates coming from the FMC150 ADC. Alternatively, data could be sent to the DDC, which down-samples data to about 964 KSPS. Ideally, this data could be relayed to the GbE through the DDC effectively without data losses. However, an efficient way to transmit data to the PC via GbE was to buffer and send as a packet of 8 to 32 bytes.

4.4.2 The NCO and DDC Cores

The NCO is used in this project to generate sinusoidal waveforms where a digital accumulator is used in order to generate the address into a sine/cosine lookup table. Different waveforms can be generated by adding other functions in the lookup table, however, for purposes of this project, an NCO of RAM size 256KB was used to generate sine and cosine waveform. The phase step of the digital accumulator is determined by the FTW which can be calculated as follows:

$$FTW = \frac{f_{out} * 2^{32}}{f_{clk}} \tag{4.4}$$

Where:
f_{out} is the output frequency
f_{clk} is the clock frequency

The NCO translates the incoming signal to baseband by mixing it with the corresponding channel frequency based on the value of the FTW. The output signal is then relayed to the DDC core,

where the sampling frequency is decimated by factor 128 resulting in a sampling frequency of 960KSPS. Therefore, it follows that the I/Q samples at the GbE core through to the connecting PC are sampled at this value.

4.5 EXPERIMENTS DESIGN

This section of the chapter will highlight the designs of the different experiments carried out to test the functionality of the block designed (RHINO UDP source), and finally the system as a whole. The designs also contribute to the evaluation and verification of the objectives of the system as mentioned in Chapter 1 Section 1.4. The system design needs to meet the following requirements: reliable connectivity and data capturing, low latency and high throughput.

4.5.1 FPGA NCO Experiment

This experiment is designed to evaluate the data capturing ability of the C/C++ program using QT. The C/C++ program is also meant as an extensible baseline for communicating with RHINO and can be built upon further for other applications besides GNU Radio. The code is implemented in developing the RHINO UDP source block. Synthetic sinusoidal data from the FPGA NCO is transmitted through a 1GbE connection to the host PC. The samples are captured and plotted using standard C++ QT plotting program in real time. Figure 4.7 shows the design of the experiment.

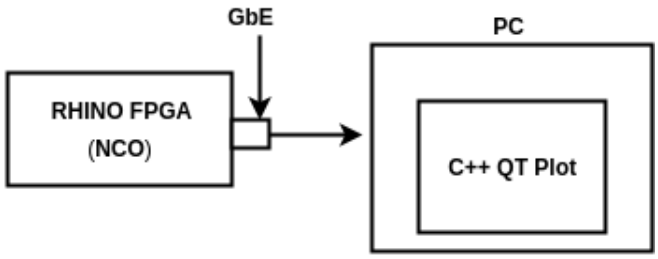


Figure 4.7: FPGA NCO connecting with C++ program on PC

4.5.2 RHINO GNU Radio UDP Source Block Test Design

The test designed for the RHINO GNU Radio block is intended to evaluate the functionality and reliability of the module on the GNU Radio software. The FPGA NCO is used in this case to generate a sinusoidal waveform which will be captured using the RHINO source block and the plots created on GRC. Figure 4.8 demonstrates the test designed to test the source block.

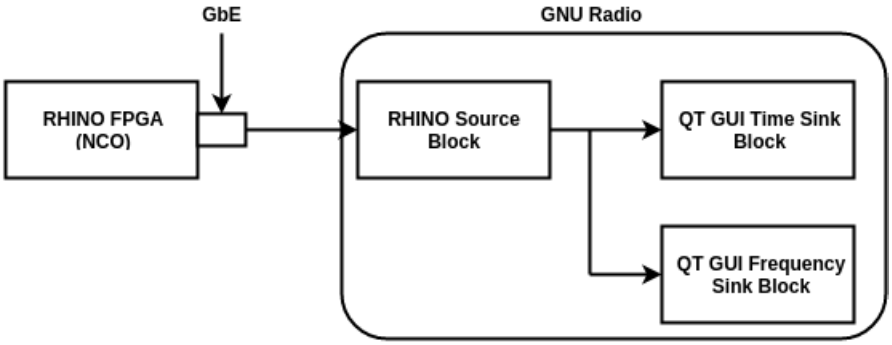


Figure 4.8: RHINO GNU Radio Source Block tested with GRC

4.6 CASE STUDY DESIGN

The case study is designed to test the system as a whole, the communication between the different components of the streamer and the results are verified and compared with the findings. Figure 4.9 shows the case study designed to test the RHINO streamer.

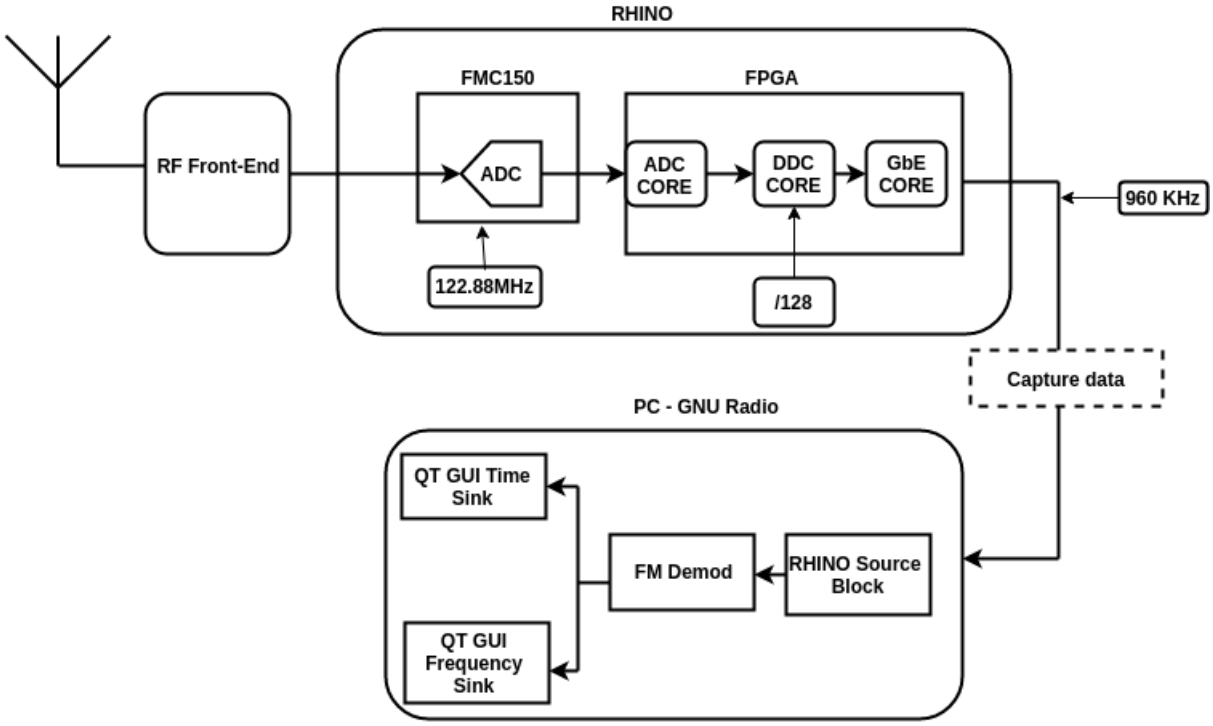


Figure 4.9: FM signal receiver chain test on GNU Radio

The design consists of an RF front end to capture radio signals, the RFFE circuit receives FM signals in the range 88-108MHz, the signal is then passed through a bandpass filter. The total RFFF gain amounts to 75dB, this includes the antenna gain and the additional amplifiers in the circuit. The FM signal is sampled by the ADC at 122.88MHz, then decimated by the DDC with

a factor 128 resulting in UDP samples at 960KHz. The I/Q samples are transmitted over the 1GBbE to the host PC. The samples can be captured for processing or streamed directly to the host PC. The FM receiver using RHINO as data acquisition platform results will be compared with the results obtained with the FM receiver using the RTL-SDR dongle.

4.7 SUMMARY

This chapter highlighted the system design of the RHINO streaming interface for GNU Radio, a data acquisition and transmission system. The streamer is made up of a firmware ready RHINO integrated with GNU Radio on a host PC. The essential firmware for this system is the ADC core, DDC core and the GbE core which all assist in the data transmission of data and are discussed in Section 4.4. The chapter thereafter highlights the implementation of the RHINO OOT modules, UDP source on GNU Radio. Furthermore, the processes involved in data capturing of the block are discussed. Finally, the designs of preliminary experiments and the case study test are elaborated. Results of these experiments will be covered in the next chapter.

CHAPTER 5

RESULTS AND DISCUSSION

This chapter outlines the evaluation of the experiments conducted to test the system, an elaborate design of these tests was outlined in Section 4.5. This chapter will also indicate the results obtained and detailed discussion will follow. The evaluation of the system is based on the system connectivity and data capturing, the reliability of the system, latency and maximum throughput obtainable as outlined in the section that follows. It follows that the experiments are designed in the manner that investigates these factors.The chapter will further discuss the final test of an FM receiver chain using the RHINO with GNU Radio. The results to this experiment will be compared to those of an FM receiver chain using an RTL-SDR dongle with GNU Radio as benchmark for the FM receiver.

5.1 EVALUATION METHODOLOGY

This section will highlight the evaluation methods used in this project as briefly mentioned in Section 3.6. This section will present further details on the evaluation methodology used in the project. The success of the system was evaluated based on the following factors:

- System connectivity and data capturing
- Latency and throughput
- Reliability of the system

The experiments designed in Section 4.5 and preliminary experiments in Subsection 3.6.1 were used to evaluate the system based on the factors mentioned above, and the case study to investigate the functionality of the system as a whole. Section 5.2 through to Section 5.4 present an elaborate discussion on these factors based on the experiments conducted.

System connectivity and data capturing refers to the ability of the system to maintain a consistent connection, that is, the connection between the host PC and the RHINO board hence the consistent transmission and capturing of data. The test for this functionality was tested with both an independent C++ program and with the RHINO UDP source block on GNU Radio. The latency of the system in sending data from the RHINO board to the host PC is measured in seconds while the throughput in bytes per second. The reliability of the system was tested by comparing the results of capturing data using the inbuilt UDP source block in GNU Radio with the RHINO UDP source block.

5.2 System Connectivity and Data Capturing Tests

This section will illustrate the tests conducted to test the connectivity of RHINO to GNU Radio. Furthermore, the tests will evaluate the data capturing capability of the RHINO UDP blocks on GNU Radio. The synthetic data used in these experiments is a 10MHz sine waveform, a 32bits packet consisting of an I/Q sample. The results will be discussed in each section and conclusions will follow in the next chapter.

5.2.1 Preliminary experiments with GNU Radio

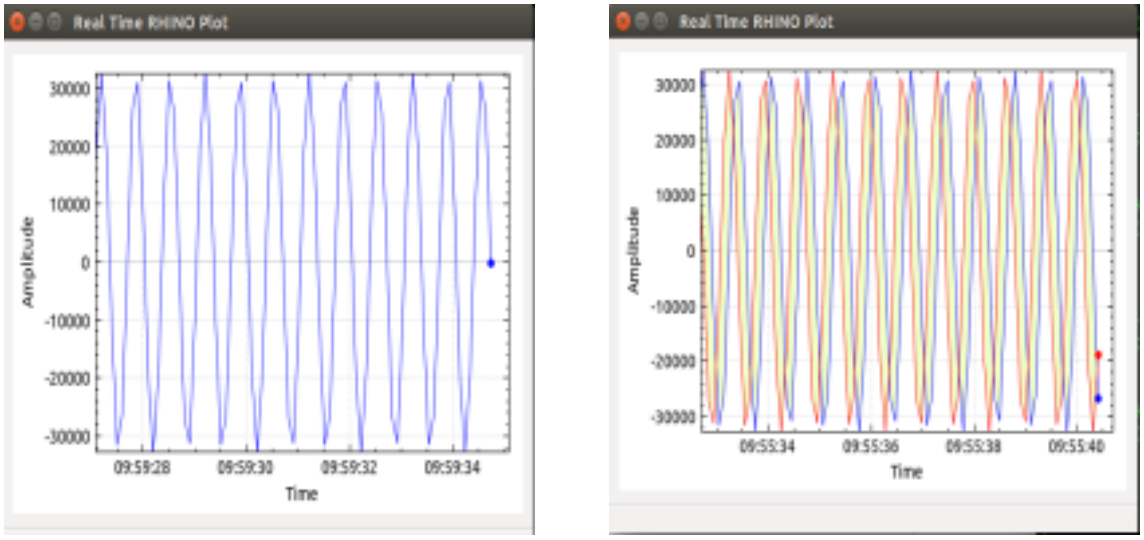
Subsection 5.2.1 gives an outline of the experiments designed to test the software on its own. The results obtained for the peer-peer connection were as expected, the data was received as is from the sending side. The same results were seen for the experiment done over the UCT network. However, there is a little discrepancy due to the fact that the data is not transmitted directly to the client but over the network. The latency, in this case, is a bit higher as compared to the peer-peer connection. In conclusion data transmission over the network poses some challenges.

The GNU Radio inbuilt UDP Source block was tested with real-time streaming data from the RHINO. This proved to be very unreliable as many packets were dropped which resulted in the loss of data. It follows that a RHINO specific UDP block is necessary, a block that is specifically designed to capture data from the RHINO board. Understanding the structure of data coming in from RHINO is, therefore, important in implementing the GNU Radio block that will function with the RHINO board reliably.

5.2.2 FPGA NCO with C++ program Test

This test was designed to test the communication of RHINO FPGA with an independent C++ program running on a PC using synthesized data. The NCO was configured to generates sine wave samples. Details on the NCO configurations can be seen in Section 4.4. The C++ program

uses UDP data transmission protocol and plots the data in real time. Details on the code can be found in Appendix B. The results shown in Figure 5.1 depicts a plot of a real-time stream of data from RHINO over the 1GbE.



(a) Real Value RHINO data plot

(b) Real and Imaginary RHINO data plot

Figure 5.1: RHINO NCO Plot on QT C++ Program

As depicted in Figure 5.1, data is plotted in real time and the results are consistent with the expected sinusoidal plot, Figure 5.1a shows a real value plot while Figure 5.1b shows the imaginary plot for RHINO NCO 10MHz data samples. It follows that this test was successful as it meets the requirement stipulated, which is to consistently capture data from the RHINO board to the host PC.

5.2.3 **RHINO UDP Source Block Test**

This test was designed to test the functionality of the RHINO UDP source block, the ability for the block to interface with RHINO and compatibility with other GNU Radio source tree blocks. Table 5.1 shows the parameters set on the block for this particular experiment.

Table 5.1: Parameters set on the RHINO UDP Source block

Parameter	Value
Output Type	Complex
Port	9930
Payload Size	1472

The block diagram on GNU Radio for the experiment shown in Figure 5.2. It shows the RHINO source block connecting to the *complex To Float* block which as the naming states, separate the complex input into real and imaginary part, the real part of the block is then connected to the *QT*

GUI Time sink and *QT GUI Frequency sink* blocks. The blue on the connector denotes *complex* output type while the orange denotes *float* as explained by Figure 2.13. The block can produce other data types such as float, short and int. Another *QT GUI Time sink* block is connected directly to the RHINO Source block to display the complex output. The important settings on the *QT GUI Time Sink* blocks is input data type set to *complex* and *float* similar to the output data type from the connecting block, the maximum and minimum amplitude value which will depend on the amplitude of the sine wave.

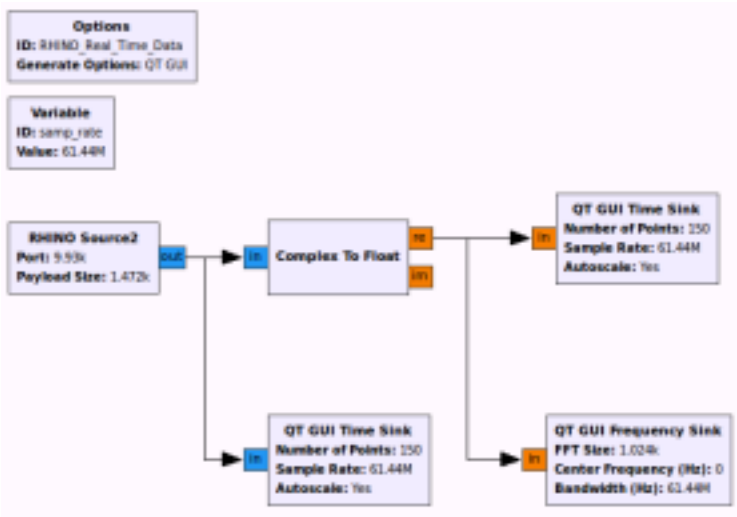
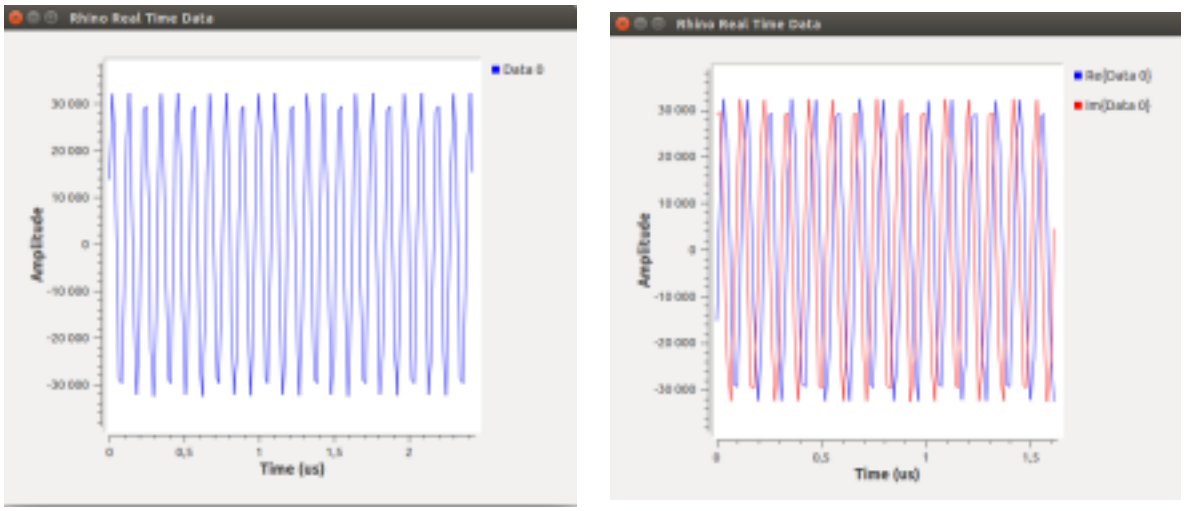


Figure 5.2: Flow Diagram For Testing RHINO GNU Radio UDP Source Block

As depicted in Figure 5.3, the sinusoid produced matches the properties of the sine waveform data generated by the NCO. The prominent signal seen in Figure 5.4 is at frequency 9.108MHz (approximately 10MHz) as seen in . Therefore, it follows that the RHINO UDP source functionality meets the specifications, consistent connectivity and data capturing. The block captures the stream of data from the RHINO board and is capable of relaying the data to the connecting blocks.

5.3 Latency and Throughput Tests

The latency test was conducted by generating data from the RHINO FPGA and capturing a fixed number of samples using the C++ program running on a PC. This test determines the latency in requesting the data and receiving the data on the PC. The experiment was run five times for each set of a number of samples. The same results attained with this experiment can be translated into results for the RHINO UDP block on GNU Radio since the same code was used to build the block. Independent C++ code is easy to debug and trace performance. This is the reason the latency and throughput tests were performed in this manner. Table 5.2 shows the time taken to capture a different number of packets (2^N - where N is numbers in the left column), with each



(a) Real Value RHINO data plot

(b) Real and Imaginary RHINO data plot

Figure 5.3: GNURadio Real time plot from NCO data

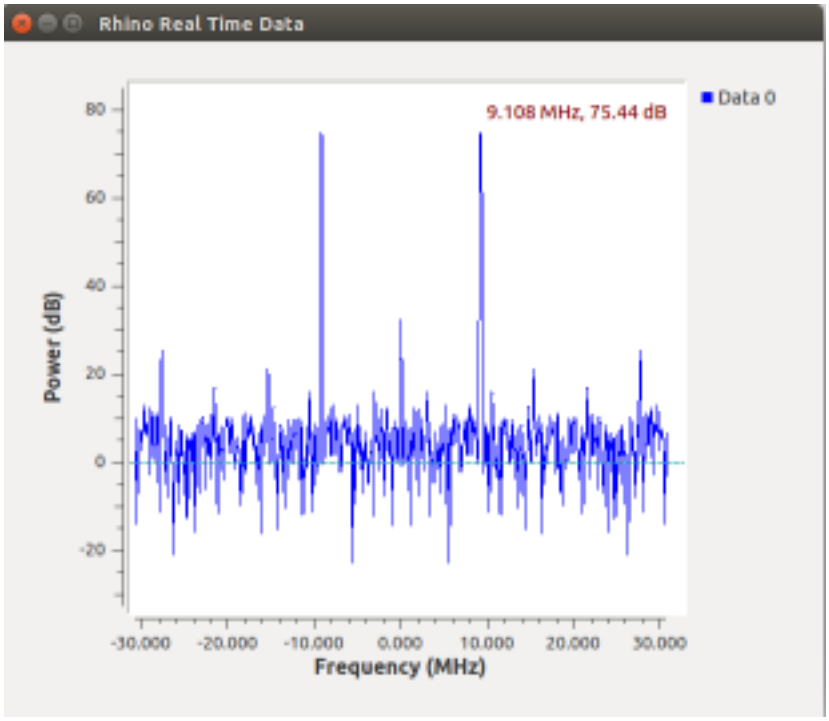


Figure 5.4: GNURadio Real Time Frequency plot of sampled data at 9.108MHz

number of packets run five times.

Table 5.2: The Relation Between Number of Samples Captured and Time taken

2^N # Packets (N =)	Time1 (sec)	Time2 (sec)	Time3 (sec)	Time4 (sec)	Time5 (sec)
0	$6 * 10^{-6}$	$5 * 10^{-6}$	$2.9 * 10^{-5}$	$6 * 10^{-6}$	$5 * 10^{-6}$
2	$3 * 10^{-6}$	$5 * 10^{-5}$	$2.2 * 10^{-5}$	$2 * 10^{-6}$	$2.8 * 10^{-5}$
4	$2.24 * 10^{-4}$	$2.35 * 10^{-4}$	$1.41 * 10^{-4}$	$1.89 * 10^{-4}$	$1.96 * 10^{-4}$
8	$3.6 * 10^{-3}$	$3.55 * 10^{-3}$	$3.60 * 10^{-3}$	$3.60 * 10^{-3}$	$3.62 * 10^{-3}$
16	0.92	0.93	0.93	0.93	0.93
18	3.7	3.71	3.71	3.71	3.71
20	14.84	14.84	14.83	14.84	14.84
21	29.67	29.68	29.68	29.67	29.68

The standard deviation (SD) is an index that indicates the variability of multiple data values, as a result of repeating an experiment, from the mean value [49]. Using the values in Table 5.2, standard deviation for each number of packets (2^N) can be calculated as follows:

$$\sigma = \sqrt{\frac{\Sigma(x - \overline{x})^2}{N}} \tag{5.1}$$

where: σ is the standard deviation, x is each value,
$\overline{x}$ is the mean value,
N is the number of values.

Table 5.3 shows the average time taken for each number of packets and the standard deviation for each set. A low standard deviation means that the range of values in a particular repeated experiment is close to the average value, which translates to the minimum error in the readings. The standard deviation calculations in Table 5.3 show values close to zero. It follows that the values taken are close to the average value for each set of readings.

Table 5.3: The Average Time Taken to Capture 2^N packets

2^N # Packets (N =)	Average Time (sec)	Standard deviation (σ)
0	$1.02 * 10^{-5}$	0
2	$2.1 * 10^{-5}$	0
4	$1.97 * 10^{-4}$	$3.16 * 10^{-5}$
8	$3.6 * 10^{-3}$	$3.16 * 10^{-5}$
16	0.93	$4.47 * 10^{-3}$
18	3.71	$4.47 * 10^{-3}$
20	14.84	$4.47 * 10^{-3}$
21	29.68	0.01

Table 5.3 is translated into a plot shown in Figure 5.5. The aim of this experiment is to demonstrate different throughputs for different sizes of data. Figure 5.5 demonstrates a linear relationship between the time taken to capture data and the increase in number of packets sent to the host PC.

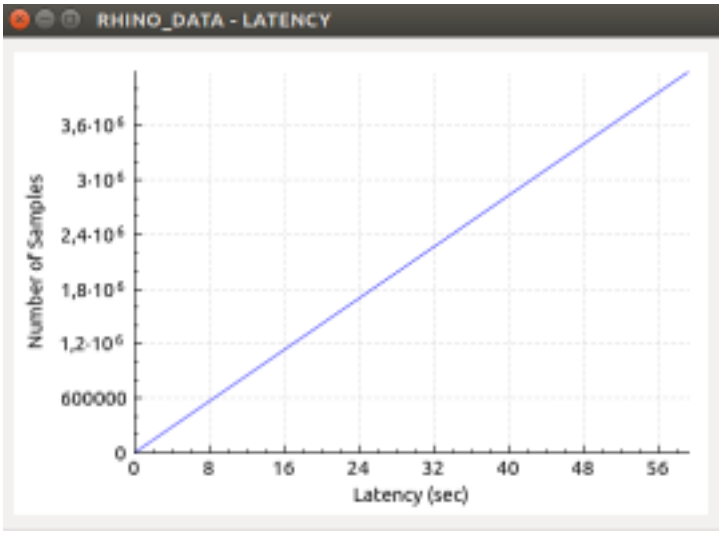


Figure 5.5: Latency Plot of Samples Captured on PC

The throughput table can be deduced from the values in Table 5.3. Each packet consists of a 32 bits I/Q pair sample and 16 bits zero padding which results in a 48 bits packet, therefore, it follows that the data throughput for each number of samples captured can be calculated as follows:

$$Datathroughput = \frac{Numberofpackets * 48bits}{8bits/Bytes * Timetake} \tag{5.2}$$

Table 5.4 shows the relation between the size of data captured and time taken.

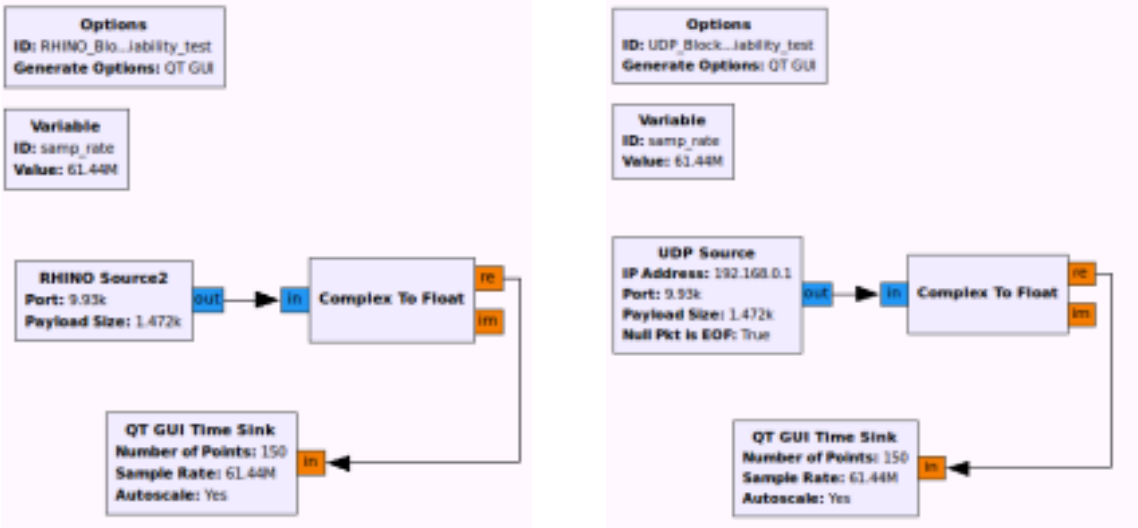
Table 5.4: The Relation Between Size of Samples Captured and Time taken

Data size (Bytes)	Time (sec)	Throughput (MBps)
6	$1.02 * 10^{-5}$	0.59
24	$2.1 * 10^{-5}$	1.14
96	$1.97 * 10^{-4}$	0.49
1536	$3.6 * 10^{-3}$	0.43
$3.93 * 10^{5}$	0.93	0.43
$1.57 * 10^{6}$	3.71	0.42
$6.29 * 10^{6}$	14.84	0.42
$1.26 * 10^{7}$	29.68	0.42

It follows that the average throughput for capturing data is : 0.54MBps.

5.4 RELIABILITY TESTS

The reliability test was based on the reliability of the RHINO UDP source block on GNU Radio in capturing data as compared to the inbuilt UDP source block in GNU Radio. The test was conducted in the following manner: NCO generated data was sent to the PC and captured using inbuilt GNU Radio UDP block, the same data was captured using the RHINO UDP source block and the results plotted. Figure 5.6 shows the block diagrams for both the inbuilt UDP source block experiment and RHINO source block.

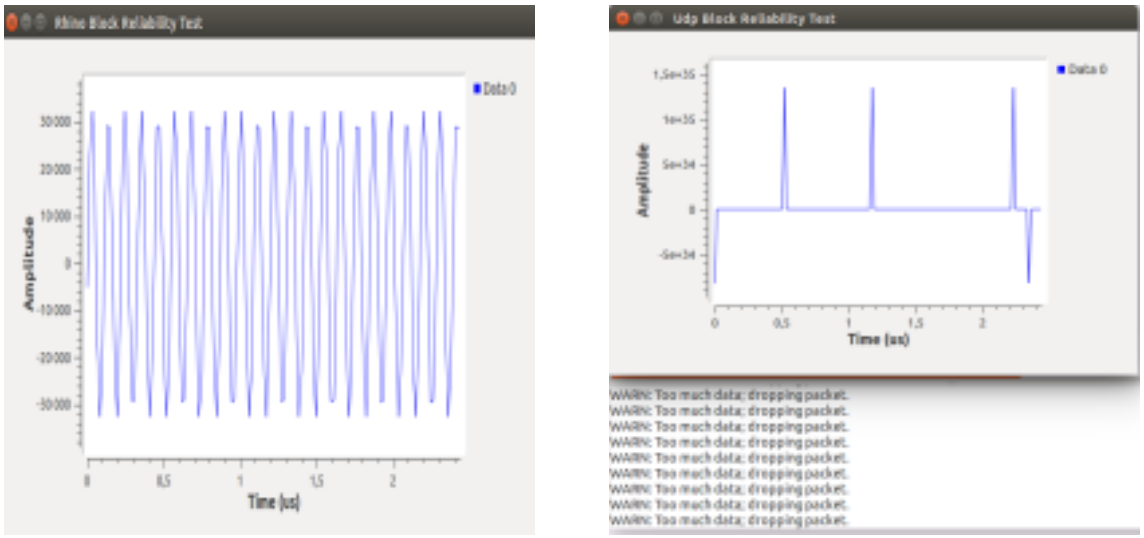


(a) RHINO UDP block Test Flow Diagram (b) GNURadio UDP Block Test Flow Diagram

Figure 5.6: Reliability Tests for Data Capturing on GNU Radio

Figure 5.7 shows the experiment results for both the RHINO UDP source block and the GNU Radio inbuilt UDP source block.

Figure 5.7b show a lot of packets are dropped when using the inbuilt UDP source block and not with the RHINO UDP source block as seen in Figure 5.7a. The reason being that the GNU radio UDP block is not optimised for RHINO data and so will not be reliable for this application. On the other hand, RHINO UDP source block is reliable in capturing data. However, the RHINO streaming system is only reliable for a certain number of samples captured. With prolonged streaming, some glitches can be witnessed on the plots. Tests have shown that for data samples of packet size 32 bytes and above, the application starts losing samples after capturing approximately 1000 samples. This is mainly due to minor issues with the firmware. More investigations and work on the firmware is necessary.



(a) RHINO UDP block Reliability Test plot (b) GNURadio UDP Block Reliability Test Plot

Figure 5.7: Reliability Tests for Data Capturing on GNU Radio

5.5 CASE STUDY: SDR FM RECEIVER

The case study as described in Section 4.6 is an FM receiver implemented using an RFFE connected to the RHINO board. RHINO acts as the signal acquisition platform which transmits the data to the host PC in the form of I/Q samples centred at DC. The case study was conducted with the hardware tuned to a 94.5MHz radio station and a total front end gain of 75dB. Figure 5.8 shows the block diagram chain implemented on GNU Radio to receive the data for signal processing.

The I/Q samples from the RHINO board are received by the RHINO source block on GNU Radio. Figure 5.8 shows a similar block diagram set-up to that in Figure 2.16 for the RTL-SDR FM receiver, however the FM receiver is tuned to a 94.5MHz radio station in this case. The rational resampler block interpolates and decimates the I/Q samples from the RHINO source block and relays them to the WBFM receiver block. This block then demodulates the samples and sends them to another rational resampler block for more interpolation and decimation to an audible frequency. The next block is the multiplying block, which multiplies the samples by a constant and sends them to an audio sink block and an FFT sink block. The audio sink block makes it possible to hear sound by sinking the signal to the PC sound card.

The results of the RHINO FM receiver are compared to the results of an FM receiver using the RTL-SDR dongle. The RTL-SDR dongle was tuned to the same frequency as that of the RHINO FM receiver experiment, 94.5MHz radio station and a 40dB gain. Figure 5.9 shows results for signals directly from the RTL-SDR dongle and the RHINO board respectively. Figure

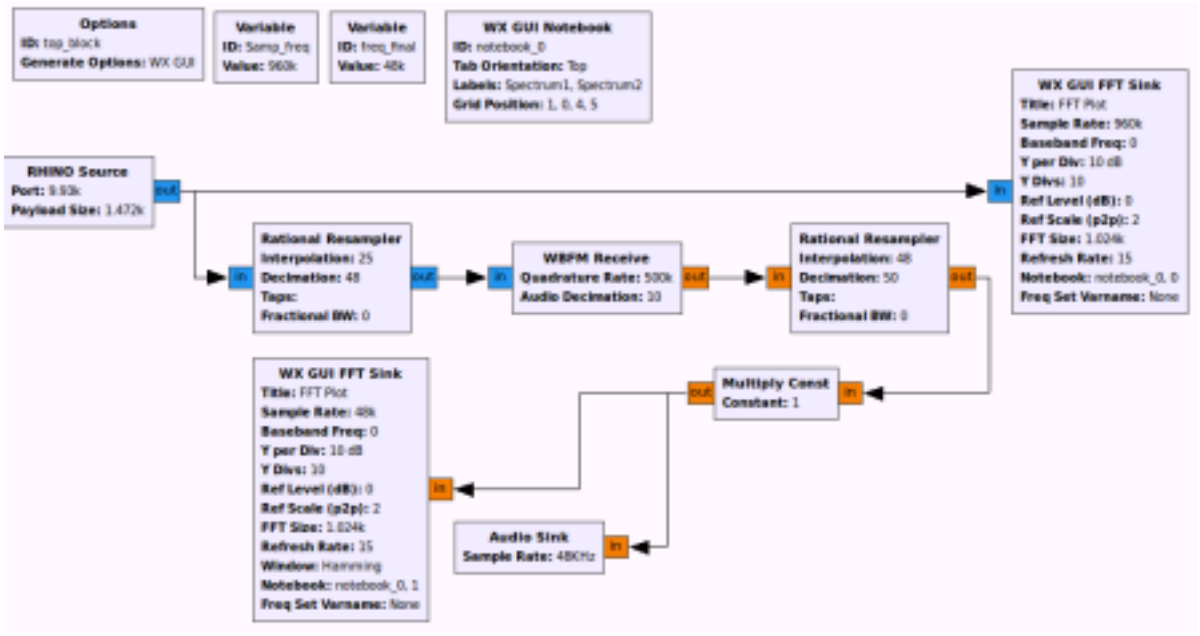
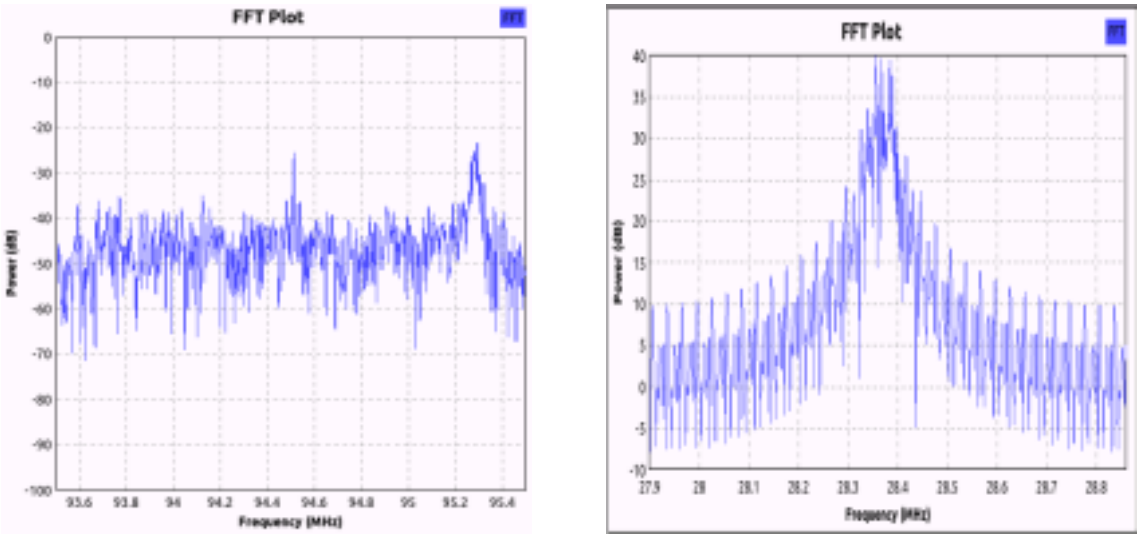


Figure 5.8: Block Diagram chain for FM receiver on GNU Radio

5.9a shows that the dongle is picking up a signal at 94.5MHz and 95.3MHz. However, the experiment concentrates on the center frequency which is 94.5MHz . Figure 5.9b shows a peak at frequency 28.4MHz instead of 94.5MHz because of the bandpass sampling. Using equation 4.2, a 94.5MHz signal will be translated to a 28.4MHz frequency. The RHINO had a total front-end gain of 75dB, hence the difference in power values between the RTL-SDR signal results and those of the RHINO signal.

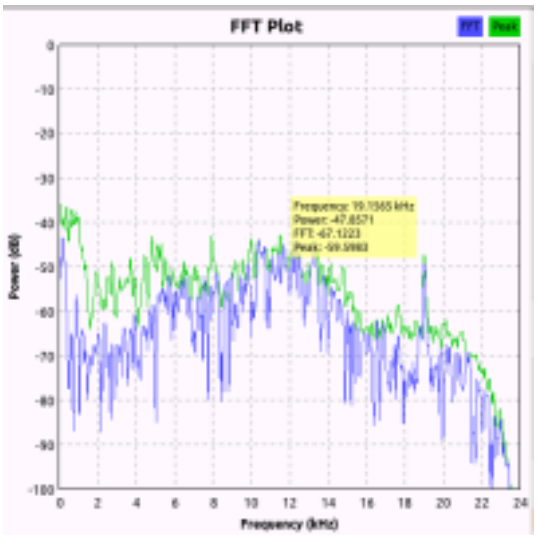


(a) RTL-SDR dongle signal output

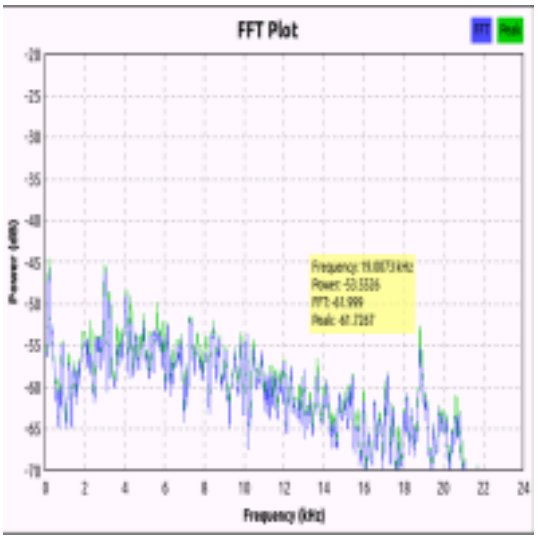
(b) RHINO signal output

Figure 5.9: FM signals directly from the RTL-SDR dongle and RHINO on GNU Radio.

Figure 5.10 shows the results of the demodulated FM signals for both the RTL-SDR dongle and RHINO FM receiver. The results of the demodulated FM signal show some similarities to the ideal spectrum of a baseband FM station [34]. The stereo pilot is clearly shown in both cases at 19 MHz, however, the other spectral components are not clearly seen due to a weak FM signal.



(a) RTL-SDR dongle demodulated signal output



(b) RHINO demodulated signal output

Figure 5.10: FM demodulated signals for the RTL-SDR dongle and RHINO FM receiver tuned to 94.5MHz radio station.

Table 5.5 details more comparisons between the two FM receivers.

Table 5.5: Comparison between RHINO FM receiver and RTL-SDR dongle

	RHINO	RTL-SDR Dongle
RFFE	limited gain	Programmable gain (maximum of 45dB) tunable frequency (24 - 1766MHz)
Sampling	Bandpass sampling	Direct down sampling
Connectivity	GbE	USB 2.0

One of the limitations with the RHINO FM receiver is the ability to set the gain and tunable frequency on the source block. To set this values, the block should be able to send this information to the FPGA at the same time receive the data samples from RHINO after the values have been set. Further development work is required to make this realisable. On the other hand, RTL-SDR dongle provides programmable gain and tunable frequency on its source block.

RHINO system uses bandpass sampling as compared to RTL-SDR dongle which uses direct down sampling. This makes the dongle more effective than the RHINO system. However,

the RHINO FM receiver is faster than the RTL-SDR dongle system because it uses the GbE connectivity with an achievable throughput speed of 122.59 MBps [53] and USB2.0 with a speed of 60MBps [19].

5.6 SUMMARY

This chapter highlighted the results and discussions for the project. It started off with a brief preview of the evaluation methodology in Section 5.1. The evaluation methodology is then further elaborated in the next sections of the chapter. Section 5.2 illustrates experiments conducted to evaluate the system connectivity and data capturing ability of the system. The experiments include sending synthetic data from the FPGA NCO and capture with the C++ QT program. One other experiment tests the RHINO UDP source block on GNU Radio. Results are highlighted and discussed. Section 5.3 evaluates the latency and throughput of the system. The experiment evaluates the time taken to capture a different number of samples on the host PC using the C++ QT program. Each experiment is run multiple times and standard deviation calculated for each set of values. Based on the average time values, data throughput is calculated. Section:5.4 discusses the reliability of the system by comparing the time plots of samples captured with RHINO UDP block and the inbuilt GNU Radio UDP block. Section 5.5 finalises the chapter by discussing a case study of an FM receiver using an RFFE, the RHINO board and GNU Radio on a host PC. The plot results using RHINO source block are compared to those obtained when using RTL-SDR dongle.

CHAPTER 6

CONCLUSIONS AND FURTHER WORK

This chapter presents the conclusions based on the experiments and results highlighted in Chapter 5. The conclusions have been aligned to reviewing the systems requirements that were met and reflecting on the objectives of the project which were described in Chapter 1 of the dissertation. Recommendations for further work will also be highlighted in this chapter.

6.1 CONCLUSIONS

This section of the write-up will briefly review the system requirements described in Section 3.2.3 and discuss the conclusions based on the experiments designed and conducted to test both the independent C/C++ application and the RHINO UDP block on GNU Radio. Based on Section 2.5, it was determined that GNU Radio is the best suited tool to be integrated with RHINO. The tests conducted in Chapter 5 to test the integration of RHINO and GNU Radio as well as test the C/C++ API were all successful.

6.1.1 Systems Requirements

A recapitulation of the system requirements as explained in Subsection 3.2.3 is as follows:

Functional Requirements

- 1Gbps Ethernet streamer
- Receive data as baseband signals
- Real-time streaming data acquisition system
- Compatibility with an SDR software tool
- Storing streamed data for later processing

Non-functional Requirements

- Minimum loss, minimum latency, maximum throughput
- Configurable
- Reliable
- User-friendly

The system successfully streamed data representing baseband signals (I/Q samples) over a 1Gbps Ethernet from the RHINO board to the host PC as per the first system requirements with a maximum latency of 3.6ms for 256 packet size. All the experiments illustrated in Section 5.2 through to Section 5.4 were conducted with data streaming over a 1Gbps Ethernet. The third system requirement was for the system to function in real time, real time streaming. The experiments in Section 5.2 also illustrated this functionality, by streaming data over Ethernet in real time. Therefore, we can conclude that the RHINO streaming interface for GNU Radio system can successfully stream baseband signals over the 1Gbps Ethernet in real-time as required.

The RHINO UDP block on GNU Radio communicates effectively with the source tree blocks on GNU Radio software framework, this is demonstrated with the experiment in Subsection 5.2.3. The experiment tests the system connectivity and data capturing ability of the RHINO UDP block, the data samples are transmitted from the RHINO board onto GNU Radio installed on the host PC. The data samples are then plotted in real time. This is facilitated by the QT GUI Time sink and the QT GUI Frequency sink blocks. The RHINO UDP block relays the data to both these blocks successfully as demonstrated by the plots in Figure 5.3 and Figure 5.4. The samples can also be stored to a file for later processing. Therefore, it follows that the RHINO UDP block does, in fact, communicates effectively with the source tree blocks on GNU Radio and stores data as required.

One of the non-functional requirements for the RHINO streamer was to function with the minimum loss of data, minimum latency, and maximum possible throughput. Reviewing the plots in Figure 5.7, it can be seen that the built-in GNU Radio UDP drops a lot of packets as demonstrated by the warning message *WARN:Too much data;dropping packet*. This is because the block is not specifically built for the RHINO data as is the RHINO UDP block. Section 5.3 highlighted the tests conducted to evaluate the latency of the system in terms of capturing data. The system can capture 2^{21} number of packets in 29.68sec (packet captured one packet after

the other and stored in a vector). As seen from the questionnaire in Appendix A, the maximum tolerable latency is 100ms. The system is more effective if a total of 1-256 packets are captured and send off for processing. This means that certain number of packets are captured and stored in a vector, then sent off for processing, and the process is repeated continuously as the system is a continuous real-time streaming system. In the case of capturing 1-256 packets, it takes a maximum of 3.6ms as seen in Table 5.3. The maximum throughput attainable in capturing data on the PC is 1.14MBps with data size of 24 bytes for 2^2 number of packets, each packet with two pairs of I/Q samples.

Subsection 4.3.1 highlighted the design of the GNU Radio RHINO UDP block, showing all the parameters that can be set by the user during run time,which makes the block configurable. From the user's perspective, the block is easy to configure and use. It conforms with the user-friendliness of the blocks in GRC. The RHINO UDP block is also reliable as demonstrated by results in section 5.4. As a conclusion, RHINO streamer has met all the requirements previously stated in Section 3.2.

Section 5.5 presented the FM receiver case study. The experiment was performed using base-band I/Q data of an FM station, 94.5MHz, captured using the RHINO source block on GNU Radio and relayed through a chain of DSP blocks for demodulation. The results demonstrated similarities to the results obtained when using the RTL-SDR dongle.

As a conclusion, the RHINO streaming framework has proved to be successful for the most part. However, improvements can be made. The following section highlights some of the recommended improvements that can be implemented on the system.

6.2 RECOMMENDATIONS FOR FURTHER WORK

The following are recommendations for future work on the system:

- Improving the functionality of the system.
- Use of 10GbE port on RHINO.
- Different data accessing patterns on the RHINO UDP block.
- Refining the FM receiver.

At the moment, the 1GbE streaming interface functions at 95% functionality and there is room for improvement. The system streams data continuously for an estimate of 1000 samples for

a 32bit packet size after which the system experiences some glitches. The recommendation in this regard is to conduct further investigations into the RHINO firmware, the I/O cores and/or the 1GbE core. Further, in terms of improving the functionality of the framework, it is planned that the following features will be implemented:1) Sending configuration commands (sampling frequency, streaming rate, tuning frequency) from the PC to the FPGA before runtime. 2) Extend the framework to include multiple PCs. 3) Stream data back and forth from the PC to the RHINO.

The streamer functions over a 1Gbps port on the RHINO board. However, as mentioned in Section 2.2, the RHINO board also has the 10Gbps port. At the moment, the interface does not have support in terms of firmware for functionality. An improvement to the speed of the streamer would be to implement an I/O core interface for the 10Gbps port. The RHINO UDP block on GNU Radio would not require much improvement to capture data from the 10Gbps port, this would improve the system's streaming speed.

Different data accessing patterns could be implemented to improve the functionality of the RHINO UDP block. The block uses a single buffer to store the data samples from the RHINO board. A matrix buffer could be implemented to decrease the delay in the transmission and relaying of data from the RHINO UDP block to the next block in GNU Radio.

The FM receiver can be improved in a number of ways. By using a better RFFE to improve and refine the FM receiver to play audio. A fully dedicated FM antenna can give a better bandwidth for the receiver and improve the audio. The antenna that was used for this experiment is a lab-based antenna. As a result, it posed some challenges. The RFFE can also be improved by increasing the gain. Moreover, the gain could be a programmable gain which would eliminate further physical components.

BIBLIOGRAPHY

[1] ALACHIOTIS, N., BERGER, S. A., AND STAMATAKIS, A. Efficient pc-fpga communication over gigabit ethernet. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* (2010), IEEE, pp. 1727–1734.

[2] ALHASAN, R. Software defined radio.

[3] BEHROUZ A. FOROUZAN, S. C. F. *DATA COMMUNICATIONS AND NETWORKING-Fourth Edition*. The McGraw-Hill Companies. Inc, 2007.

[4] BOEHM, B. W. A spiral model of software development and enhancement. *Computer 21*, 5 (1988), 61–72.

[5] BOWICK, C. *RF circuit design*. Newnes, 2011.

[6] BRIDGES, M. Tittle: Rhino sdr transeceiver. Master's thesis, University of Cape Town, 2014.

[7] CHIRISERI, V. E. Tittle: Rhino arm cluster control management system. Master's thesis, University of Cape Town, 2014.

[8] DER, L. Frequency modulation (fm) tutorial. *Silicon Laboratories Inc* (2008).

[9] DOYLE, L. *Essentials of cognitive radio*. Cambridge University Press, 2009.

[10] ETTUS RESEARCH, A. N. I. C. Implementation of a simple fm receiver in gnu radio. https://kb.ettus.com/Implementation_of_a_Simple_FM_Receiver_in_GNU_Radio#UHD_Source_Block, jul 2016. [Online; Accessed on: June 2015].

[11] FANG, F., HOE, J. C., PÜSCHEL, M., AND MISRA, S. Generation of custom dsp transform ip cores: Case study walsh-hadamard transform. *Proceedings of HPEC* (2002).

[12] FORUM, W. I. What is software defined radio. http://www.wirelessinnovation.org/introduction_to_sdr, jul 2016. [Online; Accessed: June 2016].

[13] GANDHIRAJ, R., RAM, R., AND SOMAN, K. Analog and digital modulation toolkit for software defined radio. *Procedia Engineering 30* (2012), 1155–1162.

[14] GAY, W. W. *Linux socket programming: by example*. Que Corp., 2000.

[15] GNU.ORG. Gnu octave. https://www.gnu.org/software/octave/, 2016. [online; Accessed on :October 2016].

[16] GNURADIO.ORG. Welcome to gnu radio. http:http://gnuradio.org/, jul 2016. [Online; Accessed on: March 2015].

[17] GRAYVER, E. *Implementing Software Defined Radio*. Sprinter Science+Business Media, 2013.

[18] HUSSAIN, H. M., BENKRID, K., ERDOGAN, A. T., AND SEKER, H. Highly parameterized k-means clustering on fpgas: Comparative results with gpps and gpus. In *2011 International Conference on Reconfigurable Computing and FPGAs* (2011), IEEE, pp. 475–480.

[19] JOLFAEI, F. A., MOHAMMADIZADEH, N., SADRI, M. S., AND FANISANI, F. High speed usb 2.0 interface for fpga based embedded systems. *Embedded and Multimedia Computing* (2009), 1–6.

[20] JULIALANG.ORG. Julia. http://julialang.org/, 2016. [online; Accessed on :October 2016].

[21] KAROLAK, D. W., AND KAROLAK, N. *Software engineering risk management: A just-in-time approach*. IEEE Computer Society Press, 1995.

[22] KIM, J., HYEON, S., AND CHOI, S. Implementation of an sdr system using graphics processing unit. *IEEE communications magazine 48*, 3 (2010), 156–162.

[23] KOTENG, R. M. Evaluation of sdr-implementation of ieee 802.15. 4 physical layer.

[24] LEE, C.-H., BASET, S. A., AND SCHULZRINNE, H. Tcp over udp.

[25] LILLINGTON, J. Flexible channelisation architectures for software defined radio front ends using the tuneable pipelined frequency transform. In *DSP enabled Radio, 2003 IEE Colloquium on* (2003), IET, pp. 1–13.

[26] LÖHNING, M., HENTSCHEL, T., AND FETTWEIS, G. Digital down conversion in software radio terminals. In *Signal Processing Conference, 2000 10th European* (2000), IEEE, pp. 1–4.

[27] LÓPEZ-PARRADO, A., AND VALDERRAMA-CUERVO, J.-C. Openrisc-based system-on-chip for digital signal processing. In *2014 XIX Symposium on Image, Signal Processing and Artificial Vision* (2014), IEEE, pp. 1–5.

[28] MARWANTO, A., SARIJARI, M. A., FISAL, N., YUSOF, S. K. S., AND RASHID, R. A. Experimental study of ofdm implementation utilizing gnu radio and usrp-sdr. In *Communications (MICC), 2009 IEEE 9th Malaysia International Conference on* (2009), IEEE, pp. 132–135.

[29] MECWAN, A., AND GAJJAR, N. Implementation of software defined radio on fpga. In *2011 Nirma University International Conference on Engineering* (2011), IEEE, pp. 1–5.

[30] MOHAPI, L. Design of an open-source tool-chain for the rhino platform. Master's thesis, University of Cape Town, 2016.

[31] MOHAPI, L. J., WINBERG, S., AND INGGS, M. A domain-specific language to facilitate software defined radio parallel executable patterns deployment on heterogeneous architectures. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)* (2014), IEEE, pp. 1–8.

[32] MOHAPI, L. J., WINBERG, S., AND INGGS, M. Using a domain specific language for sdr to facilitate radar signal processing in heterogeneous computing architectures. In *2015 IEEE Radar Conference* (2015), IEEE, pp. 306–311.

[33] NOLAN, K., SUTTON, P., DOYLE, L., RONDEAU, T., LE, B., AND BOSTIAN, C. Demonstration and analyses of collaboration, coexistence, and interoperability of cognitive radio platforms. In *Invited paper), in Proceedings of the 1st IEEE Workshop on Cognitive Radio Networks* (2007).

[34] PRABHAKAR, A., ET AL. *Design and prototype of an all digital system for baseband FM multiplex signal demodulation*. PhD thesis, 2015.

[35] RIGOL. Rigol mso1104 100 mhz digital oscilloscope. http://www.rigolna.com/products/digital-oscilloscopes/ds1000e/ds1102e/, November 2015. [online; Accessed on :November 2016].

[36] ROUPHAEL, T. J. *RF and Digital Signal Processing for Software-Defi ned Radio*. Elsevier Inc, 2009.

[37] RTL SDR.COM. Tutorial: Creating an fm receiver in gnu radio using an rtl-sdr source. http://www.rtl-sdr.com/tutorial-creating-fm-receiver-gnuradio-rtl-sdr/, November 2013. [online; Accessed on :August 2016].

[38] RTL SDR.COM. Rtl-sdr.com. http://www.rtl-sdr.com/, November 2016. [online; Accessed on :August 2016].

[39] SCIPY.ORG. About scipy. http://www.scipy.org/about.html, 2016. [online; Accessed on :October 2016].

[40] SCOTT, S. Reconfigurable hardware interface for computation and radio. Master's thesis, University of Cape Town, 2011.

[41] SHARAN, R., AND WEST, N. The comprehensive gnu radio archive network. http://www.cgran.org/, jul 2015. [Online; Accessed on: March 2016].

[42] SHEN, D. H., HWANG, C.-M., LUSIGNAN, B., AND WOOLEY, B. A 900 mhz integrated discrete-time filtering rf front-end. In *Solid-State Circuits Conference, 1996. Digest of Technical Papers. 42nd ISSCC., 1996 IEEE International* (1996), IEEE, pp. 54–55.

[43] SIGLENT. Sdg1000 series function/arbitrary waveform generators. http://www.siglent.com/ENs/generator/SDG1000, November 2015. [online; Accessed on :November 2016].

[44] SINGER, J., LETHBRIDGE, T., VINSON, N., AND ANQUETIL, N. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers* (2010), IBM Corp., pp. 174–188.

[45] SKLAR, B. *Digital communications*, vol. 2. Prentice Hall NJ, 2001.

[46] SO, H. K.-H., AND BRODERSEN, R. W. Improving usability of fpga-based reconfigurable computers through operating system support. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on* (2006), IEEE, pp. 1–6.

[47] STEVEN, W. S. The scientist and engineer's guide to digital signal processing. *California Technical Pub* (1997).

[48] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *ACM SIGMOD Record 34*, 4 (2005), 42–47.

[49] STREINER, D. L. Maintaining standards: differences between the standard deviation and standard error, and when to use each. *The Canadian journal of psychiatry 41*, 8 (1996), 498–502.

[50] TITLE =SOFTWARE DEFINED RESEARCHGROUP UCT, S. U. http://www.sdrg.ee.uct.ac.za/, Jul 2016. [Online; Accessed: September 2016].

[51] TONG, C., AND COETSER, J. A minimal architecture for real-time, medium range aircraft detection using fm-band illuminators of opportunity. In *2015 IEEE Radar Conference (RadarCon)* (May 2015), pp. 1250–1255.

[52] TRUONG, N. B., SUH, Y.-J., AND YU, C. Latency analysis in gnu radio/usrp-based software radio platforms. In *MILCOM 2013-2013 IEEE Military Communications Conference* (2013), IEEE, pp. 305–310.

[53] TSOEUNYANE, L. J. Tittle: Rhino software defined radio processing blocks. Master's thesis, University of Cape Town, 2015.

[54] VITTER, J. S. Efficient memory access in large-scale computation. In *Annual Symposium on Theoretical Aspects of Computer Science* (1991), Springer, pp. 26–41.

[55] WANG, S.-Y., CHAO, H.-L., LIU, K.-C., HE, T.-W., LIN, C.-C., AND CHOU, C.-L. Evaluating and improving the tcp/udp performances of ieee 802.11 (p)/1609 networks. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on* (2008), IEEE, pp. 163–168.

[56] XYLOMENOS, G., AND POLYZOS, G. C. Tcp and udp performance over a wireless lan. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (1999), vol. 2, IEEE, pp. 439–446.

[57] ZHANG, W., WEI, Z., HE, X., QIAO, P., AND LIANG, G. The design of high speed image acquisition system over gigabit ethernet. In *Wireless Communications, Networking and Information Security (WCNIS), 2010 IEEE International Conference on* (2010), Ieee, pp. 111–115.

APPENDIX A

RHINO STREAMER USER REQUIREMENTS QUESTIONNAIRE

1. What would be the realistic speed limit for sampling data on the RHINO board for streaming?

   1.1. Up to the full rate supported by the ADCs and DACs, times number of channels...ideal

   1.2. Ideally 50 MSPS

   1.3. Faster is better, but for that FPGA 200 MSps is about the limit...

   1.4. A streaming speed of 100MSps would be desirable, for 14-bit samples.

   1.5. 100 Msps

   1.6. 49.52 MSPS

2. What sampling rate do you need for projects planned for Rhino?

   2.1. I don't work with Rhino, but from other projects my experience is that about 25 MiB/s (Mebibytes / second) is practical and realistic.

   2.2. 100MSps would be desirable, however a speed of 10-20MSps could work if a suitable RF front end is arrange that will do digital down conversion.

   2.3. 100 Msps

3. What would be a tolerable latency period for streaming data from the board to the host PC? (i.e. the period from an instant data is sampled on RHINO to the time it's received on the processing software on the PC.)

   3.1. Not generally important.

3.2. Configuration latency (i.e. Setting ADC parameters, and first batch buffering - i.e. first packet) only may apply here, once streaming, we dont expect any latency. Configuration latency of 1-5 us

3.3. Latency is irrelevant. Put a time-stamp in the data header. A latency of under 100 ms would be nice though.

3.4. Could work with up to 100ms, that would be buffering possible 10M samples, there is plenty of RAM on RHINO to handle that amount of buffering.

3.5. Experiments needed to measure it. Hard to estimate.

3.6. ¡5us

4. The system can stream data continuously or in burst mode. In the case of burst mode, what would be an ideal period between the blocks of data streamed (period between bursts) ? would the blocks of data need to be received before processing starts or can processing start as soon as the first sample is received? Please also indicate in the case of continuous data streaming.

4.1. This is application specific - but continuous streaming with circular buffering is best for SDR/Radar

4.2. Processing generally starts after receiving the first PRI-related data packet. Each packet is generally 1 PRI long.

4.3. Ideally the period between blocks would be controllable. An appropriate period between blocks would be about 100ms, for example if transmitting a few radar bursts per second.

4.4. Can be determined thorough experimental tests. No actual figures as yet.

4.5. Continuous streaming every 1.5us at maximum packet size, less than that for smaller packet sizes (i.e. less 1500Bytes)

5. Elaborate on the type of processing to be done on the PC with the sampled data?

5.1. Not important

5.2. DSP algorithms such as Spectral analysis,

5.3. Everything except down-conversion. My development cycle generally involves testing the algorithms in Matlab on the PC before moving them down to the FPGA.

5.4. Ideally we would get 100MSps data coming in that can be digitally channelized, performing cognitive radio spectrum sensing. Utilization of the bands would be characterized to identify best opportunities for using channels.

5.5. Further DSP functions using MATLAB, GNU, etc and data storage

5.6. Further DSP, e.g. FM Demod, OFDM (Synchronization, decoding, demod, etc.), Radar Imaging and pulse compression,

6. Is the user likely to send processed data(i.e. not simply raw ADC samples) to the PC host application? If so please elaborate on the types of processed data that the user may be sending and ways that this data may need to be further processed (e.g. would the blocks need to be captured, are time stamps necessary), and how will the data be inspected?

6.1. Not really, as long as ADC data can be passed through the GbE without the need for multi-channel selections and/or down-sampling, raw ADC can be passed on to the host PC for further processing. Maybe DDCed or Polyphase Filter bank data for down-sampling and multi-channel system requirements respectively.

6.2. Yes. At the end of the development cycle, as much of the Matlab has been moved to the FPGA as possible. Typically, all points on my DSP chain is accessible by means of debugging streams. Only one stream active at a time though.

6.3. Yes. If the RHINO cannot send raw sample data fast enough in order to capture sufficient bandwidth, then the FFTs etc for performing spectrum analysis will need to be placed on the FPGA and sorter array describing how frequency the channel of interest are utilization will be send. Timestamps would de nitely be needed in this case.

6.4. Yes. Data being transmitted to air in analog format. Also configuration data for DSP blocks.

6.5. Channelized data - Spectral analysis will be used to inspect data

7. Please comment on the likelihood of needing to allow for more than one PC being involved in the processing chain. For example, is there a need for the PC receiving data from the RHINO to pass on potentially partly processed data for further processing?

7.1. Nice to have.

7.2. Not really, unless in the Radio Astronomy sense where Huge data streaming is required

7.3. Not in my experience, but this could be useful in future.

7.4. Yes, this would be useful, for example multiple PCs could be simultaneously receiving data and sending processing results to a central PC to combine and store/display the results.

7.5. Yes there is need. Some DSP functions may not t on RHINO.

7.6. No. One PC is sufficiently enough

8. Please elaborate on the type of sampled data that will be captured. What is the preferred format of sampled data?

8.1. FM signal, commensal Radar signal, L- and X-Band using bandpass sampling and/or analog down- sampling prior to ADC, basically any signal which can pass through the available ADC daughter board

8.2. 32-bit I/Q, interleaved. the 32-bit could be integers or single-precision floating point. After the FFT it's pointless to send only 16 bits, as the processing gain is generally good enough that the noise floor is way below the 16-bit quantisation noise. The sampled data is generally packetised by the FPGA, and then given a header with metadata.

8.3. Transmissions in the wifi ranges, 2.4GHz. Generally free / unlicensed spectrum bands.

8.4. 16-bit real samples or in I/Q format

8.5. A buffer of I/Q samples, 32byte, 255 bytes, 512bytes, 1.5kbytes etc.

9. Is there a need to store raw data streamed from the board or processed data? If so, what is likely number of samples, or processed data results that would to be store?

9.1. NextRAD data. From NetRAD, data sizes of 2048x130000 were stored in bin files

9.2. Both. Storing the raw data stream directly after down-conversion (or any other point of the DSP chain) to disk is immensely useful for debugging.

9.3. Yes, this would be very useful for trialling experimental algorithms, e.g. loading the saved samples into MATLAB to perform tests. Would probably not need to store more than 1Gbyte. Or a couple of seconds of recorded data.

9.4. It depends on the application but there is no limit. Only the storage capacity is a limit.

9.5. For RADAR yes, 130000x2048 or more

10. Is there need for facilities to replay sampled data? Please elaborate a likely scenario to help gain insight into what the user envision being a convenient approach to use such a feature.

10.1. Don't know what we mean by replay sampled data, but if we mean capture and play it back on the host PC in real-time, then Yes, for real-time spectral analysis, in surveillance Radar, etc.

10.2. Yes The data stream is stored to disk in the field. The user them brings the PC back to the lab and inject that stored data into the PC-based processing chain in order to optimise the algorithms.

10.3. This would be very useful. Ideally should have an option where the program performing processing can obtain sample data input either live streaming from the RHINO or played back from a file. The switch between the two should be quite seemless, needing minimal change to the code that uses the API to receive samples.

10.4. Don't know what we mean replay

11. Would there be need for mechanisms to visualise the incoming raw data? or to view real-time or post-real-time information about the samples being received?

11.1. Yes.

11.2. Yes. I generally have a Matlab-based data plotter that receives the raw data stream as fast as possible, and generates a new plot once every 100 ms (throwing away the intermediate packets).

11.3. Yes. It would be nice to have a very responsive, smooth scrolling, display showing the raw samples with an option to switch e.g. to frequency/FFT view that is also nice and smooth (i.e. the screen doesn't keep ashing each time the window is updated). This should work on either real-time or playback of recorded data. Would be nice to be able to control the speed of the played back data, e.g. specify a Samples/s value (could mean playing back recorded samples faster - or slower - than they were originally received).

11.4. Yes, visuals are the ultimate justification of the results obtained. This represent the information of data captured.

11.5. Yes, Spectral analysis, Radar imaging, Waterfall plots, etc.

12. Is there any particular visualisations needed of the processed data? e.g. FFT graphs?

12.1. Yes

12.2. Various needs. A plot of the packet header elds over packet number; the raw samples; the FFT of the raw samples, etc.

12.3. Bode Plot. Waterfall plot.

12.4. Yes. Same as above.

12.5. Yes - Same as above

13. Please provide any further suggestions that you have concerning the streaming framework and any features you would like to make it more useful for the user's needs.

13.1. Reconfiguration with new parameters (i.e. Fc, Fs, Streaming rate, etc.) on-the- fly.

13.2. Email me: TYLJOH010@myuct.ac.za

13.3. It would be useful to have a tutorial, example code showing how to use the framework.

13.4. Capture very high rate data at 100 Msps. Captured data must be error free. It must be easy to use. Do plots in real time such FFT, PSD etc.

13.5. Streaming back and forth the PC at Maximum streaming rate of 50MSPS.

APPENDIX B

QT C++ CODE

```
/*Implementing the Mainwindow class methods
*/

#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDebug>
#include <QDesktopWidget>
#include <QScreen>
#include <QMessageBox>
#include <QMetaEnum>
#include <algorithm>
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <iostream>
#include <math.h>
#include <sys/time.h>


//enable real time rhino data plotting
#define REALTIMEPLOT 1

//disable plotting from file
#define PLOTFILE 0
```

```
//disable RHINO time stamp plot
#define TIMESTAMP 0


MainWindow::MainWindow(QWidget *parent) :
  QMainWindow(parent),
  ui(new Ui::MainWindow)
{
  ui->setupUi(this);
  setGeometry(400, 250, 542, 390);


//setupQuadraticDemo(ui->customPlot); // Example Plot


#if REALTIMEPLOT
  if(s.Sconnect(9930))
  {
        setupRealTimeRhinoPlot(ui->customPlot);
  }
#endif

#if PLOTFILE
 filedataplot(ui->customPlot);
#endif

#if TIMESTAMP
 rhinotimestampPlot(ui->customPlot);
#endif

  setWindowTitle(demoName);

  statusBar()->clearMessage();

  ui->customPlot->replot();
```

```
}

void MainWindow::rhinotimestampPlot(QCustomPlot * customPlot)
{
demoName = "RHINO_DATA - LATENCY ";

double comm_time;
struct timeval t1,t2;
int len =12; //number of samples
QVector<double> time(len), samples(len);

std::vector<short> serverMessage;
if(s.Sconnect(9930))// Rhino Port number 9930
{
        printf("Connection successful\n");
}

for (int i=0; i<len; i++)
{
        gettimeofday(&t1,0);
        serverMessage=s.receive_data(pow(2,i));
        gettimeofday(&t2,0);

/*comm_time=((1000000.0*(t2.tv_sec-t1.tv_sec)) +
(t2.tv_usec-t1.tv_usec))/1000000.0; //in seconds*/

comm_time=((1000000.0*(t2.tv_sec-t1.tv_sec)) +
(t2.tv_usec-t1.tv_usec))/1000.0; //in milliseconds

/*comm_time=((1000000.0*(t2.tv_sec-t1.tv_sec)) +
(t2.tv_usec\\-t1.tv_usec)); //in microseconds*/

cout<<pow(2,i)<<" Samples > "<<comm_time<<" microseconds"<<endl;
//
time[i] = comm_time;
```

```
samples[i]= pow(2,i);

}
double Maxy = *max_element(time.begin(), time.end());

double Maxx = *max_element(samples.begin(), samples.end());
customPlot->addGraph();
customPlot->graph(0)->setData(samples, time);
// give the axes some labels:
customPlot->xAxis->setLabel("Number of Samples");
customPlot->yAxis->setLabel("Latency (msec)");
// set axes ranges, so we see all data: waller
customPlot->xAxis->setRange(0, Maxx);
customPlot->yAxis->setRange(0, Maxy);



}

void MainWindow::filedataplot(QCustomPlot * customPlot)
{

demoName = "RHINO_DATA";
std::ifstream out("FinalSin100.txt");
QVector<double> y(3000);

//int count=0;
//int size= x.size();
double d; int i=0;
while(out >> d)
{
        y[i] = d;
        i++;
}

out.close();
```

```
QVector<double> x(y.size());
for(int j=0;j<y.size();j++)
{ x[j]=j;}

double Maxy = *max_element(y.begin(), y.end());
double Miny = *min_element(y.begin(), y.end());
double Maxx= *max_element(x.begin(), x.end());

double Minx = *min_element(x.begin(), x.end());
cout<<"\n Maximum X Values: "<<Maxx<<"\n Maximum Y Values: "
<<Maxy<<endl;

customPlot->addGraph();
customPlot->graph(0)->setData(x, y);

// give the axes some labels:
customPlot->xAxis->setLabel("Samples");
customPlot->yAxis->setLabel("Amplitude");

// set axes ranges:
customPlot->xAxis->setRange(Minx, Maxx);
customPlot->yAxis->setRange(Miny, Maxy);

}


void MainWindow::setupRealTimeRhinoPlot(QCustomPlot *customPlot)
{


demoName = "Real Time RHINO Plot";

 customPlot->addGraph(); // blue line
 customPlot->graph(0)->setPen(QPen(Qt::blue));
 customPlot->graph(0)->setBrush(QBrush(QColor(240, 255, 200)));
```

```
 customPlot->graph(0)->setAntialiasedFill(false);
 customPlot->addGraph(); // red line
 customPlot->graph(1)->setPen(QPen(Qt::red));
 customPlot->graph(0)->setChannelFillGraph(customPlot->graph(1));

 customPlot->addGraph(); // blue dot
 customPlot->graph(2)->setPen(QPen(Qt::blue));
 customPlot->graph(2)->setLineStyle(QCPGraph::lsNone);
 customPlot->graph(2)->setScatterStyle(QCPScatterStyle::ssDisc);
 customPlot->addGraph(); // red dot
 customPlot->graph(3)->setPen(QPen(Qt::red));
 customPlot->graph(3)->setLineStyle(QCPGraph::lsNone);
 customPlot->graph(3)->setScatterStyle(QCPScatterStyle::ssDisc);

 customPlot->xAxis->setTickLabelType(QCPAxis::ltDateTime);
 customPlot->xAxis->setDateTimeFormat("hh:mm:ss");

 customPlot->xAxis->setAutoTickStep(false);
 customPlot->xAxis->setTickStep(2);
 customPlot->axisRect()->setupFullAxesBox();

// make left and bottom axes transfer their ranges to right
//and top axes:
 connect(customPlot->xAxis, SIGNAL(rangeChanged(QCPRange)),
customPlot->xAxis2, SLOT(setRange(QCPRange)));
 connect(customPlot->yAxis, SIGNAL(rangeChanged(QCPRange)),
 customPlot->yAxis2, SLOT(setRange(QCPRange)));

 /* setup a timer that repeatedly calls
  MainWindow::realtimeRhinoDataSlot:*/

 // Interval 0 means to refresh as fast as possible
connect(&dataTimer, SIGNAL(timeout()), this,
SLOT(realtimeRhinoDataSlot()));
dataTimer.start(0);
```

```
}
//

void MainWindow::realtimeRhinoDataSlot()
{
#if QT_VERSION < QT_VERSION_CHECK(4, 7, 0)
double key = 0;
#else
double key = QDateTime::currentDateTime().toMSecsSinceEpoch()
/1000.0;
#endif

std::vector<short> serverMessage;

serverMessage.resize(4);
QVector<double> x(2), y(2);// the first four values
static double lastPointKey = 0;

if (key-lastPointKey > 0.1)//at most add point every 100 ms
{

serverMessage=s.receive_data();

x[0] = (double)serverMessage[0];

y[0] = (double)serverMessage[1];

// add data to lines:
double value0 = x[0];
double value2 = y[0];

// add data to lines:
// first I value
ui->customPlot->graph(0)->addData(key, value0);
//First Q value
ui->customPlot->graph(1)->addData(key, value2);
```

```
// set data of dots:
ui->customPlot->graph(2)->clearData();
ui->customPlot->graph(2)->addData(key, value0);

ui->customPlot->graph(3)->clearData();
ui->customPlot->graph(3)->addData(key, value2);

// remove data of lines that's outside visible range:
ui->customPlot->graph(0)->removeDataBefore(key-8);
ui->customPlot->graph(1)->removeDataBefore(key-8);
// rescale value (vertical) axis to fit the current data:
ui->customPlot->graph(0)->rescaleValueAxis();

lastPointKey = key;

ui->customPlot->xAxis->setLabel("Time");
ui->customPlot->yAxis->setLabel("Amplitude");

}
// make key axis range scroll with the data
(at a constant range size of 8):
ui->customPlot->xAxis->setRange(key+0.25, 8, Qt::AlignRight);

ui->customPlot->replot();

}

//
void MainWindow::realtimeDataSlot()
{
  // calculate two new data points:
#if QT_VERSION < QT_VERSION_CHECK(4, 7, 0)
  double key = 0;
#else
  double key = QDateTime::currentDateTime().toMSecsSinceEpoch()
```

```
  /1000.0;
#endif
  static double lastPointKey = 0;
  //add point every 10 ms
  if (key-lastPointKey > 0.01)
  {
 //qSin(key*1.6+qCos(key*1.7)*2)*10 +qSin(key*1.2+0.56)*20+26;
 double value0 = qSin(key);

//qSin(key*1.3+qCos(key*1.2)*1.2)*7 +qSin(key*0.9+0.26)*24+26;
double value1 = qCos(key);

// add data to lines:
ui->customPlot->graph(0)->addData(key, value0);
ui->customPlot->graph(1)->addData(key, value1);

// set data of dots:
ui->customPlot->graph(2)->clearData();
ui->customPlot->graph(2)->addData(key, value0);
ui->customPlot->graph(3)->clearData();
ui->customPlot->graph(3)->addData(key, value1);

// remove data of lines that's outside visible range:
ui->customPlot->graph(0)->removeDataBefore(key-8);
ui->customPlot->graph(1)->removeDataBefore(key-8);

// rescale value (vertical) axis to fit the current data:
ui->customPlot->graph(0)->rescaleValueAxis();
ui->customPlot->graph(1)->rescaleValueAxis(true);
lastPointKey = key;
}

// make key axis range scroll with the data
(at a constant range size of 8):
ui->customPlot->xAxis->setRange(key+0.25, 8, Qt::AlignRight);
ui->customPlot->replot();
```

```
// calculate frames per second:
static double lastFpsKey;
static int frameCount;
++frameCount;
if (key-lastFpsKey > 2) // average fps over 2 seconds
{
 ui->statusBar->showMessage(
 QString("%1 FPS, Total Data points: %2").arg(frameCount/
 (key-lastFpsKey),0, 'f', 0).arg(ui->customPlot->graph(0)
 ->data()->count()+ui->customPlot->graph(1)->data()->count()), 0);

lastFpsKey = key;
frameCount = 0;

  }
}
//
void MainWindow::setupPlayground(QCustomPlot *customPlot)
{
  Q_UNUSED(customPlot)
}

MainWindow::~MainWindow()
{
  delete ui;
}
//
```

APPENDIX C

CHIRP SIGNAL

C.1 CHIRP.CPP

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gnuradio/io_signature.h>
#include "chirp_signal_f_impl.h"
#include <cmath>
#include <iostream>
#include <vector>
#include <string>
#include <math.h>

namespace gr {
  namespace MpatiBlocks {

    chirp_signal_f::sptr
    chirp_signal_f::make(float phase,float freq_0,float freq_1,
    float time_1)
    {
      return gnuradio::get_initial_sptr
        (new chirp_signal_f_impl(phase,freq_0,freq_1,time_1));
    }
```

```
/*
 * The private constructor
 */
chirp_signal_f_impl::chirp_signal_f_impl(float phase,float freq_0,
float freq_1,float time_1)
  :gr::sync_block("chirp_signal_f",
   gr::io_signature::make(0, 0, 0),
   gr::io_signature::make(1, 1,sizeof(float))),f_phase(phase),
   f_freq_0(freq_0),f_freq_1(freq_1),
   f_time_1(time_1)
{}

/*
 * Our virtual destructor.
 */
chirp_signal_f_impl::~chirp_signal_f_impl()
{
}

int
chirp_signal_f_impl::work(int noutput_items,
                          gr_vector_const_void_star &input_items,
                          gr_vector_void_star &output_items)
{
 float *out = (float *) output_items[0];

     std::vector<float>  t;// = 0.0:0.001:5.0;
     int L = 5000;  //number of samples
     t.resize(L);   // time interval

     //calculations of attributes
     float phase = 2*(M_PI*f_phase)/360.0f;
     float a = (M_PI *(f_freq_1-f_freq_0))/f_time_1;
     float b = 2*M_PI*f_freq_0;
     float tindexer = 0.0f;
```

```
        //
                for(int i = 0; i< L; i++)
                {
                        t[i] = tindexer;
                        tindexer = tindexer + 0.001f;
                }

                for(int i = 0; i < L; i++) // or L instead of noutput
                {
                        out[i] = cos(a*pow(t[i],2)+(b*t[i])+phase);
                }
                return noutput_items;
    }

void
chirp_signal_f_impl::set_phase(float phase)
{
  f_phase = phase;
}

void
chirp_signal_f_impl::set_start_freq(float frequency)
{
  f_freq_0= frequency;
}

void
chirp_signal_f_impl::set_freq_1(float freq)
{
  f_freq_1=freq;

}

void
chirp_signal_f_impl::set_time(float t)
{
```

```
      f_time_1 = t;
    }
  } /* namespace MpatiBlocks */
} /* namespace gr */
```

C.2 CHIRP.H

```
#ifndef INCLUDED_MPATIBLOCKS_CHIRP_SIGNAL_F_IMPL_H
#define INCLUDED_MPATIBLOCKS_CHIRP_SIGNAL_F_IMPL_H

#include <MpatiBlocks/chirp_signal_f.h>

namespace gr {
  namespace MpatiBlocks {

    class chirp_signal_f_impl : public chirp_signal_f
    {
     private:
      float f_phase;
      float f_freq_0;
      float f_freq_1;
      float f_time_1;


     public:
      chirp_signal_f_impl(float f_phase,float f_freq_0,
      float f_freq_1,float f_time_1);
      ~chirp_signal_f_impl();

      // Where all the action really happens
        int work(int noutput_items,
               gr_vector_const_void_star &input_items,
               gr_vector_void_star &output_items);


         float starting_freq() const {return f_freq_0;}
         float frequency_1() const {return f_freq_1;}
```

```
        float time_1() const {return f_time_1;}
     float phase() const {return f_phase;}


        void set_phase(float phase_0);
        void set_start_freq(float freq);
        void set_freq_1(float _freq);
        void set_time(float time);

    };

  } // namespace MpatiBlocks
} // namespace gr

#endif /* INCLUDED_MPATIBLOCKS_CHIRP_SIGNAL_F_IMPL_H */
```

APPENDIX D

RHINO UDP SOURCE BLOCK FOR GNU RADIO

D.1 RHINO-SOURCE.CPP

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <iostream>
#include <gnuradio/gr_complex.h>
#include <algorithm>
#include <stdexcept>
#include <complex>
#include <volk/volk.h>
#include <gnuradio/io_signature.h>
#include "RHINO_c_2_impl.h"

static const int64_t MAX_INT =  2147483647; //  (2^31)-1
static const int64_t MIN_INT = -2147483647; // -(2^31)-1

namespace gr {
  namespace MpatiBlocks {

RHINO_c_2::sptr
RHINO_c_2::make(size_t itemsize,int port,int payLoad)
{
```

```
  return gnuradio::get_initial_sptr
    (new RHINO_c_2_impl(itemsize, port, payLoad));
}




// The private constructor

RHINO_c_2_impl::RHINO_c_2_impl(size_t itemsize,
int port,int payLoad): gr::sync_block("RHINO_c",
 gr::io_signature::make(0, 0, 0),
 gr::io_signature::make(1, 1,itemsize)),
 d_itemsize(itemsize),c_port(port),c_payLoad(payLoad)
{

  Sconnect(c_port);
}

 /*
   Our virtual destructor.
*/

RHINO_c_2_impl::~RHINO_c_2_impl()
{
}



int RHINO_c_2_impl::work(int noutput_items,
        gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
{

   std::vector<short> serverMessage;
   serverMessage.resize(2);
```

```
   int num_bytes_to_send = (c_payLoad/d_itemsize);
   int length= serverMessage.size();

if(type==Short)
{

 short *out = (short *) output_items[0]; // |I|Q|0|0|
 for( int i =0;i<num_bytes_to_send;i++)
 {
       serverMessage=receive_data();
       out[i]=(short)serverMessage[0];
 }
}

else if(type==Int)
{

 int32_t  *out =(int32_t *) output_items[0]; // |I|Q|0|0|
 for( int i =0;i<num_bytes_to_send;i++)
 {
       serverMessage=receive_data();
       int32_t r = llrintf(serverMessage[0]);

       if (r < MIN_INT)
                r = MIN_INT;
       else if (r > MAX_INT)
                r = MAX_INT;

       out[i] = static_cast<int>(r);
       }
}

 else if(type==Float)
 {

       float *out = (float*) output_items[0]; // |I|Q|0|0|
```

```
        for( int i =0;i<num_bytes_to_send;i++)
        {
          serverMessage=receive_data();
          //take in the first value, the I value
         out[i]=(float)serverMessage[0];

        }

        }

  else if(type==Complex) // Similar to Default
  {
   |I|Q|0|0|
   gr_complex *out = (gr_complex *) output_items[0]; //
   for( int i =0;i<num_bytes_to_send;i=+2)
   {
        serverMessage=receive_data();
        out[i]=gr_complex((float)serverMessage[0],
         (float)serverMessage[1]);
   }
  }
  else
  {
    // |I|Q|0|0|
        gr_complex *out = (gr_complex *) output_items[0];
        for( int i =0;i<num_bytes_to_send;i=+2)
        {
          serverMessage=receive_data();
          out[i]=gr_complex((float)serverMessage[0],
           (float)serverMessage[1]);
        }

  }

        // Tell runtime system how many output items we produced.
return num_bytes_to_send;
```

```
}


bool RHINO_c_2_impl::Sconnect(int port)
{

  if((udpsocket=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP))<0)
  {
    //[ERROR] Could not Create Socket. Please try Again.
        exit(1);
  }

        memset((char*)&rudpserver,0,sizeof(rudpserver));
        rudpserver.sin_family = AF_INET;
        rudpserver.sin_port = htons(port);
        rudpserver.sin_addr.s_addr = htonl(INADDR_ANY);

  if(bind(udpsocket,(struct sockaddr*)&rudpserver,
  sizeof(rudpserver))<0)
  {
        exit(1); //[ERROR] Could not Bind Socket
  }

        return true;
  }

std::vector<short> RHINO_c_2_impl::receive_data()
{

        std::vector<short> IQdata;
        IQdata.resize(2);

        int rudpclient_len = sizeof(rudpclient);
        struct rhinodata indata;
        int count = 0;
```

```
        int cres;

cres = recvfrom(udpsocket,(char*)&indata,sizeof(struct rhinodata)
,0,(struct sockaddr *)&rudpclient,(socklen_t *)&rudpclient_len);
if(cres<0)
{
  //[ERROR] Problem Receiving Data. Please try Again
  return IQdata;
}

        // Packet has a pair of I Q values
        IQdata[count] = (indata.I1>>8) + (indata.I1<<8);
        IQdata[count+1] = (indata.Q1>>8) + (indata.Q1<<8);

        return IQdata;
}


  void RHINO_c_2_impl::set_port(int port)
  {
         c_port=port;
  }


  void RHINO_c_2_impl::set_payLoad(int payLoad)
  {
          c_payLoad=payLoad;
  }

  } /* namespace MpatiBlocks */
} /* namespace gr */
```

D.2 RHINO-SOURCE.H

```
#ifndef INCLUDED_MPATIBLOCKS_RHINO_C_2_IMPL_H
#define INCLUDED_MPATIBLOCKS_RHINO_C_2_IMPL_H
```

```
#include <MpatiBlocks/RHINO_c_2.h>
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <string>
#include <vector>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fstream>


namespace gr {
  namespace MpatiBlocks {


          struct rhinodata
         {
         short I1;
         short Q1;

         };

         enum Type
         {
                 Complex,
                 Float,
                 Int,
                 Short,
                 Complex2

         };
```

```
class RHINO_c_2_impl : public RHINO_c_2
{
 private:
    size_t  d_itemsize;
            int c_port;
            int c_payLoad;
            int udpsocket;
            int clen;
            struct rhinodata indata[100];
            struct sockaddr_in rudpserver;
            struct sockaddr_in rudpclient;
            Type type;

 public:
  RHINO_c_2_impl(size_t itemsize,int port,int payLoad);
  ~RHINO_c_2_impl();


       bool Sconnect(int);

       std::vector<short> receive_data();

  // Where all the action really happens
  int work(int noutput_items,
           gr_vector_const_void_star &input_items,
           gr_vector_void_star &output_items);


     int port_number () const {return c_port;}
     int payLoad_size () const {return c_payLoad;}


     void set_port(int port);
     void set_payLoad(int payLoad);
};
```

```
  } // namespace MpatiBlocks
} // namespace gr

#endif /* INCLUDED_MPATIBLOCKS_RHINO_C_2_IMPL_H */
```

APPENDIX E

CREATING OOT MODULES

E.1 CREATING A NEW MODULE

1. Use gr mod tool to create a new module
$ gr_modtool newmod [folder]

2. Enter Module directory
$ cd gr-[folder]

3. Adding a new block
$ gr_modtool add -t [type_block] [name_of_block]

where [name_of_block] = RHINO_f and [type_block] = source

4. Answer the following as follows:
4.0 Specify Language
Language: C++

4.1 Block Identifier
Block/code identifier: [block_name]

4.2 Entering parameter list
Enter valid argument list, including default arguments: [type] [name], [another_type] [another_name], etc.

4.3 QA
Add Python QA code? [Y/n] n
Add C++ QA code? [y/N] n

E.2 Creating cmake

```
$ mkdir build          # We're currently in the module's top directory
$ cd build/
$ cmake ../          # Tell CMake that all its config files are one dir up
$ make          # And start building (should work after the previous section)
```

E.3 Creating XML File

/gr-howto$ gr_modtool makexml [name_of_block]

GNU Radio module name identified: [name_of_module]

Overwrite existing GRC file? [Y/N] yes

E.4 Installing The Block in GRC

```
$ cd build/
$ make install
$ sudo ldconfig
```

E.5 Deleting a Block From Own Tree

$gr_modtool remove [block_name]

Reference to the .cpp and .h files to the block need to be manually removed from all the cmake files.

>>>to view different options for gr_modtool type the following : gr_modtool help